

As per
JNTU-Kakinada
Syllabus Regulation
2016

COMPUTER PROGRAMMING

About the Author

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

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

- *Programming in ANSI C, 7/e*
- *Fundamentals of Computers*
- *Computing Fundamentals and C Programming*
- *Programming in Java, 5/e*
- *Programming in BASIC, 3/e*
- *Programming in C#, 3/e*
- *Numerical Methods*
- *Reliability Engineering*
- *Introduction to Computing and Problem Solving using Python, 1e*

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

As per
JNTU-Kakinada
Syllabus Regulation
2016

COMPUTER PROGRAMMING

E Balagurusamy

*Chairman
EBG Foundation
Coimbatore*



McGraw Hill Education (India) Private Limited

CHENNAI

McGraw Hill Education Offices

Chennai 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
444/1, Sri Ekambara Naicker Industrial Estate, Alapakkam, Porur, Chennai - 600 116

Computer Programming

Copyright © 2017 by McGraw Hill Education (India) Private Limited.

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

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

Print Edition:

ISBN-13: 978-93-5260-388-6

ISBN-10: 93-5260-388-5

Managing Director: *Kaushik Bellani*

Director—Products (Higher Education and Professional): *Vibha Mahajan*

Manager—Product Development: *Koyel Ghosh*

Specialist—Product Development: *Sachin Kumar*

Head—Production (Higher Education and Professional): *Satinder S Baveja*

Assistant Manager—Production: *Jagriti Kundu*

AGM—Product Management (Higher Education and Professional): *Shalini Jha*

Manager—Product Management: *Kartik Arora*

General Manager—Production: *Rajender P Ghansela*

Manager—Production: *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 The Composers, 260, C.A. Apt., Paschim Vihar, New Delhi 110 063 and printed at

Cover Printer:

Visit us at: www.mheducation.co.in

Contents

Preface

xiii

Roadmap to the Syllabus

xvi

Chapter 1 Introduction to Computers and Programming 1.1–1.30

- 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 History of C 1.12
- 1.4 Data Types 1.13
 - 1.4.1 Integer Types 1.14
 - 1.4.2 Floating Point Types 1.15
 - 1.4.3 Void Types 1.16
 - 1.4.4 Character Types 1.16
- 1.5 Programming Languages 1.16
 - 1.5.1 Machine Language (Low Level Languages) 1.16
 - 1.5.2 Assembly Language (Symbolic Language) 1.16
 - 1.5.3 High-Level Languages 1.18
- 1.6 Development of C Algorithms 1.19
 - 1.6.1 Characteristics of Algorithms 1.20
 - 1.6.2 Advantages of Algorithms 1.20
 - 1.6.3 Disadvantages of Algorithms 1.20
- 1.7 Software Development Method 1.22
 - 1.7.1 Analysing the Requirements 1.23
 - 1.7.2 Feasibility Analysis 1.23
 - 1.7.3 Creating the Design 1.24
 - 1.7.4 Developing Code 1.24
 - 1.7.5 Testing the Software 1.25
 - 1.7.6 Deploying the Software 1.25
 - 1.7.7 Maintaining the Software 1.25
- 1.8 Applying Software Development Method 1.25
- Key Terms* 1.27
- Just Remember* 1.28
- Multiple Choice Questions* 1.28

Answers 1.29

Review Questions 1.29

Chapter 2 Basics of C 2.1–2.90

- 2.1 Importance of C 2.1
- 2.2 Basic Structure of C Programs 2.1
- 2.3 Programming Style 2.2
- 2.4 Executing a ‘C’ Program 2.3
- 2.5 Sample Programs 2.4
 - 2.5.1 Sample Program 1: Printing a Message 2.4
 - 2.5.2 Sample Program 2: Adding Two Numbers 2.6
 - 2.5.3 Sample Program 3: Interest Calculation 2.8
 - 2.5.4 Sample Program 4: Use of Subroutines 2.10
 - 2.5.5 Sample Program 5: Use of Math functions 2.10
- 2.6 C Character Set 2.12
 - 2.6.1 Trigraph Characters 2.13
- 2.7 C Tokens 2.14
- 2.8 Keywords and Identifiers 2.14
- 2.9 Operators and Expressions 2.15
 - 2.9.1 Arithmetic Operators 2.15
 - 2.9.2 Relational Operators 2.18
 - 2.9.3 Logical Operators 2.19
 - 2.9.4 Assignment Operators 2.20
 - 2.9.5 Increment and Decrement Operators 2.22
 - 2.9.6 Conditional Operator 2.23
 - 2.9.7 Bitwise Operators 2.25
 - 2.9.8 Special Operators 2.25
 - 2.9.9 Operator Precedence 2.27
 - 2.9.10 Precedence of Arithmetic Operators 2.29
 - 2.9.11 Some Computational Problems 2.30
 - 2.9.12 Type Conversions in Expressions 2.31
 - 2.9.13 Operator Precedence and Associativity 2.34
- 2.10 Constants 2.36
 - 2.10.1 Integer Constants 2.37
 - 2.10.2 Real Constants 2.38
 - 2.10.3 Single Character Constants 2.38
 - 2.10.4 String Constants 2.39
- 2.11 Variables 2.40
- 2.12 Declaration of Variables 2.41
 - 2.12.1 Primary Type Declaration 2.41
 - 2.12.2 User-defined Type Declaration 2.42
 - 2.12.3 Declaration of Storage Class 2.43
 - 2.12.4 Assigning Values to Variables 2.44

2.13	ANSI C Library Functions	2.50
2.14	Managing Input and Output Operations	2.53
2.14.1	Reading a Character	2.54
2.14.2	Writing a Character	2.56
2.14.3	Formatted Input	2.58
2.14.4	Points to Remember while Using scanf	2.65
2.14.5	Formatted Output	2.66
2.15	Case Studies	2.71
	<i>Key Terms</i>	2.78
	<i>Just Remember</i>	2.79
	<i>Multiple Choice Questions</i>	2.80
	<i>Answers</i>	2.84
	<i>Review Questions</i>	2.84
	<i>Debugging Exercises</i>	2.86
	<i>Programming Exercise</i>	2.87

Chapter 3 Decision Making, Branching and Looping 3.1–3.70

3.1	Introduction	3.1
3.2	Decision Making with If Statement	3.1
3.2.1	Simple If Statement	3.2
3.2.2	The If....Else Statement	3.6
3.2.3	Nesting of If....Else Statements	3.9
3.2.4	The Else If Ladder	3.11
3.3	Decision Making with Switch Statement	3.15
3.4	The ? : Operator	3.20
3.5	Decision Making with Goto Statement	3.22
3.6	Introduction to Looping Procedure	3.25
3.6.1	Sentinel Loops	3.26
3.7	The While Statement	3.27
3.8	The Do Statement	3.29
3.9	The For Statement	3.33
3.9.1	Simple 'for' Loops	3.33
3.9.2	Additional Features of For Loop	3.37
3.9.3	Nesting of For Loops	3.39
3.10	Jumps In Loops	3.43
3.10.1	Jumping Out of a Loop	3.43
3.11	Case Studies	3.45
	<i>Key Terms</i>	3.56
	<i>Just Remember</i>	3.57
	<i>Multiple Choice Questions</i>	3.58
	<i>Answers</i>	3.60
	<i>Review Questions</i>	3.61
	<i>Debugging Exercises</i>	3.65
	<i>Programming Exercises</i>	3.66

Chapter 4	User-Defined Functions	4.1–4.52
4.1	Introduction	4.1
4.2	Need for User-Defined Functions	4.1
4.3	A Multi-Function Program	4.2
4.3.1	Modular Programming	4.4
4.4	Category of Functions	4.5
4.4.1	No Arguments and No Return Values	4.5
4.4.2	Arguments but No Return Values	4.7
4.4.3	Arguments with Return Values	4.10
4.4.4	No Arguments but Returns a Value	4.16
4.4.5	Functions that Return Multiple Values	4.16
4.4.6	Nesting of Functions	4.17
4.5	Elements of User-Defined Functions	4.19
4.6	Definition of Functions	4.19
4.6.1	Function Header	4.20
4.6.2	Name and Type	4.20
4.6.3	Formal Parameter List	4.20
4.6.4	Function Body	4.21
4.7	Return Values and their Types	4.21
4.8	Function Calls	4.22
4.8.1	Function Call	4.24
4.9	Function Declaration	4.24
4.9.1	Prototypes: Yes or No	4.25
4.9.2	Parameters Everywhere!	4.25
4.10	Recursion	4.26
4.10.1	Recursion versus Iteration	4.27
4.11	Passing Arrays to Functions	4.27
4.11.1	One-Dimensional Arrays	4.27
4.11.2	Two-Dimensional Arrays	4.31
4.12	Passing Strings to Functions	4.32
4.12.1	Pass by Value versus Pass by Pointers	4.32
4.13	The Scope, Visibility, and Lifetime of Variables	4.33
4.13.1	Automatic Variables	4.33
4.13.2	External Variables	4.35
4.13.3	External Declaration	4.37
4.13.4	Static Variables	4.39
4.13.5	Register Variables	4.40
4.14	Multifile Programs	4.42
4.15	Case Study	4.43
	<i>Key Terms</i>	4.46
	<i>Just Remember</i>	4.46
	<i>Multiple Choice Questions</i>	4.47

Answers 4.48

Review Questions 4.49

Debugging Exercises 4.51

Programming Exercises 4.51

Chapter 5

Arrays

5.1–5.44

5.1 Introduction 5.1

5.1.1 Data Structures 5.2

5.2 One-Dimensional Arrays 5.2

5.3 Declaration of One-dimensional Arrays 5.3

5.4 Initialization of One-dimensional Arrays 5.6

5.4.1 Compile Time Initialization 5.6

5.4.2 Run Time Initialization 5.7

5.4.3 Searching and Sorting 5.11

5.5 Two-Dimensional Arrays 5.12

5.6 Initializing Two-Dimensional Arrays 5.16

5.6.1 Memory Layout 5.19

5.7 Multi-Dimensional Arrays 5.25

5.8 Dynamic Arrays 5.26

5.9 Case Studies 5.26

Key Terms 5.38

Just Remember 5.38

Multiple Choice Questions 5.39

Answers 5.40

Review Questions 5.40

Debugging Exercises 5.41

Programming Exercises 5.42

Chapter 6

Strings

6.1–6.34

6.1 Introduction 6.1

6.2 Declaring and Initializing String Variables 6.2

6.3 Reading Strings from Terminal 6.3

6.3.1 Using scanf Function 6.3

6.3.2 Reading a Line of Text 6.5

6.3.3 Using getchar and gets Functions 6.6

6.4 Writing Strings to Screen 6.11

6.4.1 Using printf Function 6.11

6.4.2 Using putchar and puts Functions 6.14

6.5 Arithmetic Operations on Characters 6.15

6.6 Putting Strings Together 6.16

6.7 Comparison of Two Strings 6.18

6.8 String-Handling Functions 6.18

6.8.1 strcat() Function 6.18

6.8.2	strcmp() Function	6.19
6.8.3	strcpy() Function	6.20
6.8.4	strlen() Function	6.20
6.8.5	Other String Functions	6.22
6.9	Table of Strings	6.24
6.10	Case Studies	6.26
	<i>Key Terms</i>	6.30
	<i>Just Remember</i>	6.30
	<i>Multiple Choice Questions</i>	6.30
	<i>Answers</i>	6.31
	<i>Review Questions</i>	6.31
	<i>Debugging Exercises</i>	6.33
	<i>Programming Exercises</i>	6.33

Chapter 7 Pointers

7.1–7.42

7.1	Introduction	7.1
7.2	Understanding Pointers	7.2
7.2.1	Underlying Concepts of Pointers	7.3
7.3	Initialization of Pointer Variables	7.3
7.3.1	Pointer Flexibility	7.4
7.4	Declaring Pointer Variables	7.5
7.4.1	Pointer Declaration Style	7.5
7.5	Accessing the Address of a Variable	7.6
7.6	Accessing a Variable Through its Pointer	7.8
7.7	Chain of Pointers	7.10
7.8	Pointer Expressions	7.11
7.9	Pointer Increments and Scale Factor	7.12
7.9.1	Rules of Pointer Operations	7.13
7.10	Pointers as Function Arguments	7.13
7.11	Functions Returning Pointers	7.16
7.12	Pointers to Functions	7.17
7.12.1	Compatibility and Casting	7.19
7.13	Pointers and Arrays	7.19
7.14	Pointers and Character Strings	7.23
7.15	Array of Pointers	7.25
7.16	Dynamic Memory Allocation	7.26
7.17	Allocating a Block of Memory: Malloc	7.27
7.18	Allocating Multiple Blocks of Memory: Calloc	7.29
7.19	Releasing the Used Space: Free	7.33
7.20	Case Studies	7.33
	<i>Key Terms</i>	7.38
	<i>Just Remember</i>	7.38
	<i>Multiple Choice Questions</i>	7.39

Answers 7.40

Review Questions 7.40

Debugging Exercises 7.42

Programming Exercise 7.42

Chapter 8 Structures and Unions **8.1–8.59**

8.1 Introduction 8.1

8.2 Defining a Structure 8.1

8.3 Declaring Structure Variables 8.2

8.3.1 Accessing Structure Members 8.4

8.4 Structure Initialization 8.5

8.5 Arrays of Structures 8.8

8.5.1 Arrays Within Structures 8.11

8.5.2 Structures Within Structures 8.13

8.6 Structures and Functions 8.15

8.6.1 Passing Structure Through Pointers 8.17

8.6.2 Self Referential Structure 8.18

8.7 Pointers and Structures 8.18

8.8 Unions 8.21

8.9 Bit Fields 8.23

8.10 Typedef 8.25

8.11 Command Line Arguments 8.37

8.11.1 Application of Command Line Arguments 8.48

8.12 Case Study 8.50

Key Terms 8.53

Just Remember 8.53

Multiple Choice Questions 8.54

Answers 8.54

Review Questions 8.55

Debugging Exercises 8.57

Programming Exercise 8.58

Chapter 9 Data Files **9.1–9.22**

9.1 Introduction 9.1

9.2 Defining and Opening a File 9.2

9.3 Closing a File 9.3

9.4 Input/Output Operations on Files 9.4

9.4.1 The `getc` and `putc` Functions 9.4

9.4.2 The `getw` and `putw` Functions 9.8

9.4.3 The `fprintf` and `fscanf` Functions 9.10

9.5 Error Handling During I/O Operations 9.12

9.6 Random Access to Files 9.14

Key Terms 9.20

xii *Contents*

Just Remember 9.20

Multiple Choice Questions 9.21

Answers 9.21

Review Questions 9.21

Debugging Exercise 9.22

Programming Exercise 9.22

Appendix 1 C99/C11 Features

A1.1–A1.8

Solved Question Paper Nov-Dec 2015 (Set 1– Set 4)

SQP1–SQP32

Solved Question Paper May 2016 (Set 1– Set 4)

SQP1–SQP29

Preface

INTRODUCTION

Computers play an increasingly important role in today's world and a sound knowledge of computers has become indispensable for anyone who seeks employment not only in the area of IT but also in any other field as well. Computer programming is dedicated to the understanding of computer language, and writing and testing of programs that computers follow to perform their functions. The programs are created using programming languages and C is the most prevalent, efficient and compact programming language. 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. Thus, many institutions and universities in India have introduced a subject covering Computer Programming.

This book is specially designed for first-year students of Jawaharlal Nehru Technological University Kakinada (JNTU K) and would enable them to master the necessary skills for programming with C language. The text has been infused with numerous examples and case studies to empower the learner. Furthermore, the book also covers design and implementation aspect of data structures using standard ANSI C programming language.

SIGNIFICANT FEATURES

- **New!** Completely in sync with the syllabus of JNTU Kakinada (2016 Regulation)
- **New!** Incorporates all the features of ANSI C that are essential for a C programmer.
- **New!** Solutions to latest 2015 (Nov/Dec) and 2016 (May) JNTU Kakinada question paper is placed at the end of the book (All 4 sets)
- **New! 149 Multiple Choice Questions** incorporated at the end of each chapter help students test their conceptual understanding of the subject
- **22 Case Studies** in relevant chapters with stepwise solution to demonstrate real-life applications
- **New!** Updated information on **C99/C11 features**
- **New!** Topics like ANSI C library functions, Negation, Swapping Values, Recursion v/s Iteration are covered in detail
- **Learning by example** approach ensures smooth and successful transition from a learner to a skilled C programmer
- Enhanced student-friendly chapter design including *Outline, Introduction, Section-end Solved Programs, Case Studies, Key Terms, Just Remember, Multiple Choice Questions, Review Questions, Debugging Exercises, Programming Exercises*
- Special box feature highlighting supplementary information that complements the text.

PEDAGOGICAL FEATURES

- **134 Solved C Programs** demonstrate the general principles of good programming style
- **171 Review Questions** helps in testing conceptual understanding
- **28 Debugging Exercise** helps in participating coding contests
- **179 Programming Exercises** simulate interest to practice programming applications

CHAPTER ORGANIZATION

The content is spread across 9 chapters. **Chapter 1** introduces computer systems, programming languages and environment, software development method, and algorithms. **Chapter 2** gives an overview of C and explaining the keywords, identifiers, constants, variables, data types and various case studies on these. **Chapters 3** comprises of decision-making, branching and looping methods. **Chapter 4** covers the functions which are used in C language. **Chapter 5** focuses on arrays while **Chapter 6** deals with strings. Different types of pointers and its types are discussed in **Chapter 7**. **Chapter 8** presents structures and unions while **Chapter 9** covers file types and their management. **Appendix 1** covers C99/C11 features in detail. In addition to all this, **Solved Question Papers** of Nov/Dec 2015 (4 sets) and May 2016 (4 sets) are also given in this book.

CD RESOURCES

The supplementary CD provided along with the book would help the students master programming language and write their own programs using Computer programming concepts and data structures. The CD comprises of the following resources:

- **New!** 2012, 2013, 2014, Jan/Feb 2015 solved question papers
- **New! Lab Programs** as per the new syllabus
- Two major programming projects—*Inventory* and *Record Entry* & two mini projects—*Linked List* and *Matrix Multiplication*
- **100 Programming Exercises** and **200 Objective Type Questions** aligned as per the new syllabus
- 5 Solved Model Question Papers
- 79 Additional Solved Programs
- Additional content on Matrix Operation

ACKNOWLEDGEMENTS

A number of reviewers took pains to provide valuable feedback for the book. We are grateful to all of them and their names are mentioned as follows:

S. Krishna Rao

Sir CR Reddy College of Engineering, Eluru, Andhra Pradesh

Narasimha Rao Kandula

Vishnu Institute of Technology, Bhimavaram, Andhra Pradesh

K. Phani Babu, Chundru Raja Ramesh

Sri Vasavi Engineering College, Tadepalligudem, Andhra Pradesh

Rama Rao Adimalla

Lendi Institute of Engineering and Technology, Jonnada,
Andhra Pradesh

S. Rama Sree

Aditya Engineering College, Peddapuram, Andhra Pradesh

M V S S Nagendranath

Sasi Institute of Technology & Engineering, Tadepalligudem,
Andhra Pradesh

S. Satyanarayana

Raghu Engineering College, Dakamarri, Andhra Pradesh

S C Satapathy

Anil Neerukonda Institute of Technology and Sciences,
Visakhapatnam, Andhra Pradesh

Ch Vijaya Kumar

DVR & Dr. HS MIC College of Technology, Kanchikacherla,
Andhra Pradesh

E Balagurusamy

Publisher's Note

McGraw Hill Education (India) invites suggestions and comments, all of which can be sent to info.india@mheducation.com (kindly mention the title and author name in the subject line).

Piracy-related issues may also be reported.

Roadmap to the Syllabus

Computer Programming

Revised Course from Academic Year 2016-2017

Unit 1: History and Hardware—Computer hardware, Bits and bytes, Components, Programming Languages—machine language, assembly language, low-level and high-level languages, procedural and object-oriented languages, Application and system software, Development of C algorithms, Software development process



Go to

Chapter 1: Introduction to Computers and Programming

Unit 2: Introduction to C programming, Identifiers, main () function, printf () function, Programming style, Indentation, Comments, Data types, Arithmetic operations, Expression types, Variables and declarations, Negation, Operator precedence and associativity, Declaration statements, Initialization assignment, Implicit type conversions, Explicit type conversions, Assignment variations, Mathematical library functions, Interactive input, Formatted output, Format modifiers



Go to

Chapter 1: Introduction to Computers and Programming

Chapter 2: Basics of C

Unit 3: Control flow-relational expressions—logical operators, Selection—if-else statement—nested if, examples—multi-way selection—switch—else-if, examples, Repetition—basic loop structures, Pretest and post-test loops, Counter-controlled and condition-controlled loops, While statement, For statement, Nested loops, do-while statement



Go to

Chapter 3: Decision Making, Branching and Looping

Unit 4: Modular programming: function and parameter declarations, Returning a value, Functions with empty parameter lists, Variable scope, Variable storage class, Local variable storage classes, Global variable storage classes, Pass by reference, Passing addresses to a function, Storing addresses variables, Using addresses, Declaring and using pointers, Passing addresses to a function, Swapping values, Recursion—mathematical recursion—recursion versus iteration.



Go to

Chapter 4: User-Defined Functions

Unit 5: One-dimensional arrays, Input and output of array values, Array initialization, Arrays as function arguments, Two-dimensional arrays, Larger dimensional arrays—matrices, String fundamentals, Library functions, String input and output, String processing



Go to

Chapter 5: Arrays

Chapter 6: Strings

Unit 6: Pointers—concept of a pointer, Initialisation of pointer variables, Pointers as function arguments, Passing by address, Dangling memory, Address arithmetic, Character pointers and Functions, Pointers to pointers, Dynamic memory management functions, Command line arguments

Structures—derived types, Structures declaration, Initialization of structures, Accessing structures, Nested structures, Arrays of structures, structures and functions, Pointers to structures, self-referential structures, Unions, typedef, bit-fields

Declaring, Opening, and Closing file streams, Reading from and Writing to text files, Random file access



Go to

Chapter 7: Pointers

Chapter 8: Structures and Unions

Chapter 9: Data Files

1

Introduction to Computers and Programming

CHAPTER OUTLINE

1.1 Introduction

1.2 Computer Systems

1.3 History of C

1.4 Data Types

1.5 Programming Languages

1.6 Development of C Algorithms

1.7 Software Development Method

1.8 Applying Software

Development Method

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*.

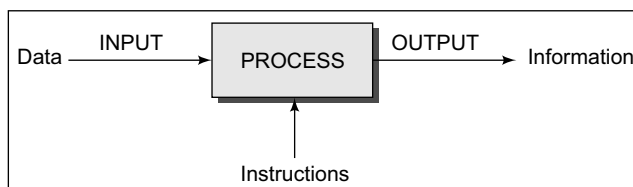


Fig. 1.1 Input–process–output concept

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

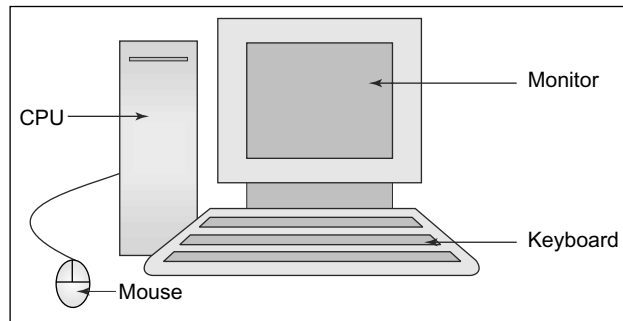


Fig. 1.2 Components of a computer

- **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

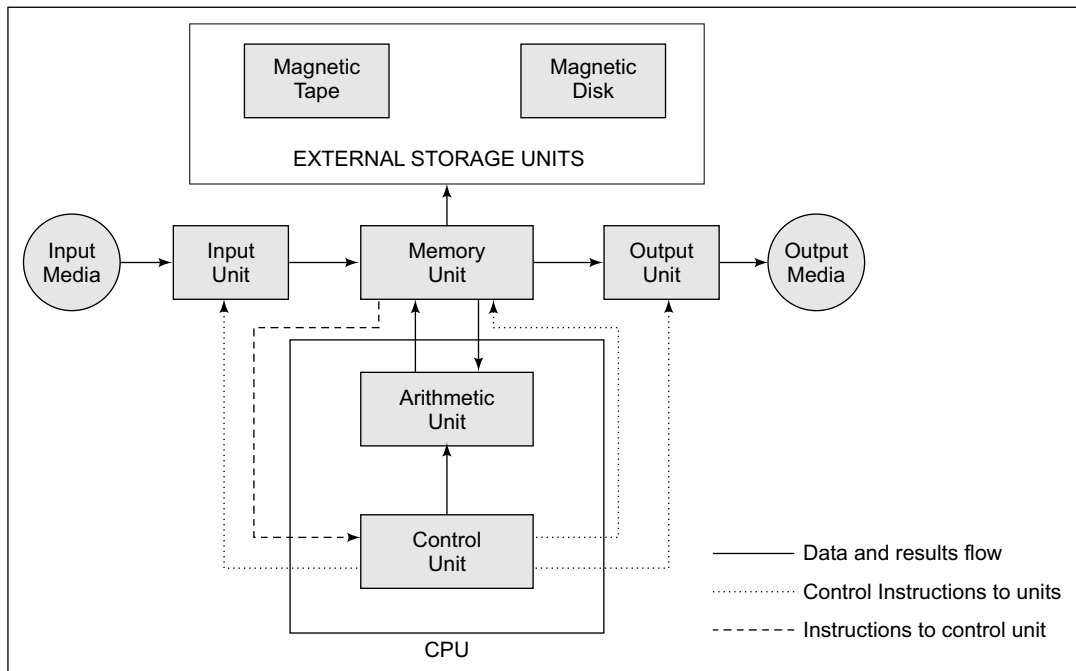


Fig. 1.3 Interaction among hardware components

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 keys such as Shift and Control keys modify the casing style of a character or symbol. The cursor movement

1.4 Computer Programming

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.

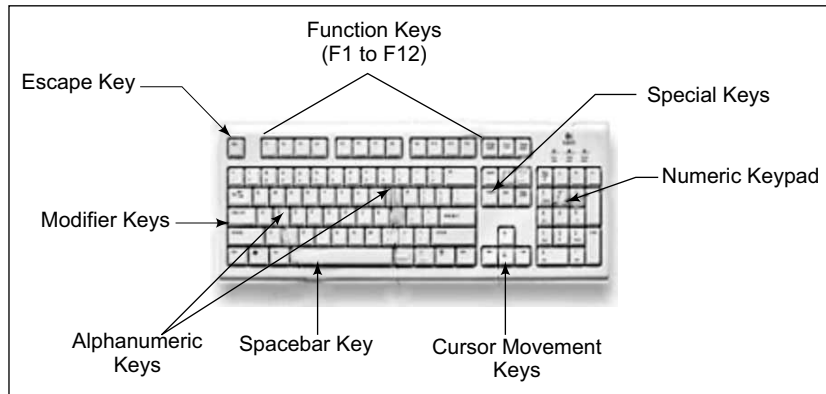


Fig. 1.4 Keyboard

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.

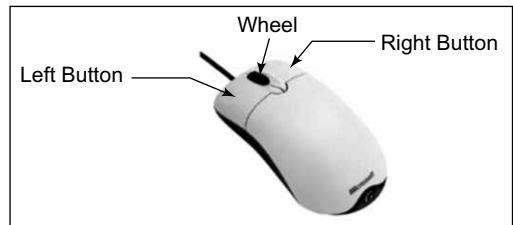


Fig. 1.5 Mouse

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.

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.



Fig. 1.6 Scanner

- **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 (Fig. 1.7). 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:

- **Buses** 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.

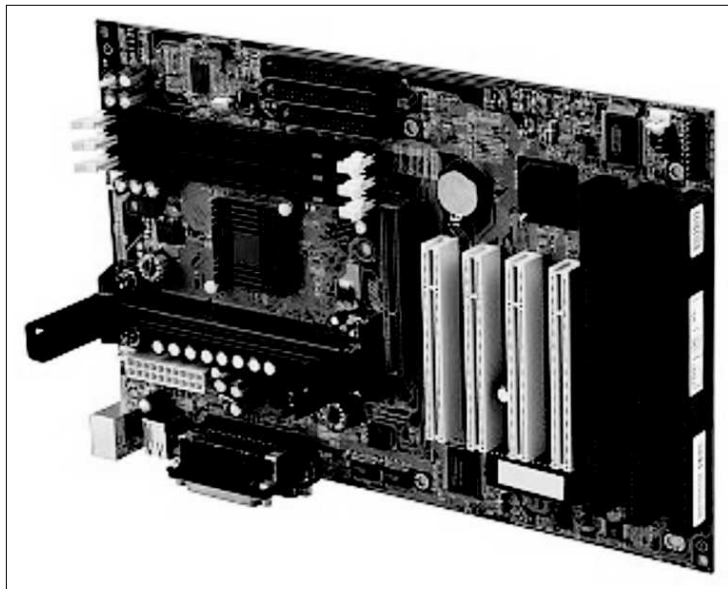


Fig. 1.7 A motherboard

1.6 Computer Programming

- **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.

1.2.3 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 (Fig. 1.8). 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.



Fig. 1.8 Monitor

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 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 (Fig. 1.9). 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.



Fig. 1.9 Printer

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.

Speaker

The speaker is an electromechanical transducer that converts an electrical signal into sound (Fig. 1.10). 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 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 (Fig. 1.11). 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.

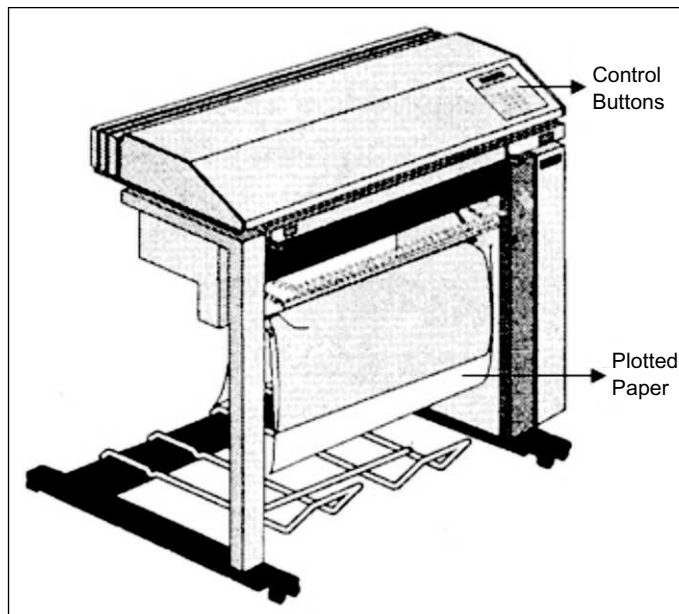


Fig. 1.11 An ink-jet plotter

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.

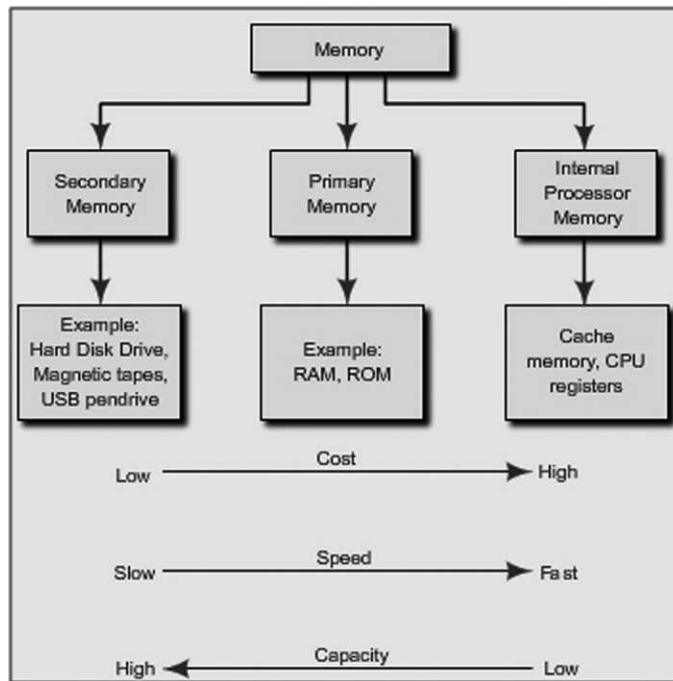


Fig. 1.12 Categorization of memory devices

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 (Fig. 1.13). 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.

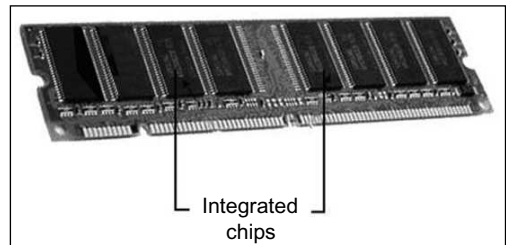


Fig. 1.13 RAM

1.10 Computer Programming

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

TABLE 1.1 Differences between RAM and ROM

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 (Fig. 1.14 and Fig. 1.15).
- **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).

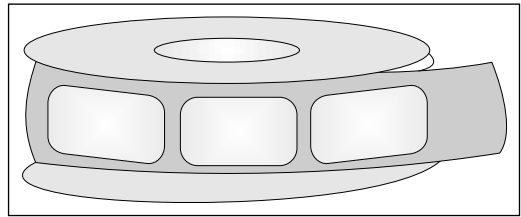


Fig. 1.14 Magnetic tape

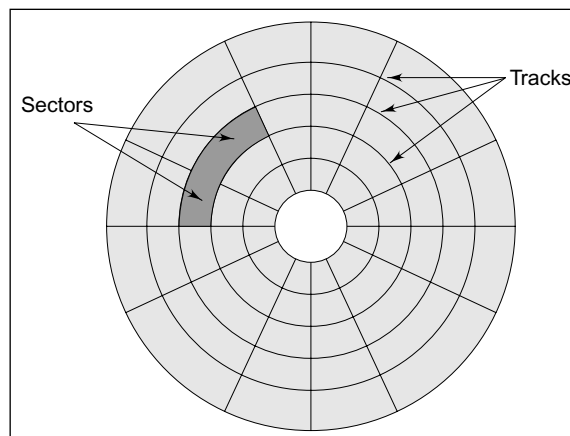


Fig. 1.15 Magnetic disk

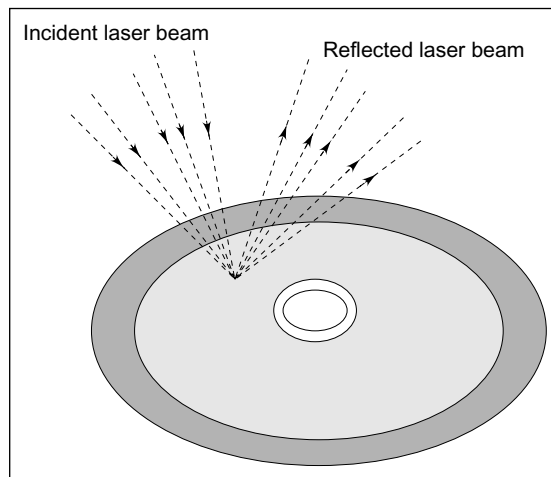


Fig. 1.16 Optical Disk

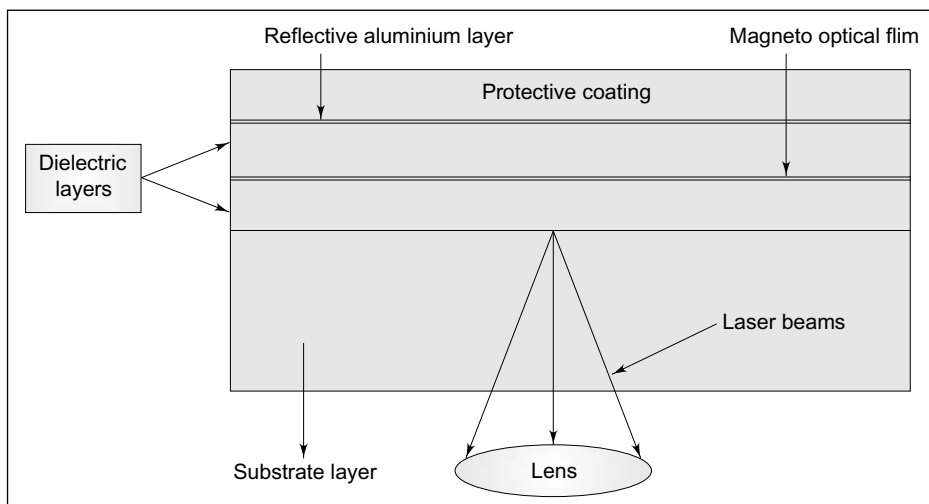


Fig. 1.17 Magneto-optical disk

- Magneto-optical storage device** The magneto-optical devices are generally used to store information, such as large programs, files and backup data (Fig. 1.17). 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 (Fig. 1.18). It is pretty



Fig. 1.18 USB drive

1.12 Computer Programming

fast and compact 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 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 they 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 gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularized 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 began to gain 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 to as C89.

During 1990’s, C++, a language entirely based on C, underwent a number of improvements and changes and became an ANSI/ISO approved language in November 1977. 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. 1.19.

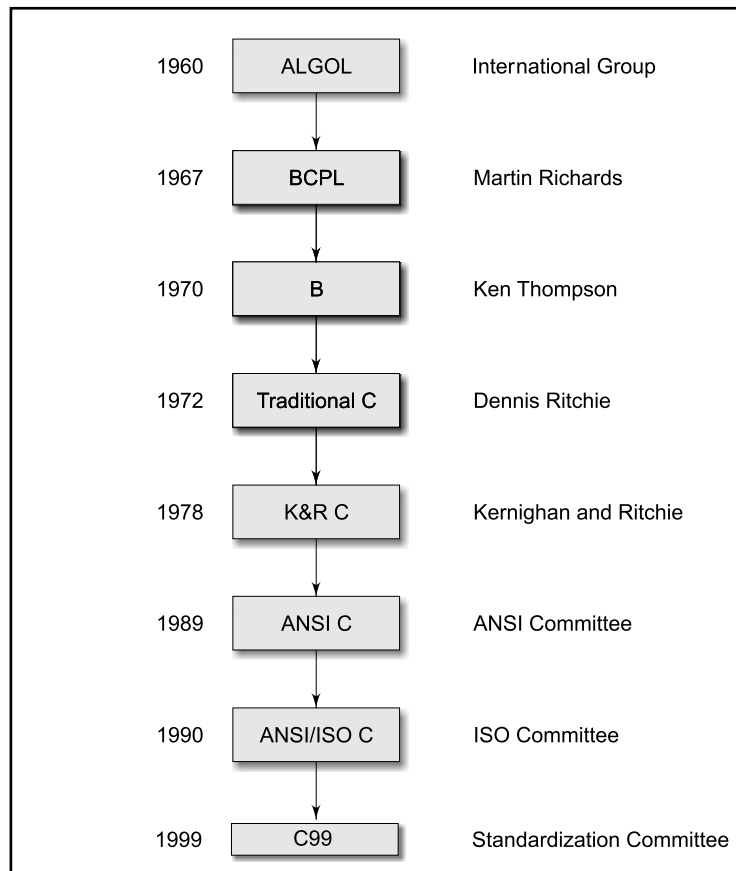


Fig. 1.19 History of C

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

1.4 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

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered.

1.14 Computer Programming

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. 1.20. The range of the basic four types are given in Table 1.2. We discuss briefly each one of them in this section.

NOTE: C99 adds three more data types, namely **_Bool**, **_Complex**, and **_Imaginary**. See Appendix 1.

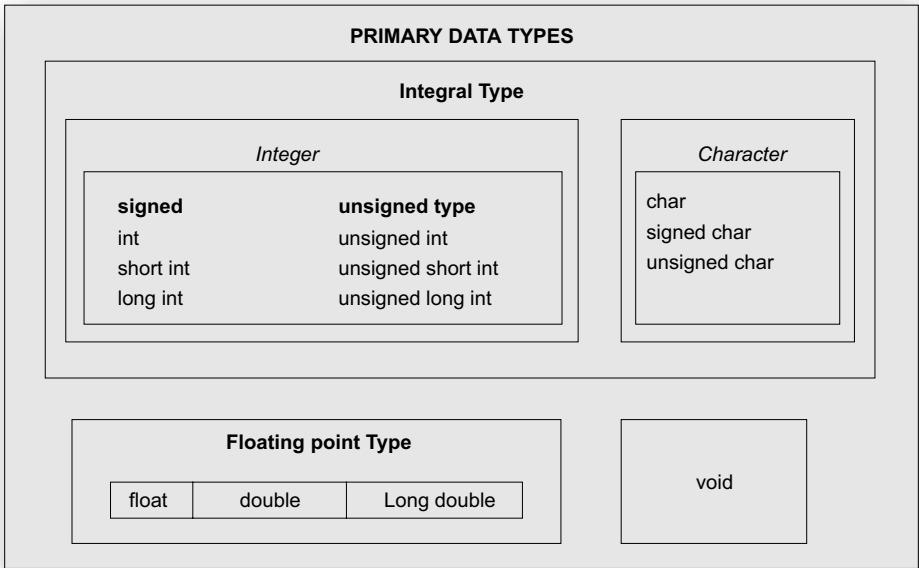


Fig. 1.20 Primary data types in C

TABLE 1.2 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+e38
double	1.7e−308 to 1.7e+308

1.4.1 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, -2^{15} to $+2^{15}-1$). 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 organized from the smallest to the largest, as shown in Fig. 1.21. 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.

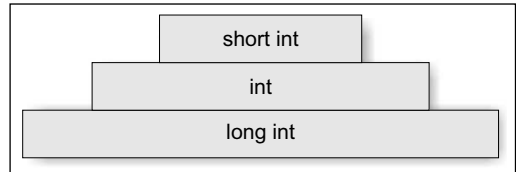


Fig. 1.21 Integer types

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. Table 1.3 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

NOTE: C99 allows **long long** integer types. See Appendix 1.

TABLE 1.3 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

1.4.2 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. 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 Fig. 1.22.

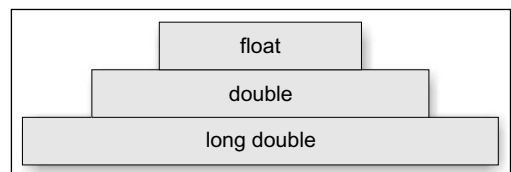


Fig. 1.22 Floating-point types

1.4.3 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.

1.4.4 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.

1.5 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:

- What operation to perform?
- Where to locate data?
- How to present results?
- 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:

- Machine languages (low level languages)
- Assembly (or symbolic) languages
- Procedure-oriented languages (high level languages)

1.5.1 Machine Language (Low Level Languages)

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.

1.5.2 Assembly Language (Symbolic 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

1.5.3 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
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)

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.

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.

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.

1.6 DEVELOPMENT OF C 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.

1.20 Computer Programming

1.6.1 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.

1.6.2 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.

1.6.3 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.

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.

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, $\text{average} = (\text{marks1} + \text{marks2} + \text{marks3})/3$

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.

Algorithm

Step 1 - Start

Step 2 - Accept an year value from the user (year)

Step 3 - If remainder of year value divided by 4 ($\text{year}\%4$) 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.

Algorithm

Step 1 - Start

Step 2 - Accept a number from the user (num)

Step 3 - If remainder of num divided by 2 ($\text{num}/2$) 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.

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
    
```

1.7 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 1.23 shows the various software development steps:

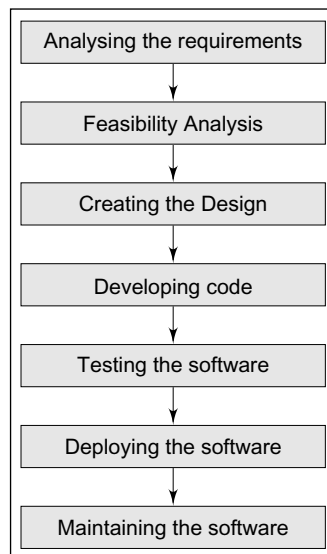


Fig. 1.23 Software development steps

1.7.1 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.

1.7.2 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

1.7.3 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.

1.7.4 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.

1.7.5 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

1.7.6 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.

1.7.7 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.

1.8 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.

Program Objective

To convert the temperature value from Fahrenheit to Celsius

Analysis

Input: Temperature value in Fahrenheit

Output: Temperature value in Celsius

1.26 Computer Programming

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
```




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 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.
- **Algorithm:** It is a complete, detailed, and precise step-by-step method for solving a problem independently.



Just Remember

1. C is a structured, high-level, machine independent language.
2. ANSI C and C99 are the standardized versions of C language.
3. C combines the capabilities of assembly language with the features of a high level language.
4. C is robust, portable and structured programming language.
5. 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.
6. Cache memory has the fastest access time followed by RAM, and secondary storage devices.
7. High-level language is the most user-friendly, followed by assembly and machine language.
8. Machine-level language is the most efficient followed by assembly and high-level language.
9. 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.
10. 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.
11. Describing the process step-by-step is called as algorithm.



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 defined as a computer program for performing a particular task on the computer system?
(a) Hardware (b) Software
(c) Processor (d) Memory
5. 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

6. 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
7. 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
8. 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
9. Which of the following is an input device?
 - (a) Monitor (b) Printer
 - (c) Keyboard (d) Speaker
10. Which of the following is an output device?
 - (a) Keyboard (b) Printer
 - (c) Mouse (d) Scanner
11. Which of the following is designed and developed for performing specific tasks?
 - (a) File management
 - (b) Application software
 - (c) System software
 - (d) Hardware
12. In which year C language was developed?
 - (a) 1951 (b) 1962
 - (c) 1972 (d) 1947
13. 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
14. Which of the following is used as a secondary memory of the computer?
 - (a) Magnetic storage device
 - (b) RAM
 - (c) Cache memory
 - (d) ROM

Answers

- | | | | | |
|---------|---------------------|---------|---------|---------|
| 1. (b) | 2. (a) | 3. (b) | 4. (b) | 5. (d) |
| 6. (a) | 7. (b), (d) and (e) | 8. (a) | 9. (c) | 10. (b) |
| 11. (b) | 12. (c) | 13. (b) | 14. (a) | |



Review Questions

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.
 - (c) Describing the process step by step is called as flowchart.
 - (d) Algorithm involves very complex process.
 - (e) When we break up a big task into smaller steps, what we actually do is to create an algorithm.
 - (f) Each step in an algorithm can be called as an instruction.

1.30 Computer Programming

- (g) In general, the steps in an algorithm can be divided in five basic categories.
- 2. Fill in the blanks with appropriate words in the following statement.
 - (a) The _____ keys include the number keys and the alphabet keys.
- 3. What are input devices? Briefly explain some popular input devices.
- 4. What is the purpose of an output device? Explain various types of output devices.
- 5. What is assembly language? What are its main advantages?
- 6. What is high-level language? What are the different types of high-level languages?
- 7. What is an algorithm?
- 8. Write an algorithm for withdrawing Rs. 1000 from the bank.
- 9. Describe the four basic data types. How could we extend the range of values they represent?
- 10. Give a brief account of history of C language.

2

Basics of C

CHAPTER OUTLINE

2.1 Importance of C	2.7 C Tokens	2.13 ANSI C Library Functions
2.2 Basic Structure of C Programs	2.8 Keywords and Identifiers	2.14 Managing Input and Output Operations
2.3 Programming Style	2.9 Operators and Expressions	2.15 Case Studies
2.4 Executing A 'C' Program	2.10 Constants	
2.5 Sample Programs	2.11 Variables	
2.6 C Character Set	2.12 Declaration of Variables	

2.1 IMPORTANCE 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 of the 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 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.

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

2.2 BASIC STRUCTURE OF C PROGRAMS

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. 2.1.

2.2 Computer Programming

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.

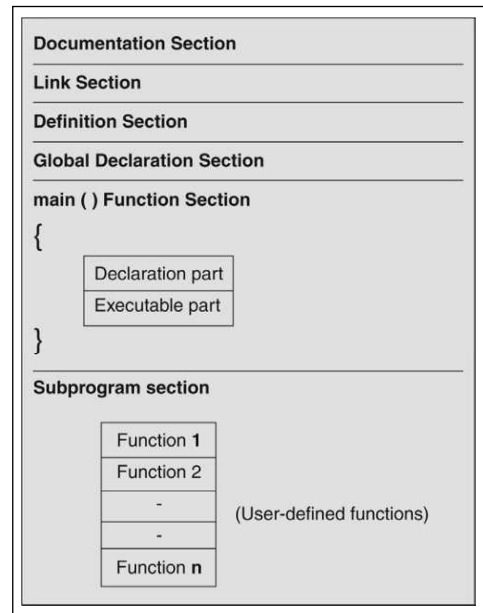


Fig. 2.1 Typical structure of a C program

2.3 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

```
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.

2.4 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 2.2 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.

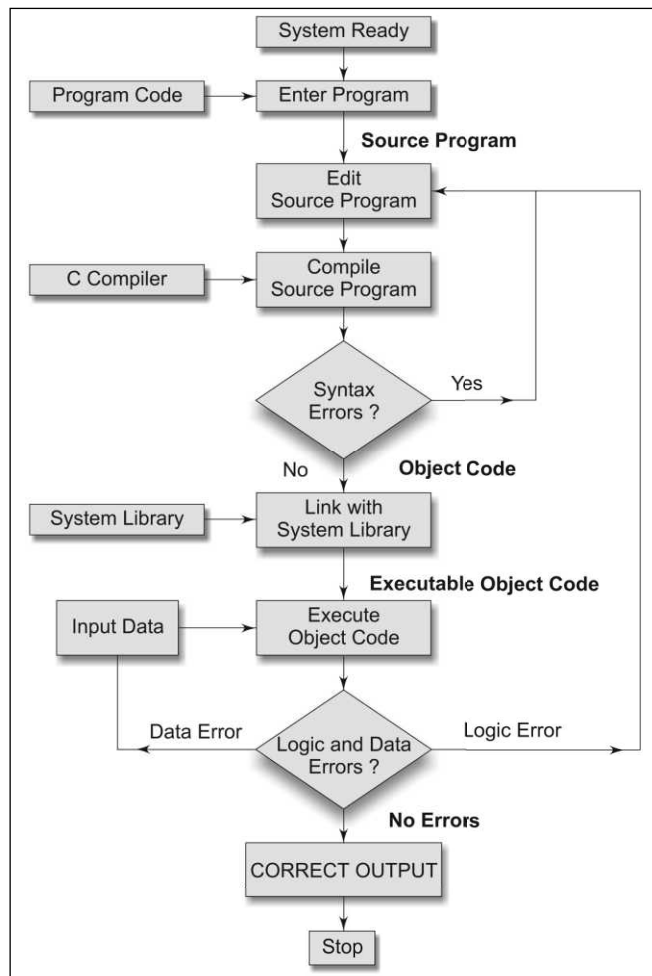


Fig. 2.2 Process of compiling and running a C program

2.5 SAMPLE PROGRAMS

2.5.1 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 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:

```
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:

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

2.6 Computer Programming

```
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.

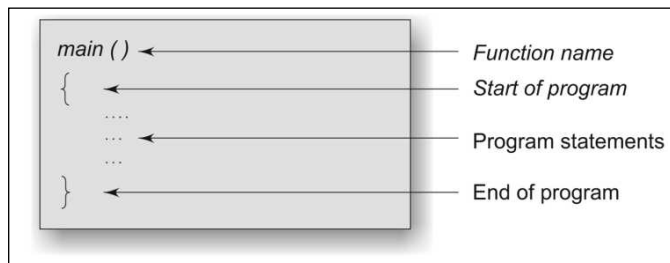


Fig. 2.4 Format of simple C programs

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.

2.5.2 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.

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);

```


2.8 Computer Programming

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.

2.5.3 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.

`#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.

2.10 Computer Programming

2.5.4 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.

2.5.5 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)
    {
        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

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.

2.6 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 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 confirm 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

(Contd.)

Characters	Notations
	<ul style="list-style-type: none"> • ? 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 •) right parenthesis • ~ tilde • [left bracket • _ under score •] right bracket • \$ dollar sign • { left brace • % percent sign • } right brace
White Spaces	<ul style="list-style-type: none"> • 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.

2.6.1 Trigraph Characters

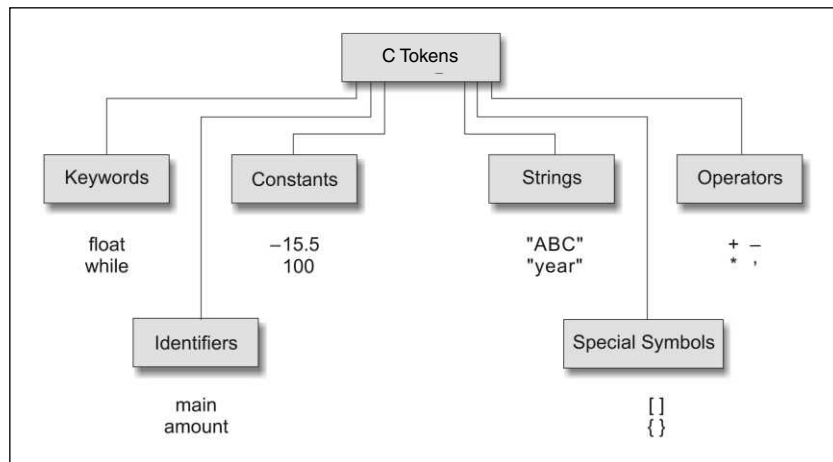
Many non-English keyboards do not support all the characters mentioned in Table 2.1. ANSI C introduces the concept of ‘trigraph’ sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed 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

2.7 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. 2.10. C programs are written using these tokens and the syntax of the language.

**Fig. 2.10** C tokens and examples

2.8 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.

2.9 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

2.9.1 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.

2.16 Computer Programming

The unary minus operator, in effect, multiplies its single operand by -1 . Therefore, a number preceded by a minus sign changes its sign.

TABLE 2.4 Arithmetic operators

Operator	Meaning
+	Addition or unary plus
−	Subtraction or unary minus
*	Multiplication
/	Division
%	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
```

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 2.1 The program in Fig. 2.11 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. 2.11 Illustration of integer arithmetic.

Example 2.2 The program in Fig. 2.12 demonstrates the working of arithmetic operators.

```

Program
#include <stdio.h>
int main()
{
    int a = 9, b = 4, c;
    c = a+b;      /* '+' operator is used to add two numbers */
    clrscr();
    printf("a+b = %d \n", c);
    c = a-b;      /* '-' operator is used to subtract two numbers */
    printf("a-b = %d \n", c);
    c = a*b;      /* '*' operator is used to multiply two numbers */
    printf("a*b = %d \n", c);
    c=a/b;        /* '/' operator is used to divide two numbers */
    printf("a/b = %d \n", c);
    c=a%b;        /* '%' operator is used to find the remainder of two numbers */
    printf("Remainder when a divided by b = %d \n", c);
}

```


2.18 Computer Programming

```
    getch();  
    return 0;  
}  
Output  
a+b = 13  
  
a-b = 5  
  
a*b = 36  
  
a/b = 2  
  
Remainder when a divided by b = 1
```

Fig. 2.12 Working of arithmetic operators

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:

$$x = 6.0/7.0 = 0.857143$$

$$y = 1.0/3.0 = 0.333333$$

$$z = -2.0/3.0 = -0.666667$$

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.

2.9.2 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

$10 < 20$ is true

but

$20 < 10$ is false

C supports six relational operators in all. These operators and their meanings are illustrated in Table 2.5.

TABLE 2.5 Relational operators

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

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:

$4.5 \leq 10$ TRUE

$5 < -10$ FALSE

$-35 \geq 0$ 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.

2.9.3 Logical Operators

In addition to the relational operators, C supports the following three logical operators.

&& meaning logical AND

|| meaning logical OR

! meaning logical NOT

2.20 Computer Programming

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 2.6. 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 2.6 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.

2.9.4 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 2.7:

TABLE 2.7 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;
```


2.22 Computer Programming

It is easier to read and understand and is more efficient because the expression $5*j-2$ is evaluated only once.

Example 2.3 The program in Fig. 2.13 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. 2.13 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.

2.9.5 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 $++n$.

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 -.

2.9.6 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.

2.24 Computer Programming

```
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.

2.9.7 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 2.8 provides a list the bitwise operators and their meanings.

TABLE 2.8 Bitwise operators

Operator	Meaning
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

2.9.8 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:

```
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;
```


Sizeof 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 2.4 Figure 2.14 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. 2.14 Further illustration of arithmetic operators.

Example 2.5 The program in Fig. 2.15 demonstrates the working of logical operators.

Program

```
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;
    clrscr();
    result = (a == b) && (c > b);
    printf("(a = b) && (c > b) equals to %d\n", result);
    result = (a == b) && (c < b);
    printf("(a = b) && (c < b) equals to %d\n", result);
    result = (a == b) || (c < b);
    printf("(a = b) || (c < b) equals to %d\n", result);
    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) equals to %d\n", result);
    result = !(a != b);
    printf("(a != b) equals to %d\n", result);
    result = !(a == b);
    printf("(a == b) equals to %d\n", result);
    getch();
    return 0;
}
```

Output

```
(a = b) && (c > b) equals to 1
(a = b) && (c < b) equals to 0
(a = b) || (c < b) equals to 1
(a != b) || (c < b) equals to 0
!(a != b) equals to 1
!(a == b) equals to 0
```

Fig. 2.15 Working of logical operators

2.9.9 Operator Precedence

Precedence of operators refers to the order in which they are operated in a program. Table 2.9 provides a list of the precedence of operators.

TABLE 2.9 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

2.28 Computer Programming

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 2.6 The program in Fig. 2.16 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. 2.16 Illustrations of evaluation of expressions

2.9.10 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. 2.17. The numbers inside parentheses refer to step numbers.

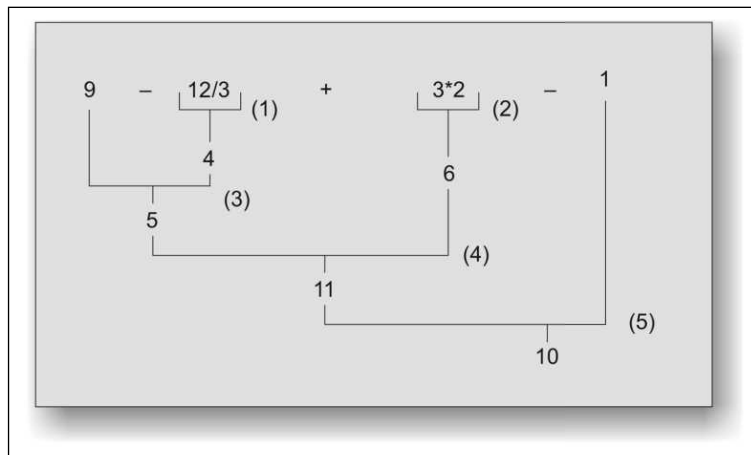


Fig. 2.17 Illustration of hierarchy of operations

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

$$9 - 12/(3+3)*(2-1)$$

2.30 Computer Programming

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).

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.

2.9.11 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 $(1.0/3.0) * 3.0$ 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 2.7 The program in Fig. 2.18 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. 2.18 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.

2.9.12 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. 2.19.

2.32 Computer Programming

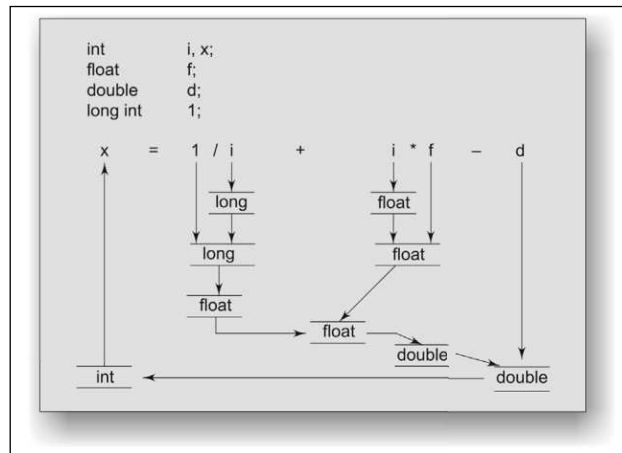


Fig. 2.19 Process of implicit type conversion

Figure 2.20 shows the conversion hierarchy for implicit-type conversion in an expression.

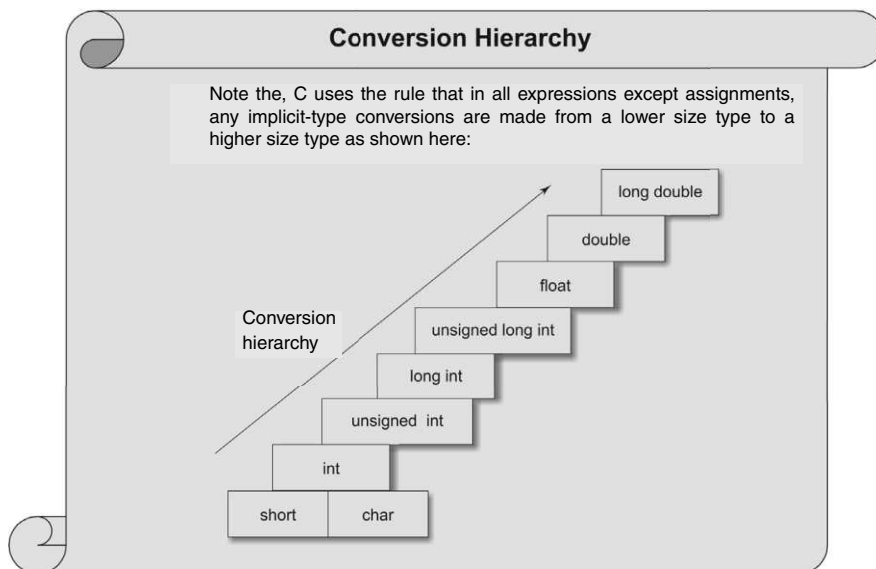


Fig. 2.20 Conversion hierarchy

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 2.10.

TABLE 2.10 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.

2.34 Computer Programming

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.

2.9.13 Operator Precedence and Associativity

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 2.11 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 2.11 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		
sizeof	Size of an object	Left to right	3
(type)	Type cast (conversion)		
*	Multiplication		
/	Division		
%	Modulus		
+	Addition		4
-	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		

(Contd.)

Operator	Description	Associativity	Rank
&	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."

2.36 Computer Programming

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”

Negation

Negation operator is also called bitwise “not” operator. This operator works on bits and converts the bit “1” to “0” and vice versa. For example, number 5 is represented in binary as 00000000 00000000 00000000 00000101. Applying the negation operator will convert a binary string to 11111111 11111111 11111111 1111010, which becomes equivalent to -6.

Negation operator is denoted by symbol ~. Thus, the following code fragment will give -6 as output:

```
int temp = 5;
int a;
a = ~ temp;
printf("%d", temp);
```

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. 2.21.

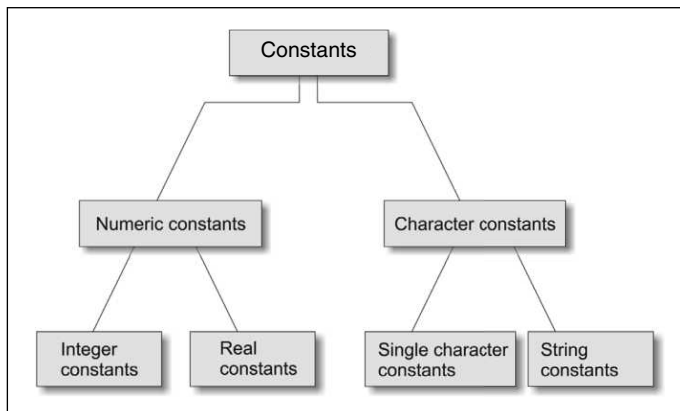


Fig. 2.21 Basic types of C constants

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 2.8 Representation of integer constants on a 16-bit computer.

The program in Fig. 2.22 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.

<pre> 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 </pre>

Fig. 2.22 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 2.12.

TABLE 2.12 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 2.13. Note that each one of them represents one character, although they consist two characters. These character combinations are known as escape sequences.

TABLE 2.13 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,

2.40 Computer Programming

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
```

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 2.14.

TABLE 2.14 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 DECLARATION OF VARIABLES

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

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.

2.12.1 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 2.15 shows various data types and their keyword equivalents.

TABLE 2.15 Data types and their keywords

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

The program given in Fig. 2.23 illustrates declaration of variables.

2.42 Computer Programming

`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. 2.23 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`.

2.12.2 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];
```


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;
```

2.12.3 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();
}
```


2.44 Computer Programming

```
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 2.16 provide a list of the various storage class specifiers along with their meanings.

TABLE 2.16 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.

2.12.4 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.

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.

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 2.9 The program in Fig. 2.24 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. 2.24 Examples of assignments

In the program given in Fig. 2.24, 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` 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.

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 2.10 The program in Fig. 2.25 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
```


2.48 Computer Programming

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. 2.25 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.

Example 2.11 Figure 2.26 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 ;
    }
}
```



```

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. 2.26 *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.

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.

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;
```

2.13 ANSI C LIBRARY FUNCTIONS

C language is accompanied by a number of library functions that perform various tasks. The ANSI committee has standardized header files which contain these functions. What follows is a list of commonly used functions and the header files where they are defined. For a more complete list, the reader should refer to the manual of the version of C that is being used.

The header files that are included in this topic are as follows (Table 2.17):

- <ctype.h>** Character testing and conversion functions
- <math.h>** Mathematical functions
- <stdio.h>** Standard I/O library functions
- <stdlib.h>** Utility functions such as string conversion routines, memory allocation routines, random number generator, etc.
- <string.h>** String manipulation functions
- <time.h>** Time manipulation functions

Note: The following function parameters are used:

- c - character type argument
- d - double precision argument
- f - file argument
- i - integer argument
- l - long integer argument
- p - pointer argument
- s - string argument
- u - unsigned integer argument

An asterisk (*) denotes a pointer

TABLE 2.17 Header Files used in C

Function	Data type returned	Task
<ctype.h>		
isalnum(c)	int	Determine if argument is alphanumeric. Return nonzero value if true; 0 otherwise.
isalpha(c)	int	Determine if argument is alphabetic. Return nonzero value if true; 0 otherwise.
isascii(c)	int	Determine if argument is an ASCII character. Return nonzero value if true; 0 otherwise.
iscntrl(c)	int	Determine if argument is an ASCII control character. Return nonzero value if true; 0 otherwise.
isdigit(c)	int	Determine if argument is a decimal digit. Return nonzero value if true; 0 otherwise.
isgraph(c)	int	Determine if argument is a graphic printing ASCII character. Return nonzero value if true; 0 otherwise.
islower(c)	int	Determine if argument is lowercase. Return nonzero value if true; 0 otherwise.
isodigit(c)	int	Determine if argument is an octal digit. Return nonzero value if true; 0 otherwise.
isprint(c)	int	Determine if argument is a printing ASCII character. Return nonzero value if true; 0 otherwise.
ispunct(c)	int	Determine if argument is a punctuation character. Return nonzero value if true; 0 otherwise.
isspace(c)	int	Determine if argument is a whitespace character. Return nonzero value if true; 0 otherwise.
isupper(c)	int	Determine if argument is uppercase. Return nonzero value if true; 0 otherwise.
isxdigit(c)	int	determine if argument is a hexadecimal digit. Return nonzero value if true; 0 otherwise.
toascii(c)	int	Convert value of argument to ASCII.
tolower(c)	int	Convert letter to lowercase.
toupper(c)	int	Convert letter to uppercase.
<math.h>		
acos(d)	double	Return the arc cosine of d.
asin(d)	double	Return the arc sine of d.
atan(d)	double	Return the arc tangent of d.
atan2(d1,d2)	double	Return the arc tangent of d1/d2.
ceil(d)	double	Return a value rounded up to the next higher integer.
cos(d)	double	Return the cosine of d.
cosh(d)	double	Return the hyperbolic cosine of d.
exp(d)	double	Raise e to the power d.
fabs(d)	double	Return the absolute value of d.
floor(d)	double	Return a value rounded down to the next lower integer.
fmod(d1, d2)	double	Return the remainder of d1/d2 (with same sign as d1).
labs(l)	long int	Return the absolute value of l.
log(d)	double	Return the natural logarithm of d.

(Contd.)

2.52 Computer Programming

Function	Data type returned	Task
log10(d)	double	Return the logarithm (base 10) of d.
pow(d1,d2)	double	Return d1 raised to the d2 power.
sin(d)	double	Return the sine of d.
sinh(d)	double	Return the hyperbolic sine of d.
sqrt(d)	double	Return the square root of d.
tan(d)	double	Return the tangent of d.
tanh(d)	double	Return the hyperbolic tangent of d.
<stdio.h>		
fclose(f)	int	Close file f. Return 0 if file is successfully closed.
feof(f)	int	Determine if an end-of-file condition has been reached. If so, return a nonzero value; otherwise, return 0.
fgetc(f)	int	Enter a single character from file f.
fgets(s, i, f)	char*	Enter string s, containing i characters, from file f.
fprint(f,...)	int	Send data items to file f.
fputc(c,f)	int	Send a single character to file f.
fputs(s,f)	int	Send string s to file f.
fread(s,i1,i2,f)	int	Enter i2 data items, each of size i1 bytes, from file f to string s.
fscanf(f,...)	int	Enter data items from file f
fseek(f,1,i)	int	Move the pointer for file f a distance 1 bytes from location i.
ftell(f)	long int	Return the current pointer position within file f.
fwrite(s,i1,i2,f)	int	Send i2 data items, each of size i1 bytes from string s to file f.
getc(f)	int	Enter a single character from file f.
getchar(void)	int	Enter a single character from the standard input device.
gets(s)	char*	Enter string s from the standard input device.
printf(...)	int	Send data items to the standard output device.
putc(c,f)	int	Send a single character to file f.
putchar(c)	int	Send a single character to the standard output device.
puts(s)	int	Send string s to the standard output device.
rewind(f)	void	Move the pointer to the beginning of file f.
scanf(...)	int	Enter data items from the standard input device.
<stdlib.h>		
abs(i)	int	Return the absolute value of i.
atof(s)	double	Convert string s to a double-precision quantity.
atoi(s)	int	Convert string s to an integer.
atol(s)	long	Convert string s to a long integer.
calloc(u1,u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes. Return a pointer to the beginning of the allocated space.

(Contd.)

Function	Data type returned	Task
exit(u)	void	Close all files and buffers, and terminate the program. (Value of u is assigned by the function, to indicate termination status).
free(p)	void	Free a block of allocated memory whose beginning is indicated by p.
malloc(u)	void*	Allocate u bytes of memory. Return a pointer to the beginning of the allocated space.
rand(void)	int	Return a random positive integer.
realloc(p, u)	void*	Allocate u bytes of new memory to the pointer variable p. Return a pointer to the beginning of the new memory space.
srand(u)	void	Initialize the random number generator.
system(s)	int	Pass command string s to the operating system. Return 0 if the command is successfully executed; otherwise, return a nonzero value typically -1.
<string.h>		
strcmp(s1, s2)	int	Compare two strings lexicographically. Return a negative value if $s1 < s2$; 0 if s1 and s2 are identical; and a positive value if $s1 > s2$.
strncmpi(s1, s2)	int	Compare two strings lexicographically, without regard to case. Return a negative value if $s1 < s2$; 0 if s1 and s2 are identical; and a value of $s1 > s2$.
strcpy(s1, s2)	char*	Copy string s2 to string s1.
strlen(s)	int	Return the number of characters in string s.
strset(s, c)	char*	Set all characters within s to c (excluding the terminating null character \0).
<time.h>		
difftime(t1, t2)	double	Return the time difference $t1 - t2$, where t1 and t2 represent elapsed time beyond a designated base time (see the time function).
time(p)	long int	Return the number of seconds elapsed beyond a designated base time.

NOTE: C99 adds many more header files and adds many new functions to the existing header files. For more details, refer Appendix 1.

2.14 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.

2.54 Computer Programming

2.14.1 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 2.12 The program in Fig. 2.27 shows the use of `getchar` function in an interactive environment.

```
Program  
#include <stdio.h>  
main()  
{  
    char answer;  
    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. 2.27 Use of `getchar` function to read a character from keyboard

The above program in Fig. 2.27 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 2.13 The program shown in Fig. 2.28 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. 2.28 Program to test the character type

The above program receives in Fig. 2.28 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 2.18. 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>
```

2.14.2 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 2.18 Character test functions

Function	Test
<code>isalnum(c)</code>	Is c an alphanumeric character?
<code>isalpha(c)</code>	Is c an alphabetic character?
<code>isdigit(c)</code>	Is c a digit?
<code>islower(c)</code>	Is c lower case letter?
<code>isprint(c)</code>	Is c a printable character?
<code>ispunct(c)</code>	Is c a punctuation mark?
<code>isspace(c)</code>	Is c a white space character?
<code>isupper(c)</code>	Is c 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 2.14 A program that reads a character from the keyboard and then prints it in reverse case is given in Fig. 2.29. 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. 2.29 Reading and writing of alphabets in reverse case

The above program in Fig. 2.29 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.

2.14.3 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:

```
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 2.15 Various input formatting options for reading integers are experimented in the program shown in Fig. 2.30.

```
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. 2.30 Reading integers using *scanf*

In the above program in Fig. 2.30, 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.

Example 2.16 The program in Fig. 2.31 reads and displays an integer value.

```

Program
#include <stdio.h>
int main()
{
    int testInteger;
    clrscr();
    printf("Enter an integer: ");
    scanf("%d",&testInteger);
    printf("Number = %d",testInteger);
    // %d format string is used in case of integers

    getch();
    return 0;
}
Output
Enter an integer: 4
Number = 4

```

Fig. 2.31 Program to read and display integer value

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 2.17 Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 2.32.

```

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. 2.32 Reading of real numbers

Example 2.18 The program in Fig. 2.33 reads and displays a float value.

```

Program
#include <stdio.h>
int main()
{
    float f;
    clrscr();
    printf("Enter a number: ");
    // %f format string is used in case of floats
    scanf("%f",&f);
    printf("Value = %f", f);
    getch();
    return 0;
}
Output
Enter a number: 23.45
Value = 23.450000

```

Fig. 2.33 Program to read and display float value

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 2.19 Reading of strings using `%wc` and `%ws` is illustrated in Fig. 2.34.

```

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. 2.34 *Reading of strings*

The program in Fig. 2.34 illustrates the use of various field specifications for reading strings. When we use `%wc` for reading a string, the system will wait until the `w`th 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:

```
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:

2.64 Computer Programming

```
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 2.20 The program shown in Fig. 2.35 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. 2.35 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 2.19 shows the commonly used `scanf` format codes:

TABLE 2.19 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

2.14.4 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

2.66 Computer Programming

- The maximum number of characters have been read or
- An error is detected, or
- The end of file is reached

2.14.5 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 2.20 illustrates the output of the number 9876 under different formats.

TABLE 2.20 Scanf format codes for the output of number 9876.

Format	Output
<code>printf("%d", 9876)</code>	9 8 7 6
<code>printf("%6d", 9876)</code>	9 8 7 6
<code>printf("%2d", 9876)</code>	9 8 7 6
<code>printf("%-6d", 9876)</code>	9 8 7 6
<code>printf("%06d", 9876)</code>	0 0 9 8 7 6

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 2.21 The program in Fig. 2.36 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. 2.36 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 2.21 illustrates the output of the number $y = 98.7654$ under different format specifications:

TABLE 2.21 Scanf format codes for the output of number 98.7654

Format	Output
printf("%.4f",y)	98.7654
printf("%.2f",y)	98.76
printf("%-7.2f",y)	98.76
printf("%f",y)	98.7654
printf("%.2e",y)	9.87654e+01
printf("%.4e",-y)	-9.87654e+01
printf("%-10.2e",y)	9.87654e+01
printf("%e",y)	9.87654e+01

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 2.22 All the options of printing a real number are illustrated in Fig. 2.37.

```
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. 2.37 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 2.38 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. 2.38 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 2.22 shows some commonly used `printf` format codes:

TABLE 2.22 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 0x

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.

2.15 CASE STUDIES

1. Salesman's Salary

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.

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. 2.39.

```

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;

```


2.72 Computer Programming

```
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. 2.39 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. 2.40. 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. 2.40 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.

3. Calculation of Average of Numbers

A program to calculate the average of a set of N numbers is given in Fig. 2.41.

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. 2.41 Average of N numbers

2.74 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.

4. Temperature Conversion Problem

The program presented in Fig. 2.42 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. 2.42 Temperature conversion–Fahrenheit–Celsius

5. Inventory Report

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

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. 2.43 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");

```


2.76 Computer Programming

```
/*.....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. 2.43 Program for inventory report

6. Reliability Graph

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.

The program given in Fig. 2.44 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.



Key Terms

- **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.
- **Size of 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.



Just Remember

1. Do not use the underscore as the first character of identifiers (or variable names) because many of the identifiers in the system library start with underscore.
2. Use only 31 or less characters for identifiers. This helps ensure portability of programs.
3. Do not use keywords or any system library names for identifiers.
4. Each variable used must be declared for its type at the beginning of the program or function.
5. All variables must be initialized before they are used in the program.
6. Do not combine declarations with executable statements.
7. Do not use semicolon at the end of **#define** directive.
8. The character **#** should be in the first column.
9. Do not give any space between **#** and **define**.
10. A variable defined before the main function is available to all the functions in the program.
11. A variable defined inside a function is local to that function and not available to other functions.
12. Use *decrement* and *increment* operators carefully. Understand the difference between **post-fix** and **prefix** operations before using them.
13. Do not use *increment* or *decrement* operators with any expression other than a *variable identifier*.
14. It is illegal to apply modulus operator **%** with anything other than integers.
15. The result of an expression is converted to the type of the variable on the left of the assignment before assigning the value to it. Be careful about the loss of information during the conversion.
16. It is an error if any space appears between the two symbols of the operators **==**, **!=**, **<=** and **>=**.
17. It is an error if the two symbols of the operators **!=**, **<=** and **>=** are reversed.
18. Use spaces on either side of binary operator to improve the readability of the code.
19. Do not use increment and decrement operators to floating point variables.
20. Do not confuse the equality operator **==** with the assignment operator **=**.
21. Use **sizeof** operator to determine the length of arrays and structures where their sizes are not already known.
22. Be aware of side effects produced by some expressions.
23. Avoid any attempt to divide by zero. It is normally undefined. It will either result in a fatal error or in incorrect results.
24. Do not forget a semicolon at the end of an expression.
25. Do not use a variable in an expression before it has been assigned a value.

26. Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary.
27. Add parentheses wherever you feel they would help to make the evaluation order clear.
28. Understand clearly the precedence of operators in an expression. Use parentheses, if necessary.
29. Associativity is applied when more than one operator of the same precedence are used in an expression. Understand which operators associate from right to left and which associate from left to right.
30. Provide proper field specifications for every variable to be read or printed.
31. Enclose format control strings in double quotes.
32. Do not forget to use address operator & for basic type variables in the input list of **scanf**.
33. Do not specify any precision in input field specifications.
34. Do not provide any white-space at the end of format string of a **scanf** statement.
35. Do not use commas in the format string of a **scanf** statement.
36. Use double quotes for character string constants.
37. Use single quotes for single character constants.
38. Provide sufficient field width to handle a value to be printed.
39. Be aware of the situations where output may be imprecise due to formatting.
40. Do not forget to close the format string in the **scanf** or **printf** statement with double quotes.
41. Using an incorrect conversion code for data type being read or written will result in runtime error.
42. Do not forget the comma after the format string in **scanf** and **printf** statements.
43. Not separating read and write arguments is an error.
44. Using an address operator & with a variable in the **printf** statement will result in runtime error.



Multiple Choice Questions

1. Who amongst the following developed the C programming language?
 - (a) James Gosling (b) Bjarne Stroustrup
 - (c) Dennis Ritchie (d) Ray Boyce
2. Which special function is used by the C system to tell the computer where the program starts?
 - (a) printf (b) begin
 - (c) main (d) scanf
3. What is a name having a few letters, numbers and special character _ (underscore) called?
 - (a) Keywords (b) Tokens
 - (c) Reserved keywords (d) Identifiers
4. The statements between a function body are indicated by:
 - (a) /**/ (b) { }
 - (c) “ ” (d) \n
5. What should be used to end every program statement in C language?
 - (a) Semicolon (b) Comma
 - (c) Full stop (d) Colon
6. Which of the following depicts the correct sequence of steps to run a program?
 - (a) Compile, create, link, execute
 - (b) Create, compile, link, execute
 - (c) Link, compile, create, execute
 - (d) Link, create, compile, execute
7. What are the words with predefined meaning but cannot be used as variables known as?
 - (a) Constant
 - (b) Identifier
 - (c) Keywords
 - (d) Datatype

8. Which of the following is not a data type?
(a) integer (b) char
(c) void (d) default
9. How many keywords are recognized by standard ANSI C?
(a) 30 (b) 32
(c) 24 (d) 36
10. Which amongst the following is not a reserved keyword for C?
(a) case (b) auto
(c) default (d) main
11. What is the size of a character variable in C language?
(a) 1 byte (b) 2 bytes
(c) 4 bytes (d) 8 bytes
12. Which of the following is not a valid name for a C variable?
(a) Learning C (b) Learning_C
(c) Learning C (d) LEARNING C
13. Which of the following statements depict the difference between declaration and definition of a variable correctly?
(a) A definition occurs once, but a declaration can occur numerous times
(b) A declaration occurs once, but a definition may occur numerous times
(c) Both can occur multiple times, but a declaration must occur first
(d) Both can occur multiple times, but a definition must occur first
14. Which of the following statements hold true for an identifier?
(a) It can use a keyword
(b) It can contain white space
(c) It can contain letters, digits and all other types of symbols
(d) First character of an identifier must be an alphabet or underscore
15. What will the following piece of code return?

```
main()
{
    printf("%d", 'a');
}
```
- (a) 0 (b) a
(c) Value assigned to a, if any
(d) ASCII value of 'a' i.e. 97
16. What is a sequence of characters enclosed in double quotes known as?
(a) Constant (b) Variable
(c) String constant (d) Character constant
17. What is the maximum value of an unsigned integer?
(a) 65535 (b) -65535
(c) 32767 (d) -32767
18. Which of the following is a valid define statement?
(a) #define MAX 200
(b) #define MAX=200
(c) # define MAX 200
(d) #define MAX 200;
19. What is the keyword typedef used for?
(a) Define the type of function
(b) Create a type of function
(c) Create a new datatype name
(d) Define the type of variable
20. Which of the following is not a basic data type in C?
(a) char (b) float
(c) array (d) double
21. Which of the following is a valid string constant?
(a) "programming" (b) "Programming
(c) //programming (d) \$programming\$
22. What is the range of signed char on 16 bit machines?
(a) -32768 to -32768 to 2767
(b) 3.4e-38 to 3.4e+e38
(c) -128 to 127
(d) -256 to 255
23. What is a real number treated as by default?
(a) integer (b) float
(c) double (d) long double
24. How many bits does a double data type use?
(a) 8 bits (b) 24 bits
(c) 32 bits (d) 64 bits

2.82 Computer Programming

25. What are those operators called that are used to compare values of operands to produce logical value in C language?
- (a) Bitwise operator
 - (b) Assignment operator
 - (c) Relational operator
 - (d) Arithmetic operator
26. Which operator is used to assign a value to a variable in order to perform arithmetic operations?
- (a) Logical operator
 - (b) Assignment operator
 - (c) Relational operator
 - (d) Arithmetic operator
27. Which of the following operators are used to perform operations on data in binary level?
- (a) Logical operator
 - (b) Assignment operator
 - (c) Bitwise operator
 - (d) Arithmetic operator
28. Which of the following statements hold true for the assignment statement: $a = b$?
- (a) Both variables, a and b are the same
 - (b) The value of b is assigned to variable a , and if variable b changes later the value of variable a will remain unchanged.
 - (c) The value of b is assigned to variable a , and if variable b changes later the value of variable a will also change.
 - (d) The value of variable a is assigned to variable b , and the value of variable b is assigned to variable a .
29. If x is an integer variable, which value will $x = 5/2$ yield?
- (a) 2.5
 - (b) 2.00000
 - (c) 2
 - (d) 0
30. Which of the following contains only hexadecimal integers?
- (a) 0x9F, 0x1, 0xbcd
 - (b) 01100, 037, 00x
 - (c) 00110, 110001, 1001
 - (d) H9F, HFF, HAA
31. What does the unary operator "&" yield when applied to a variable?
- (a) The variable's correct value
 - (b) The variable's binary form
 - (c) The variable's address
 - (d) The variable as it is
32. If an expression contains relational, assignment and arithmetic operators without any parenthesis being specified, what will be the order of evaluation of the operators?
- (a) Relational, assignment, arithmetic
 - (b) Assignment, arithmetic, relational
 - (c) Assignment, relational, arithmetic
 - (d) Arithmetic, relational, assignment
33. What will the hexadecimal number 0x001B be equal to?
- (a) 21
 - (b) 27
 - (c) 33
 - (d) 23
34. What will the octal number 033 be equal to?
- (a) 21
 - (b) 27
 - (c) 33
 - (d) 23
35. Identify the correct sequence of statements that swaps value of two statements.
- (a) $a=a+b$; $a=a-b$; $b=a-b$
 - (b) $a=a+b$; $b=a-b$; $a=a-b$
 - (c) $a=a-b$; $a=a+b$; $b=a-b$
 - (d) $a=a-b$; $a=a+b$; $b=b-a$
36. Identify the operator that accepts only integer operands.
- (a) +
 - (b) /
 - (c) *
 - (d) %
37. Which of the following operators does not associate from the left?
- (a) +
 - (b) -
 - (c) <
 - (d) =
38. Which of the following shows the ascending order of precedence of these operators: $., !, <, =$
- (a) $., !, <, =$
 - (b) $=, <, !, .$
 - (c) $=, !, <, .$
 - (d) $<, !, =, .$
39. Which among the following is the complement of the operator " $==$ "?
- (a) $/=$
 - (b) $<=$
 - (c) $>=$
 - (d) $!=$

40. Which of these statements does not hold true for the operators ++ and --?
- They are unary operators.
 - The operand can come before or after the operator.
 - They do not require variables as their operands.
 - It cannot be applied to an expression.
41. Which of the following does not depict an arithmetic operation?
- a*=10
 - a!=10
 - a/=10
 - a%=10
42. What will the output of the following C code be?
- ```
#include <stdio.h>
int main()
{
 int x=1, y=1, z;
 z= x++ +y;
 printf ("%d, %d",x,y) ;
}
```
- x= 1, y=1
  - x=1, y=2
  - x=2, y=1
  - x=2, y=2
43. Which of the following depict bitwise exclusive OR?
- &
  - ^
  - <<
  - >>
44. Which of the following operators are used to link related expressions together?
- Bitwise and operator
  - Logical and operator
  - Comma operator
  - Sizeof operator
45. Which of the following is used to input value for variable through keyboard?
- printf
  - scanf
  - get
  - void
46. Which of the following indicate standard input output header file?
- conio.h
  - stdio.h
  - math.h
  - complex.h
47. Which of the following is used to read a character from the standard input unit?
- printf
  - clrscr
  - getchar
  - putchar
48. What is the default data type returned by the function getchar?
- char
  - float
  - int
  - char\*
49. Which of the following values is returned when EOF is encountered?
- 1
  - 1
  - 0
  - 10
50. Which of the following does not use a buffer and returns entered character immediately without waiting for enter key?
- getc()
  - getch()
  - getchar()
  - getche()
51. Which of the following statements does not hold true for the function “scanf”?
- Format specifications contained in the control string must match the arguments in order.
  - The variables to be read must have a filed specification.
  - Scanf reads until a whitespace character is encountered in a numeric specification.
  - Scanf does not ignore line boundaries to look for the next character while searching for a value.
52. What does printf() function return when an error is encountered?
- Positive value
  - Negative value
  - Zero
  - Does not return anything
53. What does the format code “%e” print?
- A decimal integer
  - Print a signed decimal integer
  - Print a floating point value without exponent
  - Print a floating point value in exponent form
54. Which of the following functions is the odd one out?
- printf
  - fprintf
  - putchar
  - scanf
55. Where does the function putchar(a) always output character a to?
- Standard output
  - Screen
  - It is compiler dependent
  - Depends on the standard



## Answers

---

- |         |         |         |         |         |
|---------|---------|---------|---------|---------|
| 1. (c)  | 2. (c)  | 3. (d)  | 4. (b)  | 5. (a)  |
| 6. (b)  | 7. (c)  | 8. (d)  | 9. (b)  | 10. (d) |
| 11. (a) | 12. (c) | 13. (b) | 14. (d) | 15. (d) |
| 16. (c) | 17. (a) | 18. (a) | 19. (c) | 20. (c) |
| 21. (a) | 22. (c) | 23. (b) | 24. (d) | 25. (c) |
| 26. (b) | 27. (c) | 28. (b) | 29. (c) | 30. (d) |
| 31. (c) | 32. (d) | 33. (b) | 34. (b) | 35. (b) |
| 36. (d) | 37. (d) | 38. (c) | 39. (d) | 40. (c) |
| 41. (b) | 42. (c) | 43. (b) | 44. (c) | 45. (b) |
| 46. (b) | 47. (c) | 48. (c) | 49. (a) | 50. (b) |
| 51. (d) | 52. (b) | 53. (d) | 54. (d) | 55. (a) |



## Review Questions

---

1. State whether the following statements are True or False.
  - (a) Every line in a C program should end with a semicolon.
  - (b) Every C program ends with an END word.
  - (c) `main( )` is where the program begins its execution.
  - (d) A line in a program may have more than one statement.
  - (e) The closing brace of the `main( )` in a program is the logical end of the program.
  - (f) The purpose of the header file such as `stdio.h` is to store the source code of a program.
  - (g) Comments cause the computer to print the text enclosed between `/*` and `*/` when executed.
  - (h) Syntax errors will be detected by the compiler.
  - (i) Any valid printable ASCII character can be used in an identifier.
  - (j) All variables must be given a type when they are declared.
  - (k) Variable declarations can appear anywhere in a program.
  - (l) ANSI C treats the variables name and Name to be same.
  - (m) Floating point constants, by default, denote float type values.
  - (n) Like variables, constants have a type.
  - (o) Character constants are coded using double quotes.
  - (p) All arithmetic operators have the same level of precedence.
  - (q) A unary expression consists only one operand with no operators.
  - (r) Associativity is used to decide which of several different expressions is evaluated first.
  - (s) An expression statement is terminated with a period.
  - (t) `If` is an iterative control structure.
  - (u) The C standard function that receives a single character from the keyboard is `getchar`.
  - (v) The `scanf` function cannot be used to read a single character from the keyboard.
  - (w) A program stops its execution when a `break` statement is encountered.
  - (x) The `do...while` statement first executes the loop body and then evaluate the loop control expression.



- (y) An exit-controlled loop is executed a minimum of one time.
  - (z) The three loop expressions used in a for loop header must be separated by commas.
2. Fill in the blanks with appropriate words.
- (a) Every program statement in a C program must end with a \_\_\_\_\_.
  - (b) The \_\_\_\_\_ function is used to display the output on the screen.
  - (c) The \_\_\_\_\_ header file contains mathematical functions.
  - (d) The escape sequence character \_\_\_\_\_ causes the cursor to move to the next line on the screen.
  - (e) \_\_\_\_\_ is the largest value that an unsigned short int type variable can store.
  - (f) A global variable is also known as \_\_\_\_\_ variable.
  - (g) A variable can be made constant by declaring it with the qualifier \_\_\_\_\_ at the time of initialisation.
  - (h) The \_\_\_\_\_ operator is true only when both the operands are true.
  - (i) \_\_\_\_\_ operators are used for testing the bits, or shifting them right or left.
  - (j) The \_\_\_\_\_ statement when executed in a switch statement causes immediate exit from the structure.
  - (k) The expression  $!(x != y)$  can be replaced by the expression \_\_\_\_\_.
  - (l) The \_\_\_\_\_ operator returns the number of bytes the operand occupies.
  - (m) The order of evaluation can be changed by using \_\_\_\_\_ in an expression.
  - (n) \_\_\_\_\_ is used to determine the order in which different operators in an expression are evaluated.
  - (o) In do-while loop, the body is executed at least \_\_\_\_\_ number of time.
- (o) The \_\_\_\_\_ statement is used to skip the remaining part of the statements in a loop.
  - (q) A for loop with the no test condition is known as \_\_\_\_\_ loop.
  - (r) \_\_\_\_\_ should be avoided as part of structured programming approach.
  - (s) n++ is equivalent to the expression \_\_\_\_\_.
3. What is a variable? How are the variables declared in C?
4. List the different types of control statements in C.
5. Why and when do we use the #define directive?
6. Why and when do we use the #include directive?
7. What does void main(void) mean?
8. Distinguish between the following pairs:
- (a) main( ) and void main(void)
  - (b) int main( ) and void main( )
9. Why do we need to use comments in programs?
10. Why is the look of a program is important?
11. Where are blank spaces permitted in a C program?
12. Describe the structure of a C program.
13. What are trigraph characters? How are they useful?
14. What is an unsigned integer constant? What is the significance of declaring a constant unsigned?
15. What is a variable and what is meant by the "value" of a variable?
16. How do variables and symbolic names differ?
17. State the differences between the declaration of a variable and the definition of a symbolic name.
18. A programmer would like to use the word DPR to declare all the double-precision floating point values in his program. How could he achieve this?
19. What are enumeration variables? How are they declared? What is the advantage of using them in a program?



20. How can we use the **getchar**( ) function to read multicharacter strings?
21. How can we use the **putchar**( ) function to output multicharacter strings?
22. What is the purpose of **scanf**( ) function?
23. Describe the purpose of commonly used conversion characters in a **scanf**( ) function.
24. What happens when an input data item contains
  - (a) more characters than the specified field width and
  - (b) fewer characters than the specified field width?
25. What is the purpose of **print**( ) function?
26. Describe the purpose of commonly used conversion characters in a **printf**( ) function.
27. How does a control string in a **printf**( ) function differ from the control string in a **scanf**( ) function?
28. What happens if an output data item contains
  - (a) more characters than the specified field width and
  - (b) fewer characters than the specified field width?
29. How are the unrecognized characters within the control string are interpreted in
  - (a) **scanf** function; and
  - (b) **printf** function?



## Debugging Exercises

1. Find errors, if any, in the following program:

```
/* A simple program
int main()
{
 /* Does nothing */
}
```

2. Find errors, if any, in the following program:

```
#include (stdio.h)
void main(void)
{
 print("Hello C");
}
```

3. Find errors, if any, in the following program:

```
Include <math.h>
main { }
(
 FLOAT X;
 X = 2.5;
 Y = exp(x);
 Print(x,y);
)
```

4. Find errors, if any, in the following declaration statements.

```
Int x;
float letter,DIGIT;
double = p,q
exponent alpha,beta;
m,n,z: INTEGER
```

```
short char c;
long int m; count;
long float temp;
```

5. Identify syntax errors in the following program. After corrections, what output would you expect when you execute it?

```
#define PI 3.14159
main()
{
 int R,C; /* R-Radius of
 circle
 float perimeter; /* Circumference
 of circle */
 float area; /* Area of
 circle */

 C = PI
 R = 5;
 Perimeter = 2.0 * C *R;
 Area = C*R*R;
 printf("%f", "%d",
 &perimeter,&area)
}
```

6. What is the error, if any, in the following segment?

```
int x = 10 ;
float y = 4.25 ;
x = y%x ;
```



7. What is the error in each of the following statements?
  - (a) `if (m == 1 & n != 0)`  
    `printf("OK");`
  - (b) `if (x = < 5)`  
    `printf("Jump");`
8. Find errors, if any, in the following assignment statements and rectify them.
  - (a) `x = y = z = 0.5, 2.0, -5.75;`
  - (b) `m = ++a * 5;`
  - (c) `y = sqrt(100);`
  - (d) `p * = x/y;`
  - (e) `s = /5;`
  - (f) `a = b++ -c*2`
9. State errors, if any, in the following input statements.
  - (a) `scanf("%c%f%d", city, &price, &year);`
  - (b) `scanf("%s%d", city, amount);`
  - (c) `scanf("%f, %d, &amount, &year);`
  - (d) `scanf("\n"%f", root);`
  - (e) `scanf("%c %d %ld", *code, &count, Root);`



## Programming Exercise

1. Give the output of the following code:

```
float c = 34.78650;
printf ("%6.2f" , c);
```

2. Write a program in C to find the value of  $y$  using the relation  $y = x_2 + 2x - 1$ .
3. Write a program in C to find the sum and the average of three numbers.
4. Write a program in C to calculate simple interest.
5. Write a program in C to convert the temperature from  $^{\circ}\text{C}$  to  $^{\circ}\text{F}$ .
6. Write a program in C to evaluate the series  $S = 1 + 2 * 1 + 3 * 2 + \dots N * N - 1$ .
7. 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.
8. 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).
9. Write a program that prints the even numbers from 1 to 100.
10. 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.

11. 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
```

12. 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.
13. 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.
14. 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.
15. Write a program to illustrate the use of `typedef` declaration in a program.



## 2.88 Computer Programming

16. Write a program to illustrate the use of symbolic constants in a real-life application.
17. Write a program to display the equation of a line in the form

$$ax + by = c$$

for  $a = 5$ ,  $b = 8$  and  $c = 18$ .

18. Write a program that will print your mailing address in the following form:

First line : Name

Second line : Door No, Street

Third line : City, Pin code

19. Write a program to output the following multiplication table:

$$5 \times 1 = 5$$

$$5 \times 2 = 10$$

$$5 \times 3 = 15$$

$$\begin{array}{cc} \bullet & \bullet \\ \bullet & \bullet \end{array}$$

$$5 \times 10 = 50$$

20. Given the values of three variables  $a$ ,  $b$  and  $c$ , write a program to compute and display the value of  $x$ , where

$$x = \frac{a}{b - c}$$

Execute your program for the following values:

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

(b)  $a = 300$ ,  $b = 70$ ,  $c = 70$

Comment on the output in each case.

21. Given the radius of a circle, write a program to compute and display its area. Use a symbolic constant to define the  $\pi$  value and assume a suitable value for radius.
22. Given two integers 20 and 10, write a program that uses a function `add()` to add these two numbers and `sub()` to find the difference of these two numbers and then display the sum and difference in the following form:

$$20 + 10 = 30$$

$$20 - 10 = 10$$

23. Write a program using one print statement to print the pattern of asterisks as shown below:

```
*
* *
* * *
* * * *
```

24. Write a program that will print the following figure using suitable characters.



25. Area of a triangle is given by the formula

$$A = \sqrt{S(S-a)(S-b)(S-c)}$$

Where  $a$ ,  $b$  and  $c$  are sides of the triangle and  $2S = a + b + c$ . Write a program to compute the area of the triangle given the values of  $a$ ,  $b$  and  $c$ .

26. Write a program to display the following simple arithmetic calculator

$x =$

$y =$

sum

Difference =

Product =

Division =

27. Distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$  is governed by the formula

$$D^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Write a program to compute  $D$  given the coordinates of the points.

28. A point on the circumference of a circle whose center is  $(o, o)$  is  $(4,5)$ . Write a program to compute perimeter and area of the circle.
29. The line joining the points  $(2,2)$  and  $(5,6)$  which lie on the circumference of a circle is the diameter of the circle. Write a program to compute the area of the circle.
30. Given the values of the variables  $x$ ,  $y$  and  $z$ , write a program to rotate their values such that  $x$  has the value of  $y$ ,  $y$  has the value of  $z$ , and  $z$  has the value of  $x$ .
31. Write a program that reads a floating-point number and then displays the right-most digit of the integral part of the number.
32. Modify the above program to display the two right-most digits of the integral part of the number.
33. Write a program that will obtain the length and width of a rectangle from the user and compute its area and perimeter.



34. Given an integer number, write a program that displays the number as follows:

First line : all digits  
 Second line : all except first digit  
 Third line : all except first two digits

.....

Last line : The last digit

For example, the number 5678 will be displayed as:

5 6 7 8

6 7 8

7 8

8

35. The straight-line method of computing the yearly depreciation of the value of an item is given by

Depreciation

$$= \frac{\text{Purchase Price} - \text{Salvage Value}}{\text{Years of Service}}$$

Write a program to determine the salvage value of an item when the purchase price, years of service, and the annual depreciation are given.

36. Write a program that will read a real number from the keyboard and print the following output in one line:

Smallest integer    The given    Largest integer  
 not less than       number       not greater than  
 the number                            the number

37. The total distance travelled by a vehicle in  $t$  seconds is given by

$$\text{distance} = ut + (at^2)/2$$

Where  $u$  is the initial velocity (metres per second),  $a$  is the acceleration (metres per second<sup>2</sup>). Write a program to evaluate the distance travelled at regular intervals of time, given the values of  $u$  and  $a$ . The program should provide the flexibility to the user to select his own time intervals and repeat the calculations for different values of  $u$  and  $a$ .

38. In inventory management, the Economic Order Quantity for a single item is given by

$$\text{EOQ} = \sqrt{\frac{2 \times \text{demand rate} \times \text{setup costs}}{\text{holding cost per item per unit time}}}$$

and the optimal Time Between Orders

$$\text{TBO} = \sqrt{\frac{2 \times \text{setup costs}}{\text{demand rate} \times \text{holding cost per unit time}}}$$

Write a program to compute EOQ and TBO, given demand rate (items per unit time), setup costs (per order), and the holding cost (per item per unit time).

39. For a certain electrical circuit with an inductance  $L$  and resistance  $R$ , the damped natural frequency is given by

$$\text{Frequency} = \sqrt{\frac{1}{LC} - \frac{R^2}{4C^2}}$$

It is desired to study the variation of this frequency with  $C$  (capacitance). Write a program to calculate the frequency for different values of  $C$  starting from 0.01 to 0.1 in steps of 0.01.

40. Write a program to read a four digit integer and print the sum of its digits.

Hint: Use / and % operators.

41. Write a program to print the size of various data types in C.

42. Given three values, write a program to read three values from keyboard and print out the largest of them without using **if** statement.

43. Write a program to read two integer values  $m$  and  $n$  and to decide and print whether  $m$  is a multiple of  $n$ .

44. Write a program to read three values using **scanf** statement and print the following results:

- Sum of the values
- Average of the three values
- Largest of the three
- Smallest of the three

45. The cost of one type of mobile service is Rs. 250 plus Rs. 1.25 for each call made over and above 100 calls. Write a program to read customer codes and calls made and print the bill for each customer.

46. Write a program to print a table of **sin** and **cos** functions for the interval from 0 to 180 degrees in increments of 15 as shown here.



| x (degrees) | sin (x) | cos (x) |
|-------------|---------|---------|
| 0           | .....   | .....   |
| 15          | .....   | .....   |
| ...         |         |         |
| 180         | .....   | .....   |

47. Write a program to compute the values of square-roots and squares of the numbers 0 to 100 in steps 10 and print the output in a tabular form as shown below.

| Number | Square-root | Square |
|--------|-------------|--------|
| 0      | 0           | 0      |
| 100    | 10          | 10000  |

48. Write a program that determines whether a given integer is odd or even and displays the number and description on the same line.
49. Write a program to illustrate the use of cast operator in a real life situation.
50. Given the string "WORDPROCESSING", write a program to read the string from the terminal and display the same in the following formats:
- WORD PROCESSING
  - WORD  
PROCESSING
  - W.P.
51. Write a program to read the values of x and y and print the results of the following expressions in one line:
- $\frac{x+y}{x-y}$
  - $\frac{x+y}{2}$
  - $(x+y)(x-y)$
52. Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form:
- 35.7      50.21      - 23.73      - 46.45
53. Write a program that reads 4 floating point values in the range, 0.0 to 20.0, and prints a horizontal bar chart to represent these values using the character \* as the fill character. For the purpose of the chart, the values may be rounded off to the nearest integer. For example, the value 4.36 should be represented as follows.

```

* * * *
* * * * 4.36
* * * *

```

Note that the actual values are shown at the end of each bar.

54. Write an interactive program to demonstrate the process of multiplication. The program should ask the user to enter two two-digit integers and print the product of integers as shown below.

```

 45
 ×
 37

7 × 45 is 315
3 × 45 is 135
Add them ---
 1665

```

55. Write a program to read three integers from the keyboard using one **scanf** statement and output them on one line using:
- three **printf** statements,
  - only one **printf** with conversion specifiers, and
  - only one **printf** without conversion specifiers.
56. Write a program that prints the value 10.45678 in exponential format with the following specifications:
- correct to two decimal places;
  - correct to four decimal places; and
  - correct to eight decimal places.
57. Write a program to print the value 345.6789 in fixed-point format with the following specifications:
- correct to two decimal places;
  - correct to five decimal places; and
  - correct to zero decimal places.
58. Write a program to read the name ANIL KUMAR GUPTA in three parts using the **scanf** statement and to display the same in the following format using the **printf** statement.
- ANIL K. GUPTA      (b) A.K. GUPTA
  - GUPTA A.K.
59. Write a program to read and display the following table of data.

| Name  | Code  | Price   |
|-------|-------|---------|
| Fan   | 67831 | 1234.50 |
| Motor | 450   | 5786.70 |

The name and code must be left-justified and price must be right-justified.



---

# 3 Decision Making, Branching and Looping

---

## CHAPTER OUTLINE

|                                           |                                         |                         |
|-------------------------------------------|-----------------------------------------|-------------------------|
| 3.1 Introduction                          | 3.4 The ?: Operator                     | 3.7 The While Statement |
| 3.2 Decision Making with If Statement     | 3.5 Decision Making with Goto Statement | 3.8 The Do Statement    |
| 3.3 Decision Making with Switch Statement | 3.6 Introduction to Looping Procedure   | 3.9 The For Statement   |
|                                           |                                         | 3.10 Jumps In Loops     |
|                                           |                                         | 3.11 Case Studies       |

## 3.1 INTRODUCTION

We have seen that a C program is a set of statements which 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:

1. **if** statement
2. **switch** statement
3. Conditional operator statement
4. **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*.

## 3.2 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. 3.1.



## 3.2 Computer Programming

Some examples of decision making, using **if** statements are:

1. **if** (bank balance is zero)  
    borrow money
2. **if** (room is dark)  
    put on lights
3. **if** (code is 1)  
    person is male
4. **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:

1. Simple **if** statement
2. **if.....else** statement
3. Nested **if....else** statement
4. **else if** ladder.

We shall discuss each one of them in the next few sections.

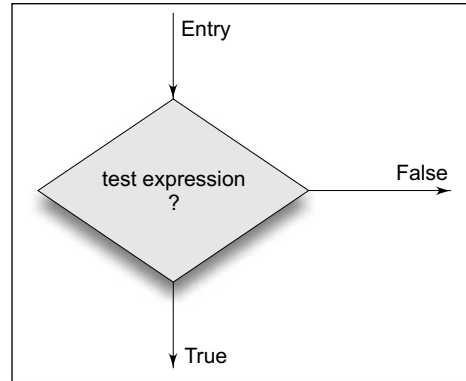


Fig. 3.1 Two-way branching

### 3.2.1 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. 3.2.

Consider the following segment of a program that is written for processing of marks obtained in an entrance examination.

```
.....
.....
if (category == SPORTS)
{
 marks = marks + bonus_marks;
}
printf("%f", marks);
.....
.....
```

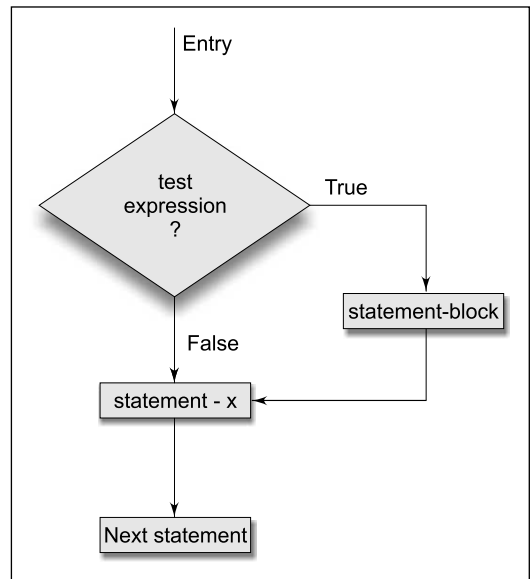


Fig. 3.2 Flowchart of simple if control



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 3.1** The program in Fig. 3.3 displays a number if user enters a negative number and if a user enters positive number, then that number won't be displayed.

---

```

Program
#include <stdio.h>
int main()
{
 int number;
 clrscr();
 printf("Enter an integer: ");
 scanf("%d", &number);
 if (number < 0)
 /*if condition is true inside if() statement then only it executes*/
 {
 printf("You entered %d.\n", number);
 }
 getch();
 return 0;
}
Output
Enter an integer: -96
You entered -96

```

**Fig. 3.3** Use of if for displaying number

---

**Example 3.2** The program in Fig. 3.4 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.

---

The program given in Fig. 3.4 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

```

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);
 }
}

```



### 3.4 Computer Programming

```
 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. 3.4** Illustration of simple if statement

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. Example 3.3 illustrates this.

---

**Example 3.3** The program in Fig. 3.5 counts the number of boys whose weight is less than 50 kg and height is greater than 170 cm.

---

The program has to test 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;
```

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.

```
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. 3.5** Use of if for counting

### 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  $!(x \&\& y \parallel !z)$ . 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

```



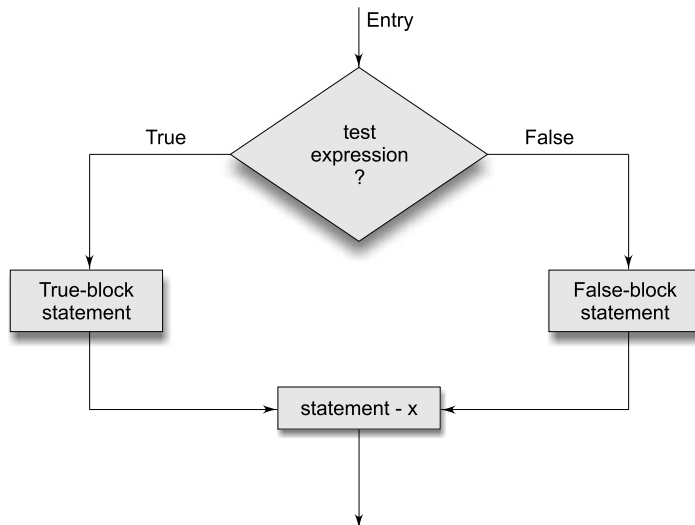
### 3.6 Computer Programming

#### 3.2.2 The If.....Else Statement

The **if...else** statement is an extension of the simple **if** statement. The general form is

```
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. 3.6. In both the cases, the control is transferred subsequently to the statement-x.



**Fig. 3.6** Flowchart of if.....else control

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:

```
.....
.....
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;
xxxxxxxxxx
.....

```

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**.

Consider the program given in Fig. 3.4. When the value (c-d) is zero, the ratio is not calculated and the program stops without any message. In such cases we may not know whether the program stopped due to a zero value or some other error. This program can be improved by adding the **else** clause as follows:

```

.....
.....
if (c-d != 0)
{
 ratio = (float)(a+b)/(float)(c-d);
 printf("Ratio = %f\n", ratio);
}
else
 printf("c-d is zero\n");
.....
.....

```

**Example 3.4** A program to evaluate the power series.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^2}{3!} + \dots + \frac{x^n}{n!}, 0 < x < 1$$

is given in Fig. 3.7. It uses if.....else to test the accuracy.

The power series contains the recurrence relationship of the type

$$T_n = T_{n-1} \left( \frac{x}{n} \right) \text{ for } n > 1$$

$$T_1 = x \text{ for } n = 1$$

$$T_0 = 1$$



### 3.8 Computer Programming

If  $T_{n-1}$  (usually known as *previous term*) is known, then  $T_n$  (known as *present term*) can be easily found by multiplying the previous term by  $x/n$ . Then

$$e^x = T_0 + T_1 + T_2 + \dots + T_n = \text{sum}$$

```
Program
#define ACCURACY 0.0001
main()
{
 int n, count;
 float x, term, sum;
 printf("Enter value of x:");
 scanf("%f", &x);
 n = term = sum = count = 1;
 while (n <= 100)
 {
 term = term * x/n;
 sum = sum + term;
 count = count + 1;
 if (term < ACCURACY)
 n = 999;
 else
 n = n + 1;
 }
 printf("Terms = %d Sum = %f\n", count, sum);
}
```

Output

```
Enter value of x:0
Terms = 2 Sum = 1.000000
Enter value of x:0.1
Terms = 5 Sum = 1.105171
Enter value of x:0.5
Terms = 7 Sum = 1.648720
Enter value of x:0.75
Terms = 8 Sum = 2.116997
Enter value of x:0.99
Terms = 9 Sum = 2.691232
Enter value of x:1
Terms = 9 Sum = 2.718279
```

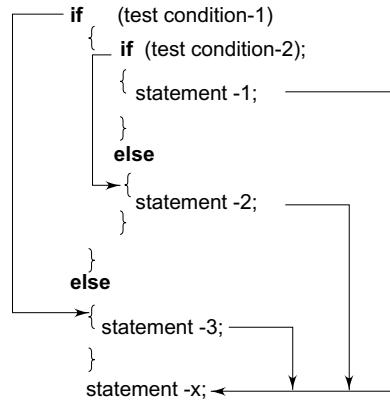
**Fig. 3.7** Illustration of *if...else* statement

The program uses **count** to count the number of terms added. The program stops when the value of the term is less than 0.0001 (**ACCURACY**). Note that when a term is less than **ACCURACY**, the value of  $n$  is set equal to 999 (a number higher than 100) and therefore the **while** loop terminates. The results are printed outside the **while** loop.



### 3.2.3 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:



The logic of execution is illustrated in Fig. 3.8. 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.

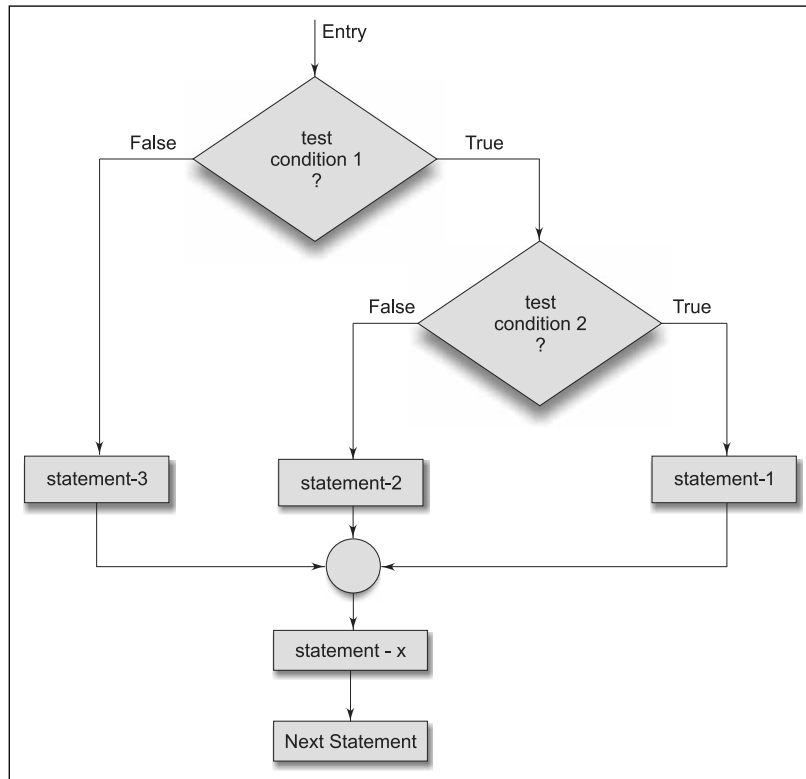


Fig. 3.8 Flow chart of nested if...else statements



### 3.10 Computer Programming

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 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 above 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.

Consider another alternative, which also looks correct:

```
if (sex is female)
{
 if (balance > 5000)
 bonus = 0.05 * balance;
}
else
 bonus = 0.02 * balance;
balance = balance + bonus;
```

In this case, **else** is associated with the outer **if** and therefore bonus is calculated for the male account holders. However, bonus for the female account holders, whose balance is equal to or less than 5000 is not calculated because of the missing **else** option for the inner **if**.



---

**Example 3.5** The program in Fig. 3.9 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. 3.9** Selecting the largest of three numbers

### 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

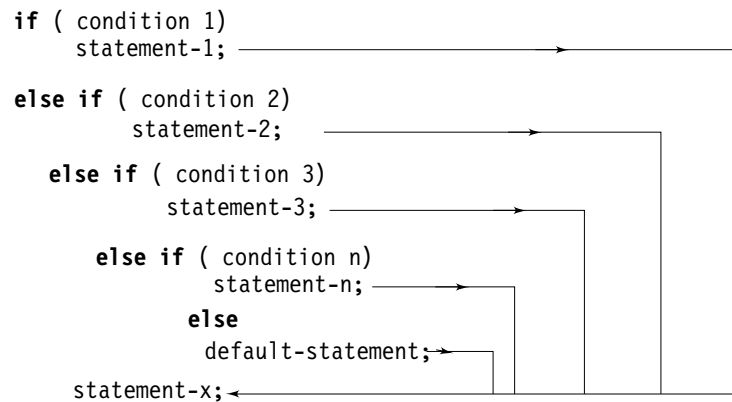
**“else is always paired with the most recent unpaired if”**

#### 3.2.4 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 following general form:



### 3.12 Computer Programming



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. Fig. 3.10 shows the logic of execution of **else if** ladder statements.

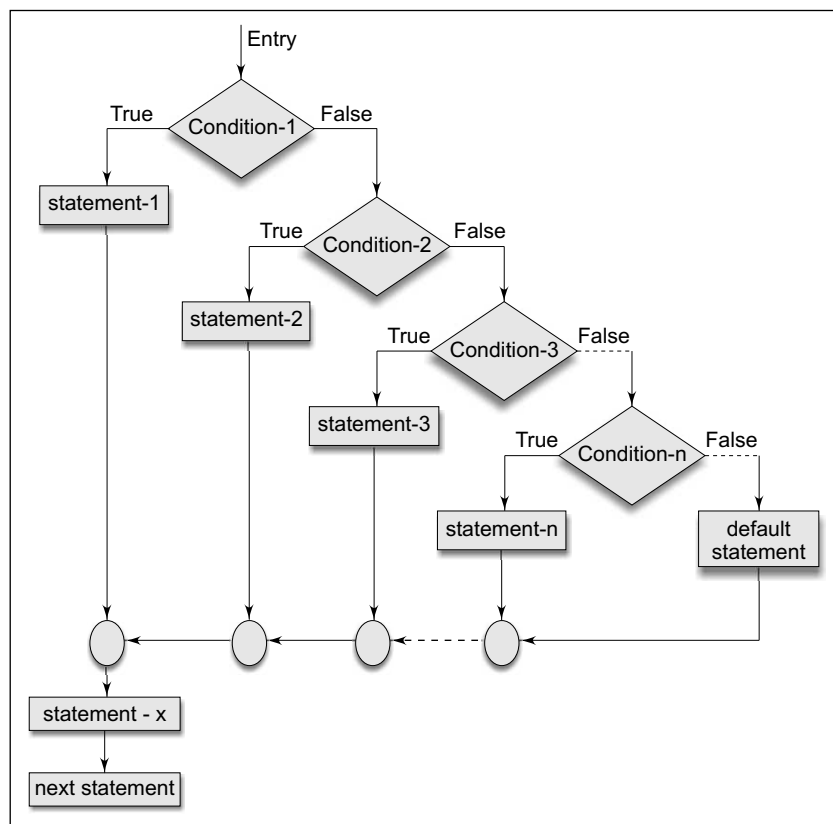


Fig. 3.10 Flow chart of else..if ladder



Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

| Average marks | Grade           |
|---------------|-----------------|
| 80 to 100     | Honours         |
| 60 to 79      | First Division  |
| 50 to 59      | Second Division |
| 40 to 49      | Third Division  |
| 0 to 39       | Fail            |

This grading can be done using the **else if** ladder as follows:

```
if (marks > 79)
 grade = "Honours";
else if (marks > 59)
 grade = "First Division";
else if (marks > 49)
 grade = "Second Division";
else if (marks > 39)
 grade = "Third Division";
else
 grade = "Fail";
printf ("%s\n", grade);
```

Consider another example given below:

```


if (code == 1)
 colour = "RED";
else if (code == 2)
 colour = "GREEN";
else if (code == 3)
 colour = "WHITE";
else
 colour = "YELLOW";


```

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.

```
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.



**Example 3.6** An electric power distribution company charges its domestic consumers as follows:

| <i>Consumption Units</i> | <i>Rate of Charge</i>                        |
|--------------------------|----------------------------------------------|
| 0 – 200                  | Rs. 0.50 per unit                            |
| 201 – 400                | Rs. 100 plus Rs. 0.65 per unit excess of 200 |
| 401 – 600                | Rs. 230 plus Rs. 0.80 per unit excess of 400 |
| 601 and above            | Rs. 390 plus Rs. 1.00 per unit excess of 600 |

The program in Fig. 3.11 reads the customer number and power consumed and prints the amount to be paid by the customer.

```

Program
main()
{
 int units, custnum;
 float charges;
 printf("Enter CUSTOMER NO. and UNITS consumed\n");
 scanf("%d %d", &custnum, &units);
 if (units <= 200)
 charges = 0.5 * units;
 else if (units <= 400)
 charges = 100 + 0.65 * (units - 200);
 else if (units <= 600)
 charges = 230 + 0.8 * (units - 400);
 else
 charges = 390 + (units - 600);
 printf("\n\nCustomer No: %d: Charges = %.2f\n",
 custnum, charges);
}

```

Output

```

Enter CUSTOMER NO. and UNITS consumed 101 150
Customer No:101 Charges = 75.00
Enter CUSTOMER NO. and UNITS consumed 202 225
Customer No:202 Charges = 116.25
Enter CUSTOMER NO. and UNITS consumed 303 375
Customer No:303 Charges = 213.75
Enter CUSTOMER NO. and UNITS consumed 404 520
Customer No:404 Charges = 326.00
Enter CUSTOMER NO. and UNITS consumed 505 625
Customer No:505 Charges = 415.00

```

**Fig. 3.11** Illustration of *else..if ladder*

## Rules for Indentation

When using control structures, a statement often controls many other statements that follow it. 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.

### 3.3 DECISION MAKING WITH 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 as shown below:

```
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**.



### 3.16 Computer Programming

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. 3.12.

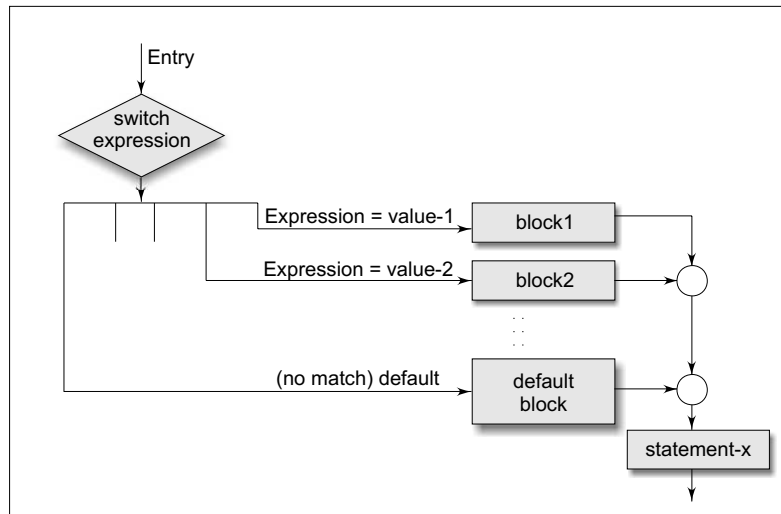


Fig. 3.12 Selection process of the switch statement

The **switch** statement can be used to grade the students as discussed in section 3.2. This is illustrated below:

```


index = marks/10
switch (index)
{
 case 10:
 case 9:
 case 8:
 grade = "Honours";
 break;
 case 7:
 case 6:
 grade = "First Division";
 break;
 case 5:
 grade = "Second Division";
 break;
 case 4:
 grade = "Third Division";
 break;
 default:
 grade = "Fail";
```



```

 break;
 }
 printf("%s\n", grade);


```

Note that we have used a conversion statement

```
index = marks / 10;
```

where, index is defined as an integer. The variable index takes the following integer values.

| <b>Marks</b> | <b>Index</b> |
|--------------|--------------|
| 100          | 10           |
| 90 - 99      | 9            |
| 80 - 89      | 8            |
| 70 - 79      | 7            |
| 60 - 69      | 6            |
| 50 - 59      | 5            |
| 40 - 49      | 4            |
| .            | .            |
| .            | .            |
| 0            | 0            |

This segment of the program illustrates two important features. First, it uses empty cases. The first three cases will execute the same statements

```

 grade = "Honours";
 break;

```

Same is the case with case 7 and case 6. Second, default condition is used for all other cases where marks is less than 40.

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;
 }

```



### 3.18 Computer Programming

```
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.

#### Rules for 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 colon.
- 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.

---

**Example 3.7** Write a complete C program that reads a value in the range of 1 to 12 and print the name of that month and the next month. Print error for any other input value.

---

Program

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

```

**Fig. 3.13** Program to read and print name of months in the range of 1 and 12

---

**Example 3.8** Program in Fig. 3.14 illustrates a simple calculator to perform addition, subtraction, multiplication or division depending upon the input from user.

---

```

Program
#include <stdio.h>
int main()
{
 char operator;
 double firstNumber,secondNumber;
 clrscr();
 printf("Enter an operator (+, -, *, /): ");
 scanf("%c", &operator);
 printf("Enter two operands: ");
 scanf("%lf %lf",&firstNumber, &secondNumber);
 switch(operator)
 /*switch jumps on the case depending upon the value of operator inside switch*/
 {
 case '+':
 printf("%.1lf + %.1lf = %.1lf",firstNumber, secondNumber,
firstNumber+secondNumber);
 break;
 case '-':
 printf("%.1lf - %.1lf = %.1lf",firstNumber, secondNumber, firstNumber-
secondNumber);
 break;
 case '*':
 printf("%.1lf * %.1lf = %.1lf",firstNumber, secondNumber,
firstNumber*secondNumber);
 break;
 case '/':
 printf("%.1lf / %.1lf = %.1lf",firstNumber, secondNumber, firstNumber/
firstNumber);
 break;
 default:
 printf("Error! operator is not correct");
 }
}

```



### 3.20 Computer Programming

```
 getch();
 return 0;
}
Output
Enter an operator (+, -, *,): -
Enter two operands: 32.5
12.4
32.5 - 12.4 = 20.1
```

Fig. 3.14 Program illustrating simple calculator

## 3.4 THE ? : OPERATOR

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and :, and takes three operands. This operator is popularly known as the *conditional operator*. The general form of use of the conditional operator is as follows:

***conditional expression ? expression1 : expression2***

The *conditional expression* is evaluated first. If the result is non-zero, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

```
if (x < 0)
 flag = 0;
else
 flag = 1;
```

can be written as

`flag = ( x < 0 ) ? 0 : 1;`

Consider the evaluation of the following function:

$y = 1.5x + 3$  for  $x \leq 2$

$y = 2x + 5$  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 as

`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.

---

**Example 3.9** An employee can apply for a loan at the beginning of every six months, but he will be sanctioned the amount according to the following company rules:

**Rule 1:** An employee cannot enjoy more than two loans at any point of time.

**Rule 2:** Maximum permissible total loan is limited and depends upon the category of the employee.

A program to process loan applications and to sanction loans is given in Fig. 3.15.

---

Program

```
#define MAXLOAN 50000
main()
{
 long int loan1, loan2, loan3, sancloan, sum23;
 printf("Enter the values of previous two loans:\n");
 scanf(" %ld %ld", &loan1, &loan2);
 printf("\nEnter the value of new loan:\n");
 scanf(" %ld", &loan3);
 sum23 = loan2 + loan3;
 sancloan = (loan1>0)? 0 : ((sum23>MAXLOAN)?
 MAXLOAN - loan2 : loan3);
 printf("\n\n");
 printf("Previous loans pending:\n%ld %ld\n", loan1, loan2);
 printf("Loan requested = %ld\n", loan3);
 printf("Loan sanctioned = %ld\n", sancloan);
}
```

Output

```
Enter the values of previous two loans:
0 20000
Enter the value of new loan:
45000
Previous loans pending:
0 20000
Loan requested = 45000
Loan sanctioned = 30000
Enter the values of previous two loans:
1000 15000
Enter the value of new loan:
25000
Previous loans pending:
1000 15000
Loan requested= 25000
Loan sanctioned= 0
```

**Fig. 3.15** Illustration of the conditional operator



### 3.22 Computer Programming

The program uses the following variables:

|                 |   |                                   |
|-----------------|---|-----------------------------------|
| <b>loan3</b>    | - | present loan amount requested     |
| <b>loan2</b>    | - | previous loan amount pending      |
| <b>loan1</b>    | - | previous to previous loan pending |
| <b>sum23</b>    | - | sum of loan2 and loan3            |
| <b>sancloan</b> | - | loan sanctioned                   |

The rules for sanctioning new loan are:

1. loan1 should be zero.
2. loan2 + loan3 should not be more than MAXLOAN.

Note the use of **long int** type to declare variables.

#### Some Guidelines for Writing Multiway Selection Statements

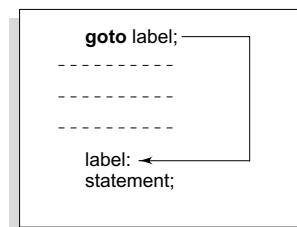
Complex multiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are some guidelines that would help improve readability and facilitate maintenance.

- Avoid compound negative statements. Use positive statements wherever possible.
- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.
- Use the most probable condition first. This will eliminate unnecessary tests, thus improving the efficiency of the program.
- The choice between the nested if and switch statements is a matter of individual's preference. A good rule of thumb is to use the switch when alter-native paths are three to ten.
- Use proper indentations (See Rules for Indentation).
- Have the habit of using default clause in switch statements.
- Group the case labels that have similar actions.

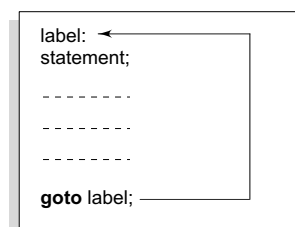
### 3.5 DECISION MAKING WITH GOTO STATEMENT

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 below:



Forward jump



Backward jump



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 will jump 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. Such infinite loops should be avoided. Example 3.10 illustrates how such infinite loops can be eliminated.

---

**Example 3.10** Program presented in Fig. 3.16 illustrates the use of the **goto** statement. The program 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.

---

```
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)
```



### 3.24 Computer Programming

```
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. 3.16** Use of the goto statement

Another use of the **goto** statement is to transfer the control out of a loop (or nested loops) when certain peculiar conditions are encountered. Example:

```


while (-----)
{
 for (-----)
 {

 if (-----)goto end_of_program;

 }

}
end_of_program: ←
```

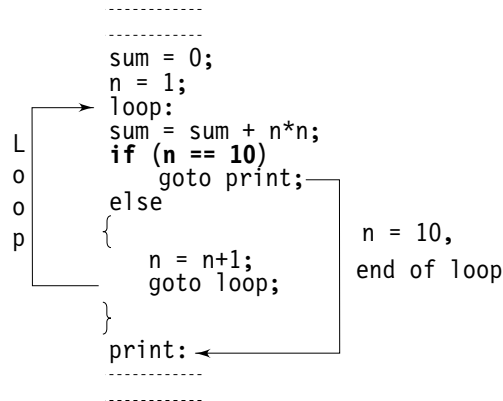
Jumping  
out of  
loops

We should try to avoid using **goto** as far as possible. But there is nothing wrong, if we use it to enhance the readability of the program or to improve the execution speed.



### 3.6 INTRODUCTION TO LOOPING PROCEDURE

It is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the **if** statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10, we can write a program using the **if** statement as follows:



This program does the following things:

1. Initializes the variable **n**.
2. Computes the square of **n** and adds it to **sum**.
3. Tests the value of **n** to see whether it is equal to 10 or not. If it is equal to 10, then the program prints the results.
4. If **n** is less than 10, then it is incremented by one and the control goes back to compute the **sum** again.

The program evaluates the statement

```
sum = sum + n*n;
```

10 times. That is, the loop is executed 10 times. This number can be increased or decreased easily by modifying the relational expression appropriately in the statement `if (n == 10)`. On such occasions where the exact number of repetitions are known, there are more convenient methods of looping in C. These looping capabilities enable us to develop concise programs containing repetitive processes without the use of **goto** statements.

In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A *program loop* therefore consists of two segments, one known as the *body of the loop* and the other known as the *control statement*. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

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 chart in Fig. 3.17 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.



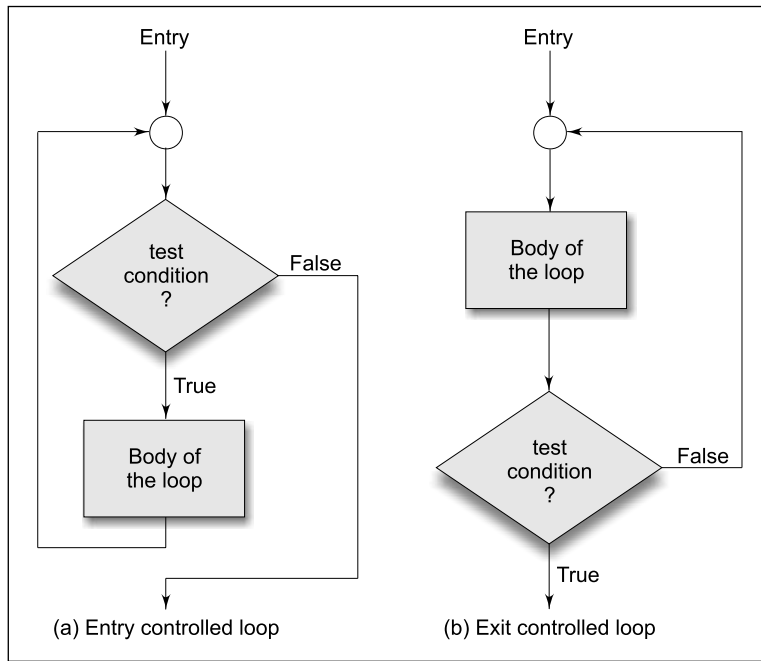


Fig. 3.17 Loop control structures

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 initialization 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.

The features and applications of each of these statements are discussed in the following sections.

### 3.6.1 Sentinel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops may be classified into following two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops



When we know in advance exactly how many times the loop will be executed, we use a *counter-controlled loop*. We use a control variable known as *counter*. The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. A counter-controlled loop is sometimes called *definite repetition loop*.

In a *sentinel-controlled loop*, a special value called a *sentinel* value is used to change the loop control expression from true to false. For example, when reading data we may indicate the “end of data” by a special value, like -1 and 999. The control variable is called **sentinel** variable. A sentinel-controlled loop is often called *indefinite repetition loop* because the number of repetitions is not known before the loop begins executing.

### 3.7 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.

We can rewrite the program loop discussed above as follows:

```

=====
sum = 0;
n = 1; /* Initialization */
while(n <= 10) /* Testing */
{
 sum = sum + n * n;
 n = n+1; /* Incrementing */
}
printf("sum = %d\n", sum);
=====

```

loop →

The body of the loop is executed 10 times for  $n = 1, 2, \dots, 10$ , each time adding the square of the value of  $n$ , which is incremented inside the loop. The test condition may also be written as  $n < 11$ ; the result would be the same. This is a typical example of counter-controlled loops. The variable  $n$  is called **counter** or **control variable**.



### 3.28 Computer Programming

Another example of **while** statement, which uses the keyboard input is shown below:

```
=====
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;. This is a typical example of sentinel-controlled loops. The character constant 'y' is called *sentinel* value and the variable **character** is the condition variable, which often referred to as the *sentinel variable*.

---

#### Example 3.11 A program to evaluate the equation

$$y = x^n$$

when n is a non-negative integer, is given in Fig. 3.18

---

The variable **y** is initialized to 1 and then multiplied by **x**, n times using the **while** loop. The loop control variable **count** is initialized outside the loop and incremented inside the loop. When the value of **count** becomes greater than **n**, the control exists the loop.

```
Program
main()
{
 int count, n;
 float x, y;
 vprintf("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. 3.18** Program to compute  $x$  to the power  $n$  using while loop

---

**Example 3.12** Write a program which accepts a positive number from the user and displays “Hello” message and reduce input by 1 every time until user input is equal to 1.

---

Program

```
#include <stdio.h>
int main()
{
 int i;
 clrscr();
 printf("Enter a positive number:");
 scanf("%d",&i);
 while (i > 0)
 /*while loop executes until condition becomes false*/
 {
 printf("Hello\n");
 i = i -1;
 if(i == 1)
 {
 break;
 }
 }
 getch();
 return 0;
}
```

Output

```
Enter a positive number: 6

Hello
Hello
Hello
Hello
Hello
```

**Fig. 3.19** Program displaying Hello Message

### 3.8 THE DO STATEMENT

The **while** loop construct that 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 form:



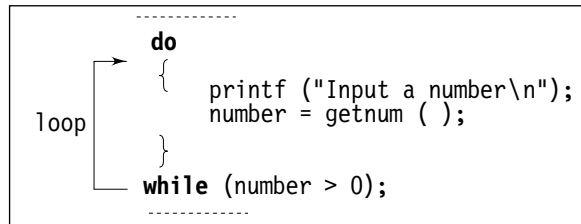
### 3.30 Computer Programming

```
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*.

A simple example of a **do...while** loop is:



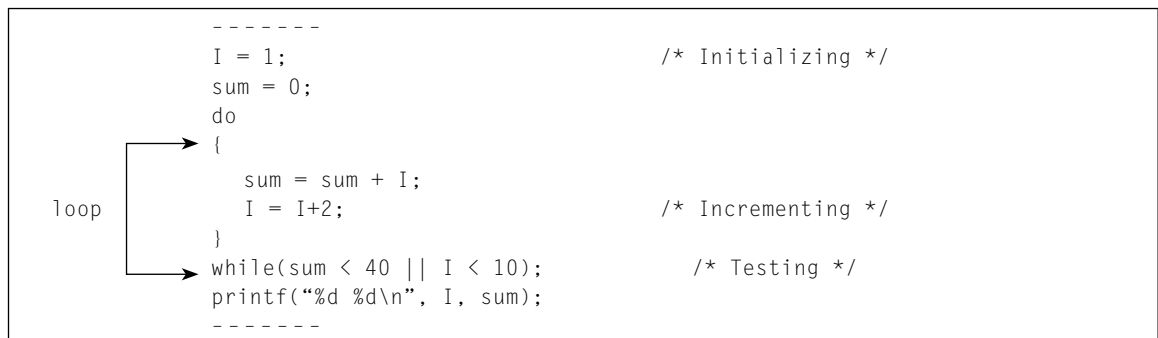
This segment of a program reads a number from the keyboard until a zero or a negative number is keyed in, and assigned to the *sentinel* variable **number**.

The test conditions may have compound relations as well. For instance, the statement

```
while (number > 0 && number < 100);
```

in the above example would cause the loop to be executed as long as the number keyed in lies between 0 and 100.

Consider another example:



The loop will be executed as long as one of the two relations is true.



**Example 3.13** A program to print the multiplication table from 1 x 1 to 12 x 10 as shown below is given in Fig. 3.20.

|    |   |   |    |       |     |
|----|---|---|----|-------|-----|
| 1  | 2 | 3 | 4  | ..... | 10  |
| 2  | 4 | 6 | 8  | ..... | 20  |
| 3  | 6 | 9 | 12 | ..... | 30  |
| 4  |   |   |    | ..... | 40  |
| -  |   |   |    |       |     |
| -  |   |   |    |       |     |
| -  |   |   |    |       |     |
| 12 | . | . |    | ..... | 120 |

This 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.

Program:

```
#define COLMAX 10
#define ROWMAX 12
main()
{
 int row,column, y;
 row = 1;
 printf(" MULTIPLICATION TABLE \n");
 printf("-----\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*/
 printf("-----\n");
}
```

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 |



### 3.32 Computer Programming

|    |    |    |    |    |    |    |    |     |     |
|----|----|----|----|----|----|----|----|-----|-----|
| 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. 3.20** Printing of a multiplication table using do...while loop

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.

---

#### Example 3.14 Write a program to add numbers until user enters zero using do statement.

---

##### Program

```
#include <stdio.h>
int main()
{
 double number, sum = 0;
 clrscr();
 // loop body is executed at least once
 do
 /*although condition is false inside while,do statement executes at least once*/
 {
 printf("Enter a number:");
 scanf("%lf", &number);
 sum += number;
 }
 while(number != 0.0);

 printf("Sum = %.2lf",sum);
 getch();
 return 0;
}
```

##### Output

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: 5
Enter a number: 6
Enter a number: 7
Enter a number: 8
Enter a number: 9
Enter a number: 10
Enter a number: 0
Sum = 55
```

**Fig. 3.21** Program to add numbers



## 3.9 THE FOR STATEMENT

### 3.9.1 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. *Initialization* 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.

**NOTE:** C99 enhances the **for** loop by allowing declaration of variables in the initialization permits portion. See Appendix 1 for details.

Consider the following segment of a program:

```

 for (x = 0 ; x <= 9 ; x = x+1)
loop {
 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*. For example, the loop discussed above 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.



### 3.34 Computer Programming

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,

```
for (x = 9; x < 9; x = x-1)
 printf("%d", x);
```

will never be executed because the test condition fails at the very beginning itself.

Let us again consider the problem of sum of squares of integers. This problem can be coded using the **for** statement as follows:

```

sum = 0;
for (n = 1; n <= 10; n = n+1)
{
 sum = sum+ n*n;
}
printf("sum = %d\n", sum);

```

The body of the loop

```
sum = sum + n*n;
```

is executed 10 times for  $n = 1, 2, \dots, 10$  each time incrementing the **sum** by the square of the value of  $n$ .

One of the important points about the **for** loop is that all the three actions, namely *initialization*, *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 3.1.

**TABLE 3.1** Comparison of the three loops

| <b>for</b>                   | <b>while</b>         | <b>do</b>             |
|------------------------------|----------------------|-----------------------|
| <b>for</b> (n=1; n<=10; ++n) | n = 1;               | n = 1;                |
| {                            | <b>while</b> (n<=10) | <b>do</b>             |
| _____                        | {                    | {                     |
| _____                        | _____                | _____                 |
| {                            | _____                | _____                 |
|                              | n = n+1;             | n = n+1;              |
|                              | }                    | }                     |
|                              |                      | <b>while</b> (n<=10); |

**Example 3.15** The program in Fig. 3.22 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. 3.22** Program to print 'Power of 2' table using for loop

The program evaluates the value

$$p = 2^n$$



### 3.36 Computer Programming

successively by multiplying 2 by itself  $n$  times.

$$q = 2^{-n} = \frac{1}{p}$$

Note that we have declared **p** as a *long int* and **q** as a *double*.

---

**Example 3.16** The program in Fig. 3.23 shows how to write a C program to print  $n$ th Fibonacci number.

---

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

void main()
{
 int num1=0, num2=1, n, i, fib;
 clrscr();

 printf("\n\nEnter the value of n: ");
 scanf ("%d", &n);

 for (i = 1; i <= n-2; i++)
 {
 fib=num1 + num2;
 num1=num2;
 num2=fib;
 }
 printf("\nnth fibonacci number (for n = %d) = %d, n,fib);
 getch();
}
```

**Fig. 3.23** Program to print  $n$ th fibonacci number

---

**Example 3.17** The program in Fig. 3.24 shows how to write a C program to print all the prime numbers between 1 and  $n$ , where ' $n$ ' is the value supplied by the user.

---

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

void main()
{
 int prime (int num):
 int n,i;
 int temp:
```



```

printf("Enter the value of n: ");
scanf ("%d", &n);

printf("Prime numbers between 1 and %d are:\n".n);
for (i=2; j<=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

```

**Fig. 3.24** Program to print all prime numbers between 1 and n

### 3.9.2 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 initialized at a time in the **for** statement. The statements

```

p = 1;
for (n=0; n<17; ++n)

```

can be rewritten as,

```

for (p=1, n=0; n<17; ++n)

```

Note that the initialization section has two parts **p = 1** and **n = 1** separated by a *comma*.

Like the initialization section, the increment section may also have more than one part. For example, the loop



### 3.38 Computer Programming

```
for (n=1, m=50; n<=m; n=n+1, m=m-1)
{
 p = m/n;
 printf("%d %d %d\n", n, m, p);
}
```

is perfectly valid. The multiple arguments in the increment section are separated by *commas*.

The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example below:

```
sum = 0;
for (i = 1; i < 20 && sum < 100; ++i)
{
 sum = sum+i;
 printf("%d %d\n", i, sum);
}
```

The loop uses a compound test condition with the counter variable **i** and sentinel variable **sum**. 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 initialization and increment sections. For example, a statement of the type

```
for (x = (m+n)/2; x > 0; x = x/2)
```

is perfectly valid.

Another unique aspect of **for** loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```

m = 5;
for (; m != 100 ;)
{
 printf("%d\n", m);
 m = m+5;
}

```

Both the initialization and increment sections are omitted in the **for** statement. The initialization 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 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. This can also be written as following:

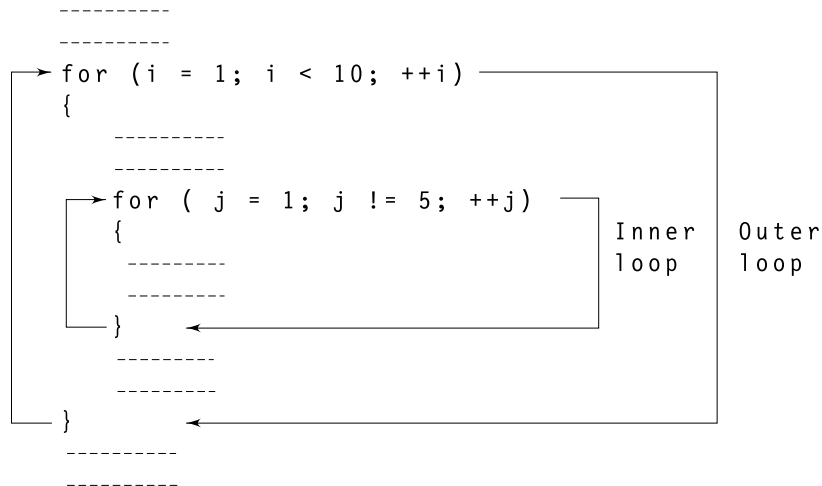
```
for (j=1000; j > 0; j = j-1)
```

This implies that the C compiler will not give an error message if we place a semicolon by mistake at the end of a **for** statement. The semicolon will be considered as a *null statement* and the program may produce some nonsense.



### 3.9.3 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 follows:



The nesting may continue up to any desired level. The loops should be properly indented so as 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).

The program to print the multiplication table discussed in Program 6.2 can be written more concisely using nested for statements as follows:

```

for (row = 1; row <= ROWMAX ; ++row)
{
 for (column = 1; column <= COLMAX ; ++column)
 {
 y = row * column;
 printf("%4d", y);
 }
 printf("\n");
}

```

The outer loop controls the rows while the inner loop controls the columns.

---

**Example 3.18** A class of  $n$  students take an annual examination in  $m$  subjects. A program to read the marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them is given in Fig. 3.25.

---

The program uses two **for** loops, one for controlling the number of students and the other for controlling the number of subjects. Since both the number of students and the number of subjects are requested by the program, the program may be used for a class of any size and any number of subjects.



### 3.40 Computer Programming

The outer loop includes three parts which are as follows:

1. reading of roll-numbers of students, one after another;
2. inner loop, where the marks are read and totalled for each student; and
3. printing of total marks and declaration of grades.

Program

```
#define FIRST 360
#define SECOND 240
main()
{
 int n, m, i, j,
 roll_number, marks, total;
 printf("Enter number of students and subjects\n");
 scanf("%d %d", &n, &m);
 printf("\n");
 for (i = 1; i <= n ; ++i)
 {
 printf("Enter roll_number : ");
 scanf("%d", &roll_number);
 total = 0 ;
 printf("\nEnter marks of %d subjects for ROLL NO %d\n",
 m, roll_number);
 for (j = 1; j <= m; j++)
 {
 scanf("%d", &marks);
 total = total + marks;
 }
 printf("TOTAL MARKS = %d ", total);
 if (total >= FIRST)
 printf("(First Division)\n\n");
 else if (total >= SECOND)
 printf("(Second Division)\n\n");
 else
 printf("(*** F A I L ***)\n\n");
 }
}
```

Output

```
Enter number of students and subjects
3 6
Enter roll_number : 8701
Enter marks of 6 subjects for ROLL NO 8701
81 75 83 45 61 59
TOTAL MARKS = 404 (First Division)
Enter roll_number : 8702
Enter marks of 6 subjects for ROLL NO 8702
```



```

51 49 55 47 65 41
TOTAL MARKS = 308 (Second Division)
Enter roll_number : 8704
Enter marks of 6 subjects for ROLL NO 8704
40 19 31 47 39 25
TOTAL MARKS = 201 (*** F A I L ***)

```

**Fig. 3.25** Illustration of nested for loops

---

**Example 3.19** The program in Fig. 3.26 shows how to write a program to display a pyramid.

---

Program

```

#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);
 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

```

Enter a number for
generating the pyramid:
7

```

```

 0
 1 0 1
 2 1 0 1 2
 3 2 1 0 1 2 3
 4 3 2 1 0 1 2 3 4
5 4 3 2 1 0 1 2 3 4 5
6 5 4 3 2 1 0 1 2 3 4 5 6
7 6 5 4 3 2 1 0 1 2 3 4 5 6 7

```

**Fig. 3.26** Program to build a pyramid



**Example 3.20** Write a program that displays the following pattern using for loop. The number of rows should be entered by the user.

```

 1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1

```

**Program**

```

#include <stdio.h>
int main()
{
 int rows, coef = 1, space, i, j;
 clrscr();
 printf("Enter number of rows: ");
 scanf("%d",&rows);
 for(i=0; i<rows; i++)
 /*for loop executes until condition 'i<rows' inside for loop becomes false*/
 {
 for(space=1; space <= rows-i; space++)
 printf(" ");
 for(j=0; j <= i; j++)
 {
 if (j==0 || i==0)
 coef = 1;
 else
 coef = coef*(i-j+1)/j;
 printf("%4d", coef);
 }
 printf("\n");
 }
 getch();
 return 0;
}

```

**Output**

```

Enter number of rows: 6

 1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1

```

**Fig. 3.27** Program to display pattern



### Selecting a Loop

Given a problem, the programmer's first concern is to decide the type of loop structure to be used. To choose one of the three loop supported by C, we may use the following strategy:

- Analyse the problem and see whether it required a pre-test or post-test loop.
- If it requires a post-test loop, then we can use only one loop, **do while**.
- If it requires a pre-test loop, then we have two choices: **for** and **while**.
- Decide whether the loop termination requires counter-based control or sentinel-based control.
- Use **for** loop if the counter-based control is necessary.
- Use **while** loop if the sentinel-based control is required.
- Note that both the counter-controlled and sentinel-controlled loops can be implemented by all the three control structures.

## 3.10 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.

### 3.10.1 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. They are illustrated in Figs 3.28 and 3.29.

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.

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.



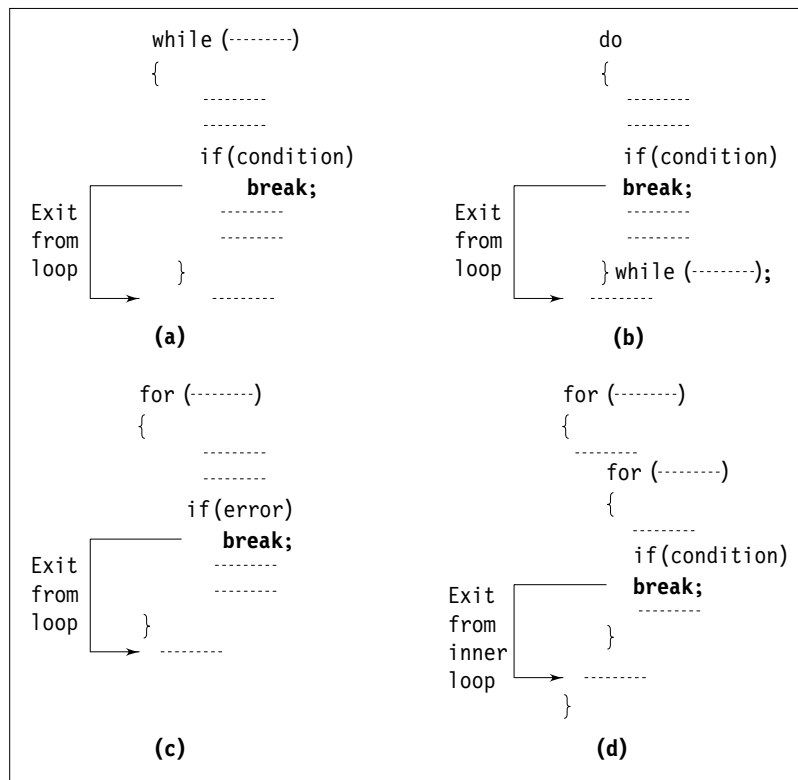


Fig. 3.28 Exiting a loop with break statement

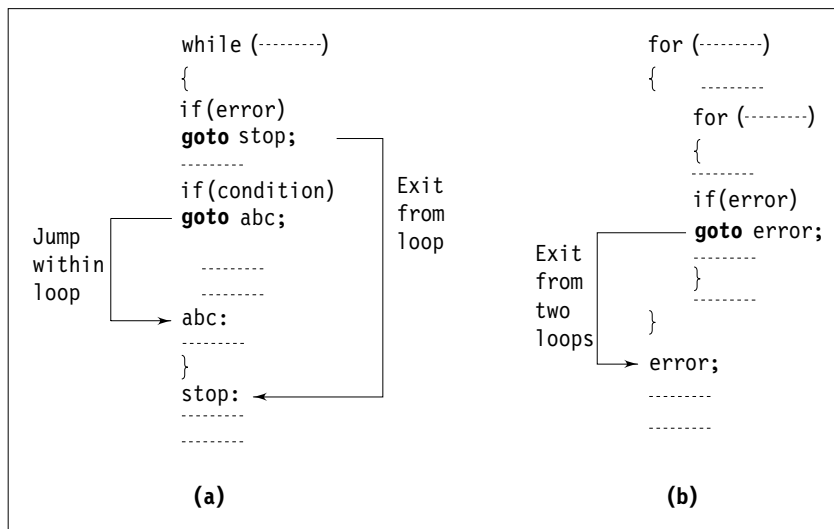


Fig. 3.29 Jumping within and exiting from the loops with goto statement



**Example 3.21** The program in Fig. 3.30 illustrates the use of the break statement in a C program.

The program reads 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.

```

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. 3.30** Use of break in a program

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.

## 3.11 CASE STUDIES

### 1. Range of Numbers

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:



### 3.46 Computer Programming

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.

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

Range = highest value – lowest value

It is therefore necessary to find the highest and the lowest values in the series.

A program to determine the range of values and the average cost of a personal computer in the market is given in Fig. 3.31.

```
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. 3.31** 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 statement

```
if (value < 0) goto output;
```

**Note** that this program can be written without using **goto** statements. Try.

## 2. Pay-Bill Calculations

A manufacturing company has classified its executives into four levels for the benefit of certain perks. The levels and corresponding perks are shown as follows:

| 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.

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.



### 3.48 Computer Programming

**Program:** A program and the results of the test data are given in Fig. 3.32. 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. 3.32** Pay-bill calculations

### 3. Table of Binomial Coefficients

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  $x$ .



### 3.50 Computer Programming

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. 3.33 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. 3.33 Program to print binomial coefficient table

#### 4. Histogram

In an organization, the employees are grouped according to their basic pay for the purpose of certain perks. The pay-range and the number of employees in each group are as follows:

| Group | Pay-Range   | Number of Employees |
|-------|-------------|---------------------|
| 1     | 750 – 1500  | 12                  |
| 2     | 1501 – 3000 | 23                  |
| 3     | 3001 – 4500 | 35                  |
| 4     | 4501 – 6000 | 20                  |
| 5     | above 6000  | 11                  |

Draw a histogram to highlight the group sizes.

**Problem Analysis:** Given the size of groups, it is required to draw bars representing the sizes of various groups. For each bar, its group number and size are to be written.

Program in Fig. 3.34 reads the number of employees belonging to each group and draws a histogram. The program uses four **for** loops and two **if.....else** statements.

```

Program
#define N 5
main()
{
 int value[N];
 int i, j, n, x;
 for (n=0; n < N; ++n)
 {
 printf("Enter employees in Group - %d : ",n+1);
 scanf("%d", &x);
 value[n] = x;
 printf("%d\n", value[n]);
 }
 printf("\n");
 printf("| \n");
 for (n = 0 ; n < N ; ++n)
 {
 for (i = 1 ; i <= 3 ; i++)
 {
 if (i == 2)
 printf("Group-%1d |",n+1);
 else
 printf("|");
 }
 }
}

```



### 3.52 Computer Programming

```
 for (j = 1 ; j <= value[n]; ++j)
 printf("*");
 if (i == 2)
 printf("(%d\n", value[n]);
 else
 printf("\n");
 }
 printf("| \n");
}
```

Output

```
Enter employees in Group - 1 : 12
12
Enter employees in Group - 2 : 23
23
Enter employees in Group - 3 : 35
35
Enter employees in Group - 4 : 20
20
Enter Employees in Group - 5 : 11
11
```

```
Group-1 |
 |*****
 |***** (12)
 |*****
 |
Group-2 |*****
 |***** (23)
 |*****
 |
Group-3 |*****
 |***** (35)
 |*****
 |
Group-4 |*****
 |***** (20)
 |*****
 |
Group-5 |*****
 |***** (11)
 |*****
 |
```

**Fig. 3.34** Program to draw a histogram



## 5. Minimum Cost

The cost of operation of a unit consists of two components  $C_1$  and  $C_2$  which can be expressed as functions of a parameter  $p$  as follows:

$$C_1 = 30 - 8p$$

$$C_2 = 10 + p^2$$

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.

$$\text{Total cost} = C_1 + C_2 = 40 - 8p + p^2$$

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. 3.35 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 A p = 4.0
```

**Fig. 3.35** Program of minimum cost problem



## 6. Plotting of Two Functions

We have two functions of the type

$$y1 = \exp(-ax)$$

$$y2 = \exp(-ax^2/2)$$

Plot the graphs of these functions for x varying from 0 to 5.0.

Initially when  $x = 0$ ,  $y1 = y2 = 1$  and the graphs start from the same point. The curves cross when they are again equal at  $x = 2.0$ . The program should have appropriate branch statements to print the graph points at the following three conditions:

1.  $y1 > y2$
2.  $y1 < y2$
3.  $y1 = y2$

The functions  $y1$  and  $y2$  are normalized and converted to integers as follows:

$$y1 = 50 \exp(-ax) + 0.5$$

$$y2 = 50 \exp(-ax^2/2) + 0.5$$

The program in Fig. 3.36 plots these two functions simultaneously. ( 0 for  $y1$ , \* for  $y2$ , and # for the common point).

```

Program
#include <math.h>
main()
{
 int i;
 float a, x, y1, y2;
 a = 0.4;
 printf(" Y ----> \n");
 printf(" 0 ----- \n");
 for (x = 0; x < 5; x = x+0.25)
 { /* BEGINNING OF FOR LOOP */
 /*.....Evaluation of functions*/
 y1 = (int) (50 * exp(-a * x) + 0.5);
 y2 = (int) (50 * exp(-a * x * x/2) + 0.5);
 /*.....Plotting when y1 = y2.....*/
 if (y1 == y2)
 {
 if (x == 2.5)
 printf(" X |");
 }
 }
}

```



```

 else
 printf("|");

 for (i = 1; i <= y1 - 1; ++i)
 printf(" ");
 printf("#\n");
 continue;
 }
/*..... Plotting when y1 > y2*/
if (y1 > y2)
{
 if (x == 2.5)
 printf(" X |");
 else
 printf(" |");
 for (i = 1; i <= y2 - 1 ; ++i)
 printf(" ");
 printf("*");
 for (i = 1; i <= (y1 - y2 - 1); ++i)
 printf("-");
 printf("0\n");
 continue;
}
/*..... Plotting when y2 > y1.....*/
if (x == 2.5)
 printf(" X |");
else
 printf(" |");
for (i = 1 ; i <= (y1 - 1); ++i)
 printf(" ");
printf("0");
for (i = 1; i <= (y2 - y1 - 1); ++i)
 printf("-");
printf("*\n");
} /*.....END OF FOR LOOP.....*/
printf(" |n");
}

```



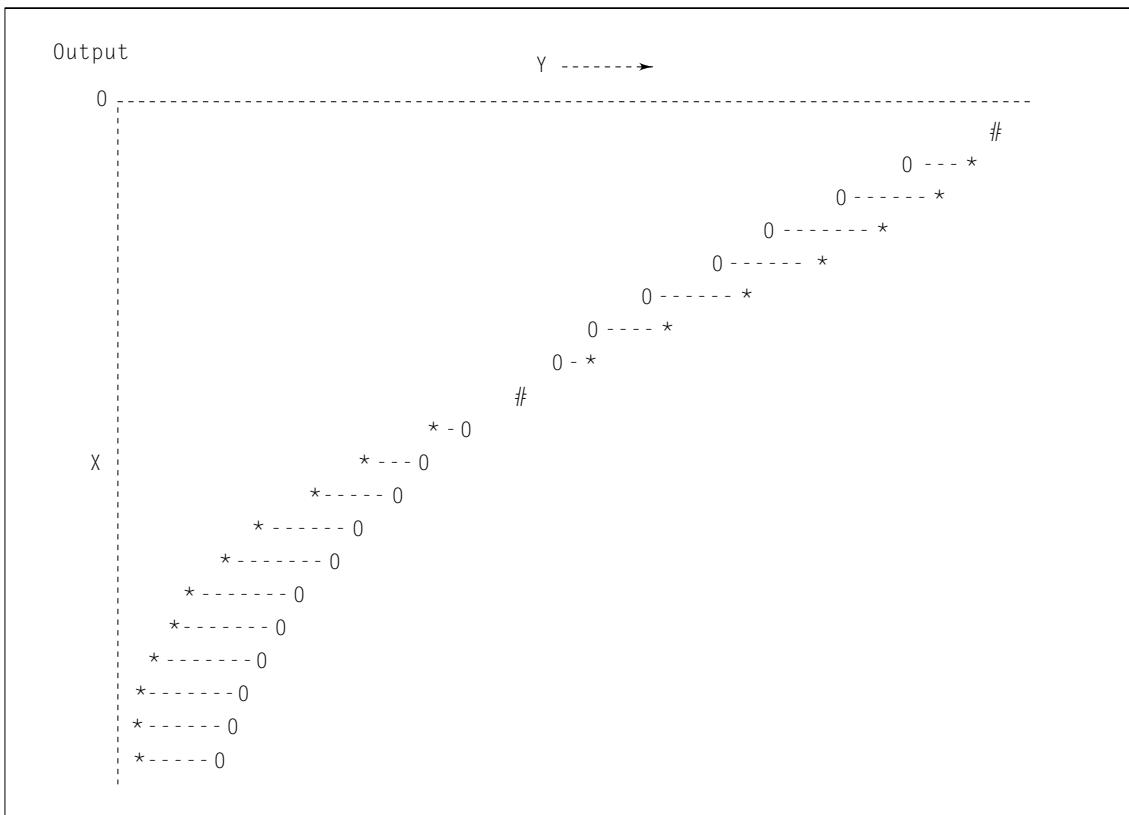


Fig. 3.36 Plotting of two functions



## Key Terms

- **Decision-making statements:** Are the statements that control the flow of execution in a program.
- **Switch statement:** Is a multi-way decision making statement that chooses the statement block to be executed by matching the given value with a list of case values.
- **Conditional operator:** Is a two-way decision making statement that returns one of the two expression values based on the result of the conditional expression.
- **Goto statement:** Is used for unconditional branching. It transfers the flow of control to the place where matching label is found.
- **Infinite loop:** Is a condition where a set of instructions is repeatedly executed.
- **Control statement:** Tests certain conditions and directs the repeated execution of the body of the loop.
- **Program loop:** Executes a sequence of statements repeatedly until some conditions for termination of the loop are satisfied.
- **While statement:** Is an entry-controlled loop that evaluates the test-condition first and then executes the body of the loop if the condition is true.
- **Do statement:** Executes the body of the loop first and then evaluates the test-condition in the subsequent while statement.



- **Break statement:** Terminates the loop and takes the program control to the statement immediately following the loop.

- **Continue statement:** Skips the remaining part of the loop and takes the program control to the next loop iteration.



## Just Remember

1. Be aware of any side effects in the control expression such as `if(x++)`.
2. Check the use of `=operator` in place of the equal operator `=`.
3. Do not give any spaces between the two symbols of relational operators `=`, `!=`, `>=` and `<=`.
4. Writing `!=`, `>=` and `<=` operators like `!=`, `=>` and `=<` is an error.
5. Remember to use two ampersands (`&&`) and two bars (`||`) for logical operators. Use of single operators will result in logical errors.
6. Do not forget to place parentheses for the if expression.
7. It is an error to place a semicolon after the if expression.
8. Do not use the equal operator to compare two floating-point values. They are seldom exactly equal.
9. Avoid using operands that have side effects in a logical binary expression such as `(x-- && ++y)`. The second operand may not be evaluated at all.
10. Be aware of dangling **else** statements.
11. Use braces to encapsulate the statements in **if** and **else** clauses of an `if... else` statement.
12. Do not forget to use a **break** statement when the cases in a switch statement are exclusive.
13. Although it is optional, it is a good programming practice to use the default clause in a switch statement.
14. It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.)
15. Do not use the same constant in two case labels in a switch statement.
16. Try to use simple logical expressions.
17. Be careful while placing a `goto` label in a program as it may lead to an infinite loop condition.
18. It is a common error to use wrong relational operator in test expressions. Ensure that the loop is evaluated exactly the required number of times.
19. Avoid a common error using `=` in place of `=` operator.
20. Do not compare floating-point values for equality.
21. When performing an operation on a variable repeatedly in the body of a loop, make sure that the variable is initialized properly before entering the loop.
22. Indent the statements in the body of loops properly to enhance readability and understandability.
23. Use of blank spaces before and after the loops and terminating remarks are highly recommended.
24. Do not forget to place the semicolon at the end of **do ....while** statement.
25. Do not forget to place the *increment* statement in the body of a **while** or **do... while** loop.
26. Avoid using **while** and **for** statements for implementing exit-controlled (post-test) loops. Use **do...while** statement. Similarly, do not use **do...while** for pre-test loops.
27. Placing a semicolon after the control expression in a **while** or **for** statement is not a syntax error but it is most likely a logic error.
28. Using commas rather than semicolon in the header of a **for** statement is an error.



29. Do not change the control variable in both the **for** statement and the body of the loop. It is a logic error.
30. Although it is legally allowed to place the initialization, testing, and increment sections outside the header of a **for** statement, avoid them as far as possible.
31. Although it is permissible to use arithmetic expressions in initialization and increment section, be aware of round off and truncation errors during their evaluation.
32. Although statements preceding a **for** and statements in the body can be placed in the **for** header, avoid doing so as it makes the program more difficult to read.
33. The use of **break** and **continue** statements in any of the loops is considered unstructured programming. Try to eliminate the use of these jump statements, as far as possible.
34. Avoid the use of **goto** anywhere in the program.
35. Use the function **exit()** only when breaking out of a program is necessary.



### Multiple Choice Questions

1. What will the output of the following program be?  

```
#include <stdio.h>
int main()
{
 int a= 4;
 float b= 4.0;
 if(a==b)
 printf("a and b are equal");
 else
 printf("a and b are not equal");
}
```

  - (a) a and b are equal
  - (b) a and b are not equal
  - (c) a and b are same
  - (d) error
2. What will the output of the following code be?  

```
#include <stdio.h>
int main()
{
 Int a=50, b=100;
 If(a==b)
 Printf("%d,%d",a,b);
 return 0;
}
```

  - (a) 50,100
  - (b) Garbage value
  - (c) Raise an error
  - (d) Will print nothing
3. Which of the following is an invalid if-else statement?  
  - (a) if (if (a == 1)){}
    - (b) if(func1(a)){}
    - (c) if(a){}
    - (d) if((char)a){}
4. What will the output of the following code be?  

```
#include <stdio.h>
int main()
{
 int x = 1;
 if (x)
 printf("Hello C ");
 printf("It is fun\n");
 else
 printf("Hi!\n");
}
```

  - (a) Hello C It is fun
  - (b) It is fun Hi!
  - (c) Hi!
  - (d) Will give compile time error during compilation
5. Which of the following data types can a controlling statement of a SWITCH statement not be?  
  - (a) int
  - (b) float
  - (c) char
  - (d) short
6. Identify the type with which CONTINUE statement cannot be used?  
  - (a) do
  - (b) for
  - (c) while
  - (d) switch



7. What will be the output of the following code?

```
User enters 10 as input.
#include <stdio.h>
void main()
{
 int ch;
 printf("enter a value btw
10to 20:");
 scanf("%d", &ch);
 switch (ch, ch + 1)
 {
 case 1:
 printf("1\n");
 break;
 case 2:
 printf("2");
 break;
 }
}
```

- (a) 10                      (b) 11  
(c) 12                      (d) Run time error
8. Which of the statements hold true in case of SWITCH statements?
- (a) Default statement should mandatorily be present in all SWITCH case programs.  
(b) Default statement gets executed when the value of the expression matches with the first case value.  
(c) Default statement gets executed when the value of the expression matches with the last case value.  
(d) Default statement gets executed if the value of the expression does not match with any of the case values.
9. How many operands does the conditional operator (?:) takes?
- (a) One                      (b) Two  
(c) Three                      (d) Four
10. Identify the correct general form of the conditional operator.
- (a) Conditional expression ? expression1 : expression2  
(b) Conditional expression : expression1 ? expression2  
(c) expression1 : expression2 ? Conditional expression

- (d) expression1 ? expression2 : Conditional expression

11. What will the output of the following code be?

```
#include <stdio.h>
int main()
{
 printf("%d ", 28);
 goto l1;
 printf("%d ", 29);
 l1:goto l2;
 printf("%d ", 30);
 l2:printf("%d ", 31);
}
```

- (a) 28 29 30  
(b) 28 29 31  
(c) 28 31  
(d) Compilation error
12. What will the output of the following code be?
- ```
#include <stdio.h>
void main()
{
    int x;
    for(x=1; x<=10; printf("%d"x));
    x++;
}
```
- (a) Compile time error
(b) Garbage value
(c) 1 to 10
(d) Infinite loop
13. How many times will the given loop run?
- ```
for(i=0; i=5; i++)
```
- (a) Five times  
(b) Six times  
(c) It will run infinitely  
(d) It will not run at all
14. Identify the correct statement.
- (a) A do while statement is useful when we want that the statement within the loop must be executed at least once.  
(b) A do while statement is useful when we want that the statement within the loop must be executed at only once.  
(c) A do while statement is useful when we want that the statement within the loop must be executed more than once.



### 3.60 Computer Programming

- (d) A do while statement is useful when we want that the statement within the loop must not be executed at all.
15. Identify the correct format of a do while statement.
- (a) do  
{  
    Condition  
}  
while (test condition);
- (b) do  
{  
    Body of the loop  
}  
while (test condition);
- (c) do  
{  
    Body of the loop  
}  
while (test condition)
- (d) do  
{  
    while (test condition);  
}
16. What will the output for the following piece of code be?
- ```
#include <stdio.h>
void main()
{
    int x = 0;
    do
    {
        printf("Hi there!");
    } while (x != 0);
```
- (a) Hi there!
(b) Hi there! Gets printed infinite no. of times
(c) Run time error
(d) Nothing gets printed
17. What will the output of the following piece of code be?
- ```
#include <stdio.h>
void main()
{
 int j= 0;
 while (++j)
 {
 printf("Hey");
 }
}
```
- (a) Compile time error  
(b) Nothing gets printed  
(c) Hey is printed infinite times  
(d) Varies
18. Goto function is used to jump from main to within a function.
- (a) True  
(b) False  
(c) Varies from code to code  
(d) Depends on user input
19. Which of the following has the capability to initialize more than one variable at a time?
- (a) do                      (b) while  
(c) for                     (d) if
20. Which of the following statements is capable of skipping the rest of a loop and then resume from the start of the loop?
- (a) break                    (b) resume  
(c) skip                     (d) continue
21. Which of the following functions should be used to break out of a program and return to the operating system?
- (a) break                    (b) goto  
(c) exit()                   (d) skip

### Answers

- |         |         |         |         |         |
|---------|---------|---------|---------|---------|
| 1. (a)  | 2. (d)  | 3. (a)  | 4. (d)  | 5. (b)  |
| 6. (d)  | 7. (b)  | 8. (d)  | 9. (c)  | 10. (a) |
| 11. (c) | 12. (d) | 13. (d) | 14. (a) | 15. (b) |
| 16. (a) | 17. (c) | 18. (b) | 19. (c) | 20. (d) |
| 21. (c) |         |         |         |         |





## Review Questions

1. State whether the following are *true* or *false*:
  - (a) A **switch** expression can be of any type.
  - (b) A program stops its execution when a **break** statement is encountered.
  - (c) Each case label can have only one statement.
  - (d) The **default** case is required in the **switch** statement.
  - (e) When **if** statements are nested, the last **else** gets associated with the nearest **if** without an **else**.
  - (f) One **if** can have more than one **else** clause.
  - (g) Each expression in the **else if** must test the same variable.
  - (h) A **switch** statement can always be replaced by a series of **if..else** statements.
  - (i) Any expression can be used for the **if** expression.
  - (j) The predicate  $!(x \geq 10)(y == 5)$  is equivalent to  $(x < 10) \&\& (y != 5)$ .
  - (k) In a pretest loop, if the body is executed **n** times, the test expression is executed **n + 1** times.
  - (l) The number of times a control variable is updated always equals the number of loop iterations.
  - (m) The **do...while** statement first executes the loop body and then evaluate the loop control expression.
  - (n) An exit-controlled loop is executed a minimum of one time.
  - (o) The three loop expressions used in a **for** loop header must be separated by commas.
  - (p) **while** loops can be used to replace **for** loops without any change in the body of the loop.
  - (q) Both the pretest loops include initialization within the statement.
  - (r) In a **for** loop expression, the starting value of the control variable must be less than its ending value.
  - (s) The initialization, test condition and increment parts may be missing in a **for** statement.
  - (t) The use of **continue** statement is considered as unstructured programming.
2. Fill in the blanks in the following statements.
  - (a) The \_\_\_\_\_ operator is true only when both the operands are true.
  - (b) Multiway selection can be accomplished using an **else if** statement or the \_\_\_\_\_ statement.
  - (c) The \_\_\_\_\_ statement when executed in a **switch** statement causes immediate exit from the structure.
  - (d) The expression  $!(x != y)$  can be replaced by the expression \_\_\_\_\_.
  - (e) The ternary conditional expression using the operator **?:** could be easily coded using \_\_\_\_\_ statement.
  - (f) The sentinel-controlled loop is also known as \_\_\_\_\_ loop.
  - (g) In a counter-controlled loop, variable known as \_\_\_\_\_ is used to count the loop operations.
  - (h) In an exit-controlled loop, if the body is executed **n** times, the test condition is evaluated \_\_\_\_\_ times.
  - (i) A **for** loop with the no test condition is known as \_\_\_\_\_ loop.
  - (j) The \_\_\_\_\_ statement is used to skip a part of the statements in a loop.
3. The following is a segment of a program:
 

```

x = 1;
y = 1;
if (n > 0)
 x = x + 1;
 y = y - 1;
printf(" %d %d", x, y);

```



### 3.62 Computer Programming

What will be the values of x and y if n assumes a value of (a) 1 and (b) 0.

4. Rewrite each of the following without using compound relations:

- (a) `if (grade <= 59 && grade >= 50)`  
    `second = second + 1;`
- (b) `if (number > 100 || number < 0)`  
    `printf(" Out of range");`  
    `else`  
        `sum = sum + number;`
- (c) `if ((M1 > 60 && M2 > 60) || T > 200)`  
    `printf(" Admitted\n");`  
    `else`  
        `printf(" Not admitted\n");`

5. Assuming x = 10, state whether the following logical expressions are true or false.

- (a) `x == 10 && x > 10 && !x`
- (b) `x == 10 || x > 10 && !x`
- (c) `x == 10 && x > 10 || !x`
- (d) `x == 10 || x > 10 || !x`

6. Find errors, if any, in the following switch related statements. Assume that the variables x and y are of int type and x = 1 and y = 2

- (a) `switch (y);`
- (b) `case 10;`
- (c) `switch (x + y)`
- (d) `switch (x) {case 2: y = x + y; break};`

7. Simplify the following compound logical expressions

- (a) `!(x <= 10)`
- (b) `!(x == 10) || ! ( (y == 5) || (z < 0) )`
- (c) `! ( (x + y == z) && !(z > 5) )`
- (d) `!( (x <= 5) && (y == 10) && (z < 5) )`

8. Assuming that x = 5, y = 0, and z = 1 initially, what will be their values after executing the following code segments?

- (a) `if (x && y)`  
    `x = 10;`  
    `else`  
        `y = 10;`
- (b) `if (x || y || z)`  
    `y = 10;`

- `else`  
        `z = 0;`
- (c) `if (x)`  
    `if (y)`  
        `z = 10;`  
    `else`  
        `z = 0;`
- (d) `if (x == 0 || x && y)`  
    `if (!y)`  
        `z = 0;`  
    `else`  
        `y = 1;`

9. Assuming that x = 2, y = 1 and z = 0 initially, what will be their values after executing the following code segments?

- (a) `switch (x)`  
    {  
        case 2:  
            `x = 1;`  
            `y = x + 1;`  
        case 1:  
            `x = 0;`  
            `break;`  
        default:  
            `x = 1;`  
            `y = 0;`  
    }

- (b) `switch (y)`  
    {  
        case 0:  
            `x = 0;`  
            `y = 0;`  
        case 2:  
            `x = 2;`  
            `z = 2;`  
        default:  
            `x = 1;`  
            `y = 2;`  
    }

10. What is the output of the following program?

```
main ()
{
 int m = 5 ;
 if (m < 3) printf("%d" , m+1) ;
 else if(m < 5) printf("%d", m+2);
 else if(m < 7) printf("%d", m+3);
 else printf("%d", m+4);
}
```



11. What is the output of the following program?

```
main ()
{
 int m = 1;
 if (m==1)
 {
 printf (" Delhi ") ;
 if (m == 2)
 printf("Chennai") ;
 else
 printf("Bangalore") ;
 }
 else;
 printf(" END");
}
```

12. What is the output of the following program?

```
main()
{
 int m ;
 for (m = 1; m<5; m++)
 printf("%d\n", (m%2) ? m : m*2);
}
```

13. What is the output of the following program?

```
main()
{
 int m, n, p ;
 for (m = 0; m < 3; m++)
 for (n = 0; n<3; n++)
 for (p = 0; p < 3;; p++)
 if (m + n + p == 2)
 goto print;

 print :
 printf("%d, %d, %d", m, n, p);
}
```

14. What will be the value of x when the following segment is executed?

```
int x = 10, y = 15;
x = (x<y)? (y+x) : (y-x) ;
```

15. What will be the output when the following segment is executed?

```
int x = 0;
if (x >= 0)
if (x > 0)
 printf("Number is positive");
else
printf("Number is negative");
```

16. What will be the output when the following segment is executed?

```
char ch = 'a' ;
switch (ch)
{
 case 'a' :
 printf("A") ;
 case 'b':
 Printf ("B") ;
 default :
 printf(" C ") ;
}
```

17. What will be the output of the following segment when executed?

```
int x = 10, y = 20;
if((x<y) || (x+5) > 10)
printf("%d", x);
else
printf("%d", y);
```

18. What will be output of the following segment when executed?

```
int a = 10, b = 5;
if (a > b)
{
 if(b > 5)
 printf("%d", b);
}
else
 printf("%d", a);
```

19. Can we change the value of the control variable in **for** statements? If yes, explain its consequences.

20. What is a null statement? Explain a typical use of it.

21. Use of **goto** should be avoided. Explain a typical example where we find the application of **goto** becomes necessary.

22. How would you decide the use of one of the three loops in C for a given problem?

23. How can we use **for** loops when the number of iterations are not known?

24. Explain the operation of each of the following **for** loops.

(a) for ( n = 1; n != 10; n += 2)  
sum = sum + n;



### 3.64 Computer Programming

- (b) 

```
for (n = 5; n <= m; n -=1)
 sum = sum + n;
```
- (c) 

```
for (n = 1; n <= 5;)
 sum = sum + n;
```
- (d) 

```
for (n = 1; ; n = n + 1)
 sum = sum + n;
```
- (e) 

```
for (n = 1; n < 5; n ++)
 n = n -1
```
25. What would be the output of each of the following code segments?
- (a) 

```
count = 5;
while (count -- > 0)
 printf(count);
```
- (b) 

```
count = 5;
while (-- count > 0)
 printf(count);
```
- (c) 

```
count = 5;
do printf(count);
while (count > 0);
```
- (d) 

```
for (m = 10; m > 7, m -=2)
 printf(m);
```
26. Compare, in terms of their functions, the following pairs of statements:
- (a) while and do...while  
 (b) while and for  
 (c) break and continue  
 (d) break and goto  
 (e) continue and goto
27. Analyse each of the program segments that follow and determine how many times the body of each loop will be executed.
- (a) 

```
x = 5;
y = 50;
while (x <= y)
{
 x = y/x;

}
```
- (b) 

```
int m = 10;
int n = 7;
while (m % n >= 0)
{

 m = m + 1;
```
- ```
        n = n + 2;
        ---
    }
```
- (c)

```
m = 1;
do
{
    -----
    m = m+2;
}
while (m < 10);
```
- (d)

```
int i;
for (i = 0; i <= 5; i = i+2/3)
{
    -----
    -----
    -----
}
```
28. Write a **for** statement to print each of the following sequences of integers:
- (a) 1, 2, 4, 8, 16, 32
 (b) 1, 3, 9, 27, 81, 243
 (c) -4, -2, 0, 2, 4
 (d) -10, -12, -14, -18, -26, -42
29. Change the following **for** loops to **while** loops:
- (a)

```
for (m = 1; m < 10; m = m + 1)
    printf(m);
```
- (b)

```
for ( ; scanf("%d", & m) != -1;)
    printf(m);
```
30. Change the **for** loops in Exercise 6.13 to **do** loops.
31. What is the output of following code?
- ```
int m = 100, n = 0;
while (n == 0)
{
 if (m < 10)
 break;
 m = m-10;
```
32. What is the output of the following code?
- ```
int m = 0 ;
do
{
    if (m > 10 )
        continue ;
```



```

        m = m + 10 ;
    } while ( m < 50 ) ;
    printf("%d", m);
33. What is the output of the following code?
    int n = 0, m = 1 ;
    do
    {
        printf(m) ;
        m++ ;

```

```

    }
    while (m <= n) ;
34. What is the output of the following code?
    int n = 0, m ;
    for (m = 1; m <= n + 1 ; m++ )
        printf(m);
35. When do we use the following statement?
    for ( ; ; )

```



Debugging Exercises

1. Find errors, if any, in each of the following segments:

- (a)

```
if (x + y = z && y > 0)
    printf(" ");
```
- (b)

```
if (p < 0) || (q < 0)
    printf ("    sign    is
    negative");
```
- (c)

```
if (code > 1);
    a = b + c
else
    a = 0
```

2. Find the error, if any, in the following statements:

- (a)

```
if ( x > = 10 ) then
    printf ( "\n" ) ;
```
- (b)

```
if x > = 10
    printf ( "OK" ) ;
```
- (c)

```
if (x = 10)
    printf ("Good" ) ;
```
- (d)

```
if (x = < 10)
    printf ("Welcome") ;
```

3. Find errors, if any, in each of the following looping segments. Assume that all the variables have been declared and assigned values.

- (a)

```
while (count != 10);
{
    count = 1;
    sum = sum + x;
    count = count + 1;
}
```
- (b)

```
name = 0;
do { name = name + 1;
    printf("My name is John\n");}
while (name = 1)
```
- (c)

```
do;
total = total + value;
scanf("%f", &value);
while (value != 999);
```
- (d)

```
for (x = 1, x > 10; x = x + 1)
{
    -----
    -----
    -----
}
```
- (e)

```
m = 1;
n = 0;
for ( ; m+n < 10; ++n);
printf("Hello\n");
m = m+10
```
- (f)

```
for (p = 10; p > 0;)
p = p - 1;
printf("%f", p);
```




Programming Exercises

- Write a program to determine whether a given number is 'odd' or 'even' and print the message

NUMBER IS EVEN

or

NUMBER IS ODD

- (a) without using else option, and (b) with else option.
- Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.
- A set of two linear equations with two unknowns x_1 and x_2 is given below:

$$ax_1 + bx_2 = m$$

$$cx_1 + dx_2 = n$$

The set has a unique solution

$$x_1 = \frac{md - bn}{ad - cb}$$

$$x_2 = \frac{na - mc}{ad - cb}$$

provided the denominator $ad - cb$ is not equal to zero.

Write a program that will read the values of constants a , b , c , d , m , and n and compute the values of x_1 and x_2 . An appropriate message should be printed if $ad - cb = 0$.

- Given a list of marks ranging from 0 to 100, write a program to compute and print the

number of students:

- who have obtained more than 80 marks,
- who have obtained more than 60 marks,
- who have obtained more than 40 marks,
- who have obtained 40 or less marks,
- in the range 81 to 100,
- in the range 61 to 80,
- in the range 41 to 60, and
- in the range 0 to 40.

The program should use a minimum number of **if** statements.

- Admission to a professional course is subject to the following conditions:

- Marks in Mathematics ≥ 60
- Marks in Physics ≥ 50
- Marks in Chemistry ≥ 40
- Total in all three subjects ≥ 200

or

Total in Mathematics and Physics ≥ 150

Given the marks in the three subjects, write a program to process the applications to list the eligible candidates.

- Write a program to print a two-dimensional Square Root Table as shown below, to provide the square root of any number from 0 to 9.9. For example, the value x will give the square root of 3.2 and y the square root of 3.9.

SQUARE ROOT TABLE

Number	0.0	0.1	0.2	0.9
0.0					
1.0					
2.0					
3.0			x		y
9.0					

7. Shown below is a Floyd's triangle.

```

1
2 3
4 5 6
7 8 9 10
11 .. ... 15
.
.
79 .. ... .. 91

```

- (a) Write a program to print this triangle.

- (b) Modify the program to produce the following form of Floyd's triangle.

```

1
0 1
1 0 1
0 1 0 1
1 0 1 0 1

```

8. A cloth showroom has announced the following seasonal discounts on purchase of items:

Purchase amount	Discount	
	Mill cloth	Handloom items
0 – 100	–	5%
101 – 200	5%	7.5%
201 – 300	7.5%	10.0%
Above 300	10.0%	15.0%

Write a program using **switch** and **if** statements to compute the net amount to be paid by a customer.

9. Write a program that will read the value of x and evaluate the following function

$$y = \begin{cases} 1 & \text{for } x < 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x > 0 \end{cases}$$

using

- (a) nested if statements,
 (b) else if statements, and
 (c) conditional operator ?:

10. Write a program to compute the real roots of a quadratic equation

$$ax^2 + bx + c = 0$$

The roots are given by the equations

$$x_1 = -b + \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = -b - \frac{\sqrt{b^2 - 4ac}}{2a}$$

The program should request for the values of the constants a , b and c and print the values of x_1 and x_2 . Use the following rules:

- (a) No solution, if both a and b are zero
 (b) There is only one root, if $a = 0$ ($x = -c/b$)
 (c) There are no real roots, if $b^2 - 4ac$ is negative
 (d) Otherwise, there are two real roots

Test your program with appropriate data so that all logical paths are working as per your design. Incorporate appropriate output messages.

11. Write a program to read three integer values from the keyboard and displays the output stating that they are the sides of right-angled triangle.
12. An electricity board charges the following rates for the use of electricity:
 For the first 200 units: 80 P per unit
 For the next 100 units: 90 P per unit
 Beyond 300 units: Rs 1.00 per unit
 All users are charged a minimum of Rs. 100 as meter charge. If the total amount is more

than Rs. 400, then an additional surcharge of 15% of total amount is charged.

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

13. Write a program to compute and display the sum of all integers that are divisible by 6 but not divisible by 4 and lie between 0 and 100. The program should also count and display the number of such values.
14. Write an interactive program that could read a positive integer number and decide whether the number is a prime number and display the output accordingly.
Modify the program to count all the prime numbers that lie between 100 and 200.
NOTE: A prime number is a positive integer that is divisible only by 1 or by itself.
15. Write a program to read a double-type value x that represents angle in radians and a character-type variable T that represents the type of trigonometric function and display the value of
 - (a) $\sin(x)$, if s or S is assigned to T ,
 - (b) $\cos(x)$, if c or C is assigned to T , and
 - (c) $\tan(x)$, if t or T is assigned to T
 using (i) **if.....else** statement, and (ii) **switch** statement.
16. Given a number, write a program using **while** loop to reverse the digits of the number. For example, the number
12345
should be written as
54321
(Hint: Use modulus operator to extract the last digit and the integer division by 10 to get the $n-1$ digit number from the n digit number.)
17. The factorial of an integer m is the product of consecutive integers from 1 to m . That is, factorial $m = m! = m \times (m-1) \times \dots \times 1$.
Write a program that computes and prints a table of factorials for any given m .
18. Write a program to compute the sum of the digits of a given integer number.

19. The numbers in the sequence

1 1 2 3 5 8 13 21

are called Fibonacci numbers. Write a program using a **do....while** loop to calculate and print the first m Fibonacci numbers.

(Hint: After the first two numbers in the series, each number is the sum of the two preceding numbers.)

20. Rewrite the program of the Example 6.1 using the **for** statement.
21. Write a program to evaluate the following investment equation

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

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

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

r : 0.10, 0.11, 0.12,, 0.20

n : 1, 2, 3,, 10

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

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

$$P = V$$

That is, the value of money at the end of first year becomes the principal amount for the next year and so on.)

22. Write programs to print the following outputs using **for** loops.

(a) 1	(b) * * * * *
2 2	* * * *
3 3 3	* * *
4 4 4 4	* *
5 5 5 5 5	*

23. Write a program to read the age of 100 persons and count the number of persons in the age group 50 to 60. Use **for** and **continue** statements.
24. Rewrite the program of case study 6.4 (plotting of two curves) using **else...if** constructs instead of **continue** statements.
25. Write a program to print a table of values of the function
 $y = \exp(-x)$

for x varying from 0.0 to 10.0 in steps of 0.10. The table should appear as follows:

Table for $Y = \text{EXP}(-X)$

x	0.1	0.2	0.3	0.9
0.0					
1.0					
2.0					
3.0					
.					
.					
.					
9.0					

26. Write a program that will read a positive integer and determine and print its binary equivalent.

(Hint: The bits of the binary representation of an integer can be generated by repeatedly dividing the number and the successive quotients by 2 and saving the remainder, which is either 0 or 1, after each division.)

27. Write a program using **for** and **if** statement to display the capital letter S in a grid of 15 rows and 18 columns as shown below.

```

*****
* _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ *
* * * * * _ _ _ _ _ _ _ _ _ _ _ *
* * * *
* * * *
* * * *
* * * * _ _ _ _ _ _ _ _ _ _ _ *
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ *
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ *
* * * *
* * * *
* * * *
* * * * _ _ _ _ _ _ _ _ _ _ _ *
* * _ _ _ _ _ _ _ _ _ _ _ _ _ _ *
* _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ *

```

28. Write a program to compute the value of Euler's number e, that is used as the base

of natural logarithms. Use the following formula.

$$e = 1 + 1/1! + 1/2! + 1/3! + \dots + 1/n!$$

Use a suitable loop construct. The loop must terminate when the difference between two successive values of e is less than 0.00001.

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

(a) $\sin x = x - x^3/3! + x^5/5! - x^7/7! + \dots$

(b) $\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \dots$

(c) $\text{SUM} = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \dots$

...

30. The present value (popularly known as book value) of an item is given by the relationship.

$$P = c(1-d)^n$$

where c = original cost

d = rate of depreciation (per year)

n = number of years

p = present value after y years.

If P is considered the scrap value at the end of useful life of the item, write a program to compute the useful life in years given the original cost, depreciation rate, and the scrap value.

The program should request the user to input the data interactively.

3.70 Computer Programming

31. Write a program to print a square of size 5 by using the character **S** as shown below:

(a) S S S S S
S S S S S
S S S S S
S S S S S
S S S S S

(b) S S S S S
S S S S S
S S S S S
S S S S S
S S S S S

32. Write a program to graph the function $y = \sin(x)$ in the interval 0 to 180 degrees in steps of 15 degrees. Use the concepts discussed in the Case Study 4 in Chapter 6.
33. Write a program to print all integers that are **not divisible** by either 2 or 3 and lie between

1 and 100. Program should also account the number of such integers and print the result.

34. Modify the program of Exercise 6.16 to print the character **O** instead of **S** at the center of the square as shown below.

S S S S S
S S S S S
S S O S S
S S S S S
S S S S S

35. Given a set of 10 two-digit integers containing both positive and negative values, write a program using **for** loop to compute the sum of all positive values and print the sum and the number of values added. The program should use **scanf** to read the values and terminate when the sum exceeds 999. Do not use **goto** statement.

4

User-Defined Functions

CHAPTER OUTLINE

4.1 Introduction	4.6 Definition of Functions	4.12 Passing Strings to Functions
4.2 Need for User-Defined Functions	4.7 Return Values and their Types	4.13 The Scope, Visibility, and Lifetime of Variables
4.3 A Multi-Function Program	4.8 Function Calls	4.14 Multifile Programs
4.4 Category of Functions	4.9 Function Declaration	4.15 Case Studies
4.5 Elements of User-Defined Functions	4.10 Recursion	
	4.11 Passing Arrays To Functions	

4.1 INTRODUCTION

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.

4.2 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**’.

4.2 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. 4.1. 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.

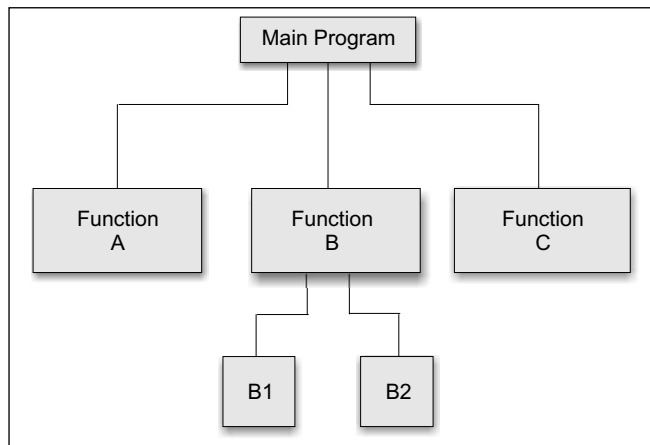


Fig. 4.1 Top-down modular programming using functions

4.3 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 **println**, which could print a line of 39-character length. This function can be used in a program as follows:

```
void println(void); /* declaration */
main( )
{
    println( );
    printf("This illustrates the use of C functions\n");
    println();
}

void println(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

println() function

As we know, the program execution always begins with the **main** function. During execution of the **main**, the first statement encountered is

```
println( );
```

which indicates that the function **println** is to be executed. At this point, the program control is transferred to the function **println**. After executing the **println** 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 **println** function for printing the line once more.

The **main** function calls the user-defined **println** function two times and the library function **printf** once. We may notice that the **println** 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 4.2 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.

4.4 Computer Programming

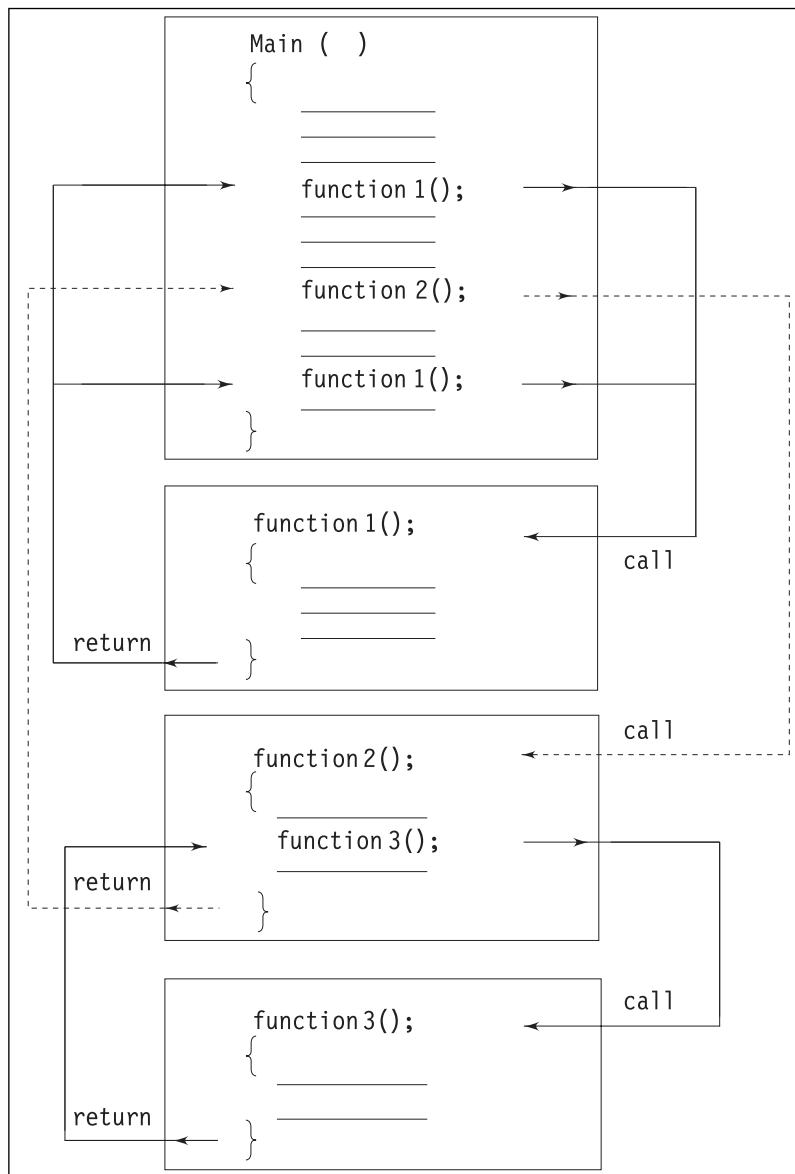


Fig. 4.2 Flow of control in a multi-function program

4.3.1 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 as follows:

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.

4.4 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.

4.4.1 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. 4.3. The dotted lines indicate that there is only a transfer of control but not data.

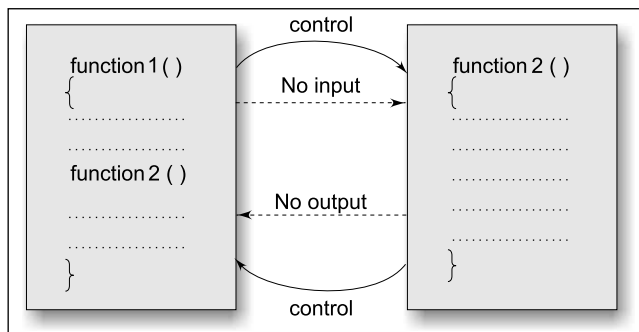


Fig. 4.3 No data communication between functions

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.

Example 4.1 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. 4.4. **main** is the calling function that calls **printline** and **value** functions. Since both the called functions contain no arguments, there are no argument declarations. The **printline** 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 printline (void);
void value (void);
main()
{
    printline();
    value();
    printline();
}
/*      Function1: printline( )          */

void printline(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;
    }
}

```

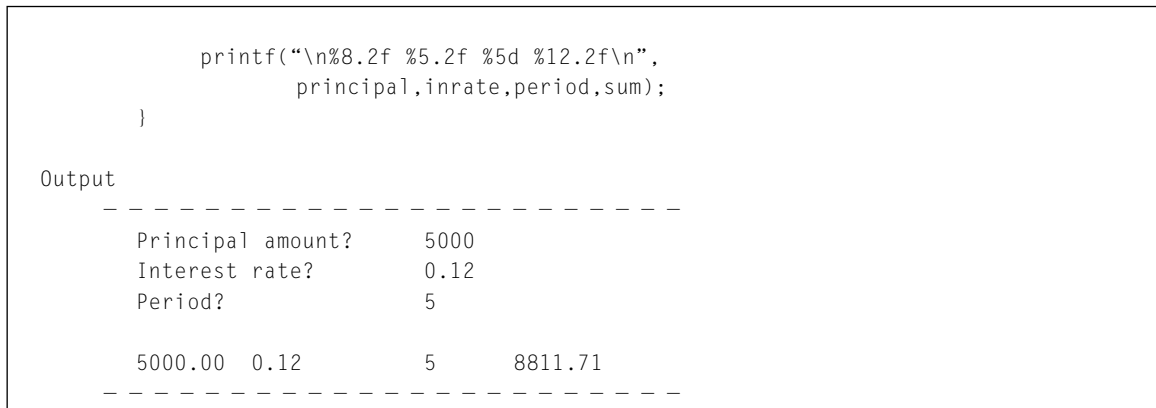



Fig. 4.4 Functions with no arguments and no return values

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 **printline** 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.

4.4.2 Arguments but No Return Values

In Fig. 4.4 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. 4.5.

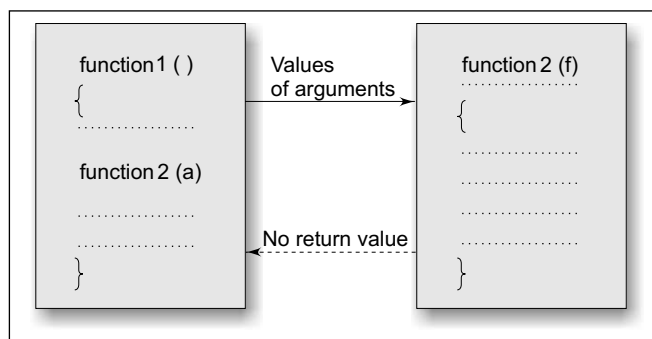


Fig. 4.5 One-way data communication

4.8 Computer Programming

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. 4.6.

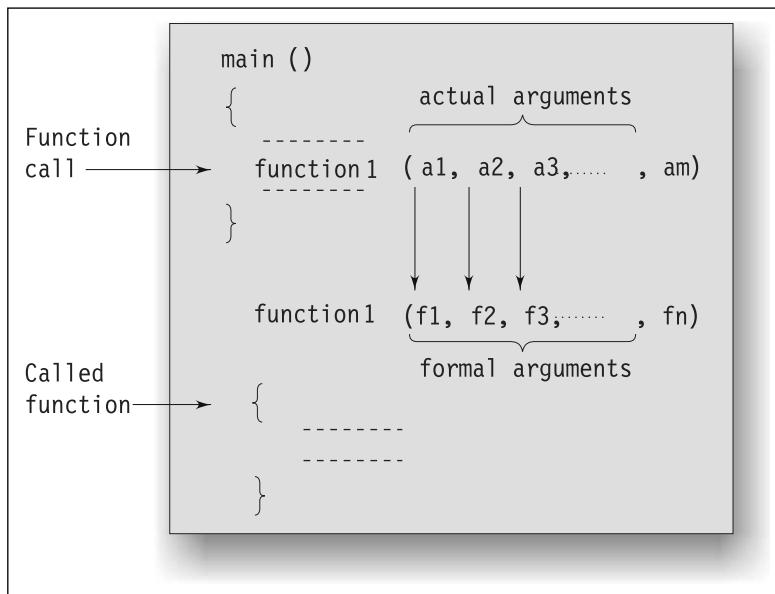


Fig. 4.6 Arguments matching between the function call and the called function

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.

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 4.2 Modify the program of Example 4.1 to include the arguments in the function calls.

The modified program with function arguments is presented in Fig. 4.7. Most of the program is identical to the program in Fig. 4.4. 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;
```


4.10 Computer Programming

```
while(year <= n)
{
    sum = sum * (1+r);
    year = year +1;
}
printf("%f\t%f\t%d\t%f\n",p,r,n,sum);
}
```

Output

[illegible]

Fig. 4.7 *Functions with arguments but no return values*

The variables declared inside a function are known as *local variables* and therefore their values are local to the function and cannot be accessed by any other function. We shall discuss more about this later in the chapter.

The function **value** calculates the final amount for a given period and prints the results as before. Control is transferred back on reaching the closing brace of the function. Note that the function does not return any value.

The function **prntline** is called twice. The first call passes the character 'Z', while the second passes the character 'C' to the function. These are assigned to the formal argument **ch** for printing lines (see the output).

Variable Number of Arguments

Some functions have a variable number of arguments and data types which cannot be known at compile time. The **printf** and **scanf** functions are typical examples. The ANSI standard proposes new symbol called the *ellipsis* to handle such functions. The *ellipsis* consists of three periods (...) and used as shown below:

double area(float d,...)

Both the function declaration and definition should use ellipsis to indicate that the arguments are arbitrary both in number and type.

4.4.3 Arguments with Return Values

The function **value** in Fig. 4.7 receives data from the calling function through arguments, but does not send back any value. Rather, it displays the results of calculations at the terminal. However, we may not always wish to have the result of a function displayed. We may use it in the calling function for further processing. Moreover, to assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations. For example, different programs may require different output formats for display of results. These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like a ‘black box’ that receives a predefined form of input and outputs a desired value. Such functions will have two-way data communication as shown in Fig. 4.8.

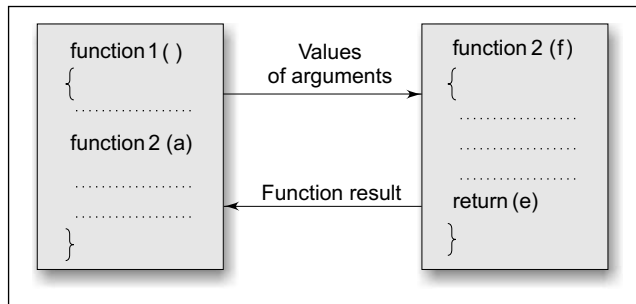


Fig. 4.8 Two-way data communication between functions

We shall modify the program in Fig. 4.7 to illustrate the use of two-way data communication between the *calling* and the *called functions*.

Example 4.3 In the program presented in Fig. 4.7 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. 4.9. 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 */

```


4.12 Computer Programming

```
{
    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. 4.9 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** in Example 4.3 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 4.4 Write a C program to find the area of a rectangle using user defined function.

Program

```
#include <stdio.h>
int findArea(int a, int b);
/*user defined function declaration*/
int main()
{
    int n1,n2,area;
    clrscr();
    printf("Enter length and width: ");
    scanf("%d %d",&n1,&n2);
    area = findArea(n1, n2);
    /*user defined function calling*/
    printf("Area of rectangle = %d",area);
    getch();
    return 0;
}
int findArea(int a,int b)
/*user defined function definition*/
{
    int result;
    result = a*b;
    return result;
}
```

Output

```
Enter length and width: 2 6
Area of rectangle = 12
```

Fig. 4.10 Finding the area of a rectangle using user defined function

Example 4.5 Write a function **power** that computes x raised to the power y for integers x and y and returns double-type value.

Figure 4.11 shows a **power** function that returns a **double**. The prototype declaration

```
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. 4.11 Power functions: Illustration of return of float values

Example 4.6 The program in Fig. 4.12 shows how to write a C program (float x [], int n) that returns the position of the first minimum value among the first n elements of the given array x.

Program

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

Fig. 4.12 Program to return the position of the first minimum value in an array

4.16 Computer Programming

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**.

4.4.4 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);
}
```

4.4.5 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:

```
* 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”. Pointers and their applications are discussed in detail in Chapter 7.

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 *****.

4.4.6 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.

4.18 Computer Programming

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.

4.5 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. 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*.

4.6 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
6. A return statement.

All the six elements are grouped into two parts, namely,

- Function header (First three elements)
- 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;
```


4.20 Computer Programming

```
    . . . . .  
    . . . . .  
    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.

4.6.1 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.

4.6.2 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.

4.6.3 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 the 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.

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.

4.6.4 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.

4.7 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.

4.22 Computer Programming

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: 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.

4.8 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;
```

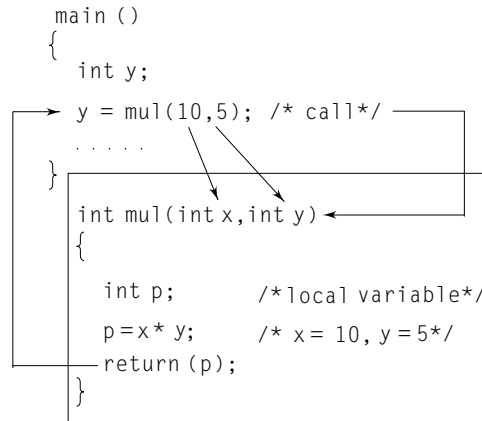


```

    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 4.3 belongs to this category. Such functions may be called in by simply stating their names as independent statements.

4.24 Computer Programming

Example:

```
main( )
{
    printf( );
}
```

Note the presence of a semicolon at the end.

4.8.1 Function Call

A function call is a postfix expression. The operator (.) is at a very high level of precedence. 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.

4.9 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.
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 as follows:

```
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.

4.9.1 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 are 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.

4.9.2 Parameters Everywhere!

Parameters (also known as arguments) are used in following 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.

4.10 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 $4 = 4 \times 3 \times 2 \times 1 = 24$

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.

Example 4.7 Write a C program to find the HCF of two positive numbers using recursion.

Program

```
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
    int n1, n2;
    clrscr();
    printf("Enter two positive integers: ");
    scanf("%d%d", &n1, &n2);
    printf("H.C.F of %d and %d = %d", n1, n2, hcf(n1,n2));
    getch();
    return 0;
}
int hcf(int n1, int n2)
{
    if (n2!=0)
        return hcf(n2, n1%n2);
    /*recursion is a function which calls itself until condition goes false*/
    else
        return n1;
}
```

Output

```
Enter two positive integers: 8    4
H.C.F of 8 and 4 = 4
```

Fig. 4.13 Finding HCF of two positive numbers using recursion

4.10.1 Recursion versus Iteration

Recursion involves creating the function and this function calls itself again and again till the given condition is met. In the complete process, whenever the function calls itself, it pushes information like parameter value and other information onto stack. So the number of times a function calls itself, the information (activation record) gets pushed onto stack. This increases the memory requirement for the recursive program. However, in case of an iterative program, no additional memory is required. Iterative programs are hard to write but are good performance wise. Recursive programs are easy to write but are bad performance wise.

4.11 PASSING ARRAYS TO FUNCTIONS

4.11.1 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)

4.28 Computer Programming

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 4.8 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 as:

$$S.D = \sqrt{\frac{1}{n} \sum_{i=1}^n (\bar{x} - x_i)^2}$$

Where \bar{x} is the mean of the values.

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. 4.14 Passing of arrays to a function

A multifunction program consisting of **main**, **std_dev**, and **mean** functions is shown in Fig. 4.14. **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. Program 4.6 highlights these concepts.

Example 4.9 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. 4.15. 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. 4.15 *Sorting of array elements using a function*

4.11.2 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.
4. 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]=

```


4.32 Computer Programming

```
        {
            {1,2},
            {3,4},
            {5,6}
        };

        mean = average(matrix, M, N);
        . . . . .
        . . . . .
    }
```

4.12 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.

4.12.1 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 following 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.

4.13 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.

4.13.1 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 4.10 Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 4.16. **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.

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. 4.16 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.

4.13.2 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 4.11 Write a multifunction program to illustrate the properties of global variables.

A program to illustrate the properties of global variables is presented in Fig. 4.17. 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. 4.17 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 **func1** will, therefore, assign 1 to **y**.

4.13.3 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 */
```


4.38 Computer Programming

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.

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.

4.13.4 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 4.12 Write a program to illustrate the properties of a static variable.

The program in Fig. 4.18 explains the behaviour 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. 4.18 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
```


4.40 Computer Programming

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**.

4.13.5 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.

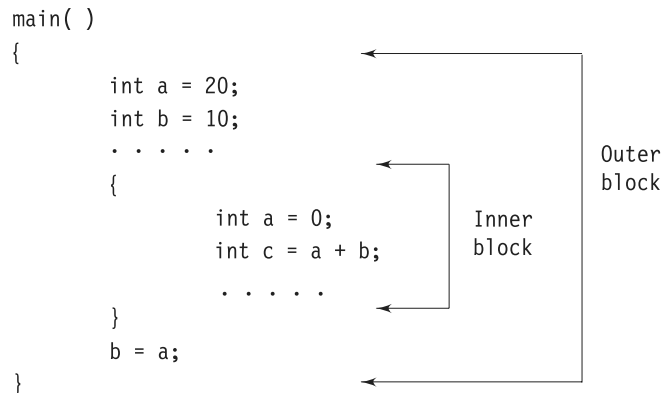
Table 4.1 summarizes the information on the visibility and lifetime of variables in functions and files.

TABLE 4.1 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

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:



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 value 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.

4.14 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 4.19 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.

file1.c	file2.c
<pre> main() { extern int m; int i; } function1() { int j; } </pre>	<pre> int m /* global variable */ function2() { int i; } function3() { int count; } </pre>

Fig. 4.19 Use of **extern** in a multifile program

The multifile program shown in Fig. 4.20 can be modified as shown in Fig. 4.19.

file1.c	file2.c
<pre> int m; /* global variable */ main() </pre>	<pre> extern int m; function2() </pre>


```

{      {
      int i;
      . . . . .
}      }
function1( )      function3( )
{      {
      int j;      int count;
      . . . . .
}      }

```

Fig. 4.20 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**.

4.15 CASE STUDIES

1. 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. 4.21.

The program in Fig. 4.22 calculates the area for a curve of the function

$$f(x) = x^2 + 1$$

between any two given limits, say, A and B.

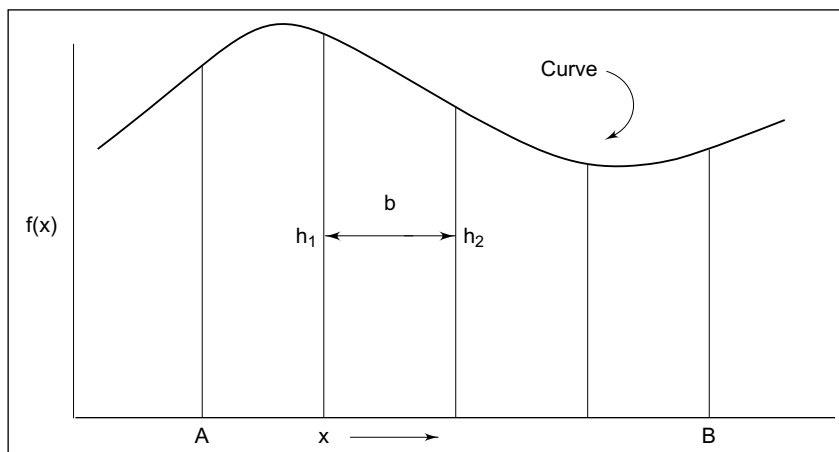


Fig. 4.21 Area under a curve

4.44 Computer Programming

Input

Lower limit (A)
Upper limit (B)
Number of trapezoids

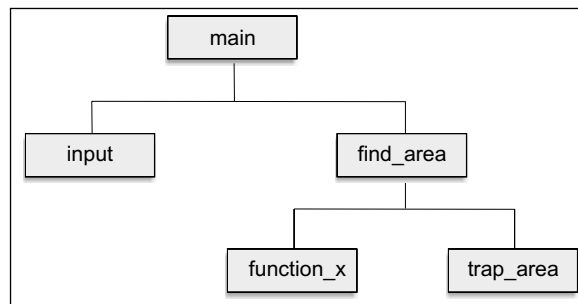
Output

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 shown:



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).

```
Program
#include <stdio.h>
float    start_point,          /* GLOBAL VARIABLES */
        end_point,
        total_area;
int      numtraps;
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. 4.22 Computing area under a curve



Key Terms

- **Arguments:** Are the set of values that are passed to a function to enable the function to perform the desired task.
- **Block statement:** Is a set of statements enclosed within a set of braces.
- **Function:** Is an independently coded subprogram that performs a specific task.
- **Modular Programming:** Is a software development approach that organizes a large program into small, independent program segments called modules.
- **Calling program:** Is the program or function that calls another function.
- **Function body:** Contains the statement block for performing the required task.
- **Function type:** Specifies the type of value that the function will return.
- **Parameter list:** Is a list of variables that will receive data values at the time of function call.
- **Program definition:** Is an independent program module that is written to perform specific task. It is also referred as function definition.
- **Recursion:** Is a scenario where a function calls itself.
- **External variable:** Is a variable that is active throughout the program. It is also referred as global variable.
- **Local variable:** Is a variable that is active only within a specific function or statement block. It is also referred as internal variable.



Just Remember

1. A function that returns a value can be used in expressions like any other C variable.
2. A function that returns a value cannot be used as a stand-alone statement.
3. Where more functions are used, they may be placed in any order.
4. It is a syntax error if the types in the declaration and function definition do not match.
5. It is a syntax error if the number of actual parameters in the function call do not match the number in the declaration statement.
6. It is a logic error if the parameters in the function call are placed in the wrong order.
7. Placing a semicolon at the end of header line is illegal.
8. Forgetting the semicolon at the end of a prototype declaration is an error.
9. A **return** statement can occur anywhere within the body of a function.
10. A function definition may be placed either after or before the **main** function.
11. A **return** statement is required if the return type is anything other than **void**.

12. If a function does not return any value, the return type must be declared **void**.
13. If a function has no parameters, the parameter list must be declared **void**.
14. Using **void** as return type when the function is expected to return a value is an error.
15. Trying to return a value when the function type is marked **void** is an error.
16. Defining a function within the body of another function is not allowed.
17. It is an error if the type of data returned does not match the return type of the function.
18. It will most likely result in logic error if there is a mismatch in data types between the actual and formal arguments.
19. Functions return integer value by default.
20. A function without a return statement cannot return a value, when the parameters are passed by value.
21. When the value returned is assigned to a variable, the value will be converted to the type of the variable receiving it.
22. Function cannot be the target of an assignment.
23. 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.
24. A function can have more than one return statement.
25. A recursive function must have a condition that forces the function to return without making the recursive call; otherwise the function will never return.
26. It is illegal to use the name of a formal argument as the name of a local variable.
27. 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).
28. Use parameter passing by values as far as possible to avoid inadvertent changes to variables of calling function in the called function.
29. Although not essential, include parameter names in the prototype declarations for documentation purposes.
30. A global variable used in a function will retain its value for future use.
31. A local variable defined inside a function is known only to that function. It is destroyed when the function is exited.
32. A global variable is visible only from the point of its declaration to the end of the program.
33. 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.
34. A local variable declared **static** retains its value even after the function is exited.
35. Static variables are initialized at compile time and therefore, they are initialized only once.
36. Avoid the use of names that hide names in outer scope.



Multiple Choice Questions

1. What is a self contained block of code that performs a particular task known as?
 - (a) Pointer
 - (b) Function
 - (c) Loop
 - (d) Array
2. What will the output for the following piece of code be?


```
main()
{
    printf("%p",main)
}
```

 - (a) Garbage value
 - (b) Infinite loop
 - (c) Some address will get printed
 - (d) Runtime error
3. Which of the following is the default parameter passing mechanism?
 - (a) Call by value
 - (b) Call by reference
 - (c) Call by value result
 - (d) There is no default parameter passing mechanism

4.48 Computer Programming

4. Which of the following rules does not hold true for a function declaration?
 - (a) The parameter list must be separated using commas.
 - (b) The parameter names do not need to be the same in the prototype declaration and the function definition.
 - (c) It is mandatory to use parameter names in the declaration.
 - (d) The return type must be void if no value is returned.
5. A function declaration consists of four parts. Which of them is wrongly mentioned?
 - (a) Function type
 - (b) Function name
 - (c) Parameter list
 - (d) Terminating comma
6. What will the output of the following piece of code be?

```
main()
{
    static int x=3;
    printf("%d", x--);
    if (x)
        main();
}
```

 - (a) 3 3 3
 - (b) 3 2 1
 - (c) Compilation error
 - (d) Infinite loop
7. Identify the complete function among the following.
 - (a) `int func();`
 - (b) `int func(intx){return x=x+1;}`
 - (c) `void func(int){printf("Welcome");}`
 - (d) `void func(x)`
8. What does the function `scanf()` return?
 - (a) The actual values read for each argument
 - (b) ASCII value of the input read
 - (c) 1
 - (d) 0
9. What kind of scope do functions have?
 - (a) Local scope
 - (b) Block scope
 - (c) Function scope
 - (d) File scope
10. Which of the following is created automatically when the function is called and then destroyed automatically once the function is exited?
 - (a) Register Variables
 - (b) Static variables
 - (c) Automatic variables
 - (d) External variables
11. Which of the following are also known as global variables?
 - (a) External variables
 - (b) Register variables
 - (c) Static variables
 - (d) Automatic variables
12. In what form are automatic variables stored when a function is recursively called?
 - (a) Stack
 - (b) Queue
 - (c) Array
 - (d) Linked list
13. What type of value do functions return by default?
 - (a) char
 - (b) string
 - (c) integer
 - (d) float

Answers

- | | | | | |
|---------|---------|---------|--------|---------|
| 1. (b) | 2. (c) | 3. (a) | 4. (c) | 5. (d) |
| 6. (b) | 7. (b) | 8. (b) | 9. (d) | 10. (c) |
| 11. (a) | 12. (a) | 13. (c) | | |



Review Questions

1. State whether the following statements are *true* or *false*.
 - (a) Any name can be used as a function name.
 - (b) A function without a **return** statement is illegal.
 - (c) A function prototype must always be placed outside the calling function.
 - (d) The variable names used in prototype should match those used in the function definition.
 - (e) The return type of a function is **int** by default.
 - (f) When variable values are passed to functions, a copy of them are created in the memory.
 - (g) A function can be defined within the **main** function.
 - (h) A function can be defined and placed before the **main** function.
 - (i) C functions can return only one value under their function name.
 - (j) A function in C should have at least one argument.
 - (k) Only a **void** type function can have **void** as its argument.
 - (l) Program execution always begins in the main function irrespective of its location in the program.
 - (m) In parameter passing by pointers, the formal parameters must be prefixed with the symbol ***** in their declarations.
 - (n) In parameter passing by pointers, the actual parameters in the function call may be variables or constants.
 - (o) An user-defined function must be called at least once; otherwise a warning message will be issued.
 - (p) A function can call itself.
 - (q) In passing arrays to functions, the function call must have the name of the array to be passed without brackets.
 - (r) In passing strings to functions, the actual parameter must be name of the string post-fixed with size in brackets.
 - (s) Global variables are visible in all blocks and functions in the program.
 - (t) Global variables cannot be declared as **auto** variables.
2. Fill in the blanks in the following statements.
 - (a) The parameters used in a function call are called _____.
 - (b) In prototype declaration, specifying _____ is optional.
 - (c) A _____ aids the compiler to check the matching between the actual arguments and the formal ones.
 - (d) In passing by pointers, the variables of the formal parameters must be prefixed with _____ in their declaration.
 - (e) By default, _____ is the return type of a C function.
 - (f) A function that calls itself is known as a _____ function.
 - (g) A variable declared inside a function is called _____.
 - (h) _____ refers to the region where a variable is actually available for use.
 - (i) If a local variable has to retain its value between calls to the function, it must be declared as _____.
 - (j) A variable declared inside a function by default assumes _____ storage class.
3. The **main** is a user-defined function. How does it differ from other user-defined functions?
4. Describe the two ways of passing parameters to functions. When do you prefer to use each of them?
5. What is prototyping? Why is it necessary?
6. Distinguish between the following:
 - (a) Actual and formal arguments
 - (b) **&** operator and ***** operator
 - (c) Global and local variables
 - (d) Automatic and static variables
 - (e) Scope and visibility of variables
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.
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);`
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)`
10. 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)`
11. 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)`

12. 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);
}

```

13. 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;
}

```

14. 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);

```

- 15. Enumerate the rules that apply to a function call.
- 16. Summarize the rules for passing parameters to functions by pointers.
- 17. What are the rules that govern the passing of arrays to function?
- 18. State the problems we are likely to encounter when we pass global variables as parameters to functions.





## Debugging Exercises

1. 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;
}
```

2. Find errors in the following function calls:

(a) `void xyz ( );`  
 (b) `xyx ( void );`  
 (c) `xyx ( int x, int y);`  
 (d) `xyzz ( );`  
 (e) `xyz ( ) + xyz ( );`



## Programming Exercises

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.

2. Write a function **space(x)** that can be used to provide a space of **x** positions between two output numbers. Demonstrate its application.

3. Use recursive function calls to evaluate

$$f(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

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.

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  $15$ .

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.
7. Write a function **prime** that returns 1 if its argument is a prime number and returns zero otherwise.
8. Write a function that will scan a character string passed as an argument and convert all lowercase characters into their uppercase equivalents.
9. Develop a top\_down modular program to implement a calculator. The program should request the user to input two numbers and



#### 4.52 Computer Programming

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.

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.

$$\text{Perimeter} = a + b + c$$

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

where  $s = (a + b + c)/2$

11. Write a function that can be called to find the largest element of an m by n matrix.
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.
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.
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.

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.

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.

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).

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.

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?

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).



---

# 5

# Arrays

---

## CHAPTER OUTLINE

- |                                           |                                              |                              |
|-------------------------------------------|----------------------------------------------|------------------------------|
| 5.1 Introduction                          | 5.4 Initialization of One-Dimensional Arrays | 5.7 Multi-Dimensional Arrays |
| 5.2 One-Dimensional Arrays                | 5.5 Two-Dimensional Arrays                   | 5.8 Dynamic Arrays           |
| 5.3 Declaration of One-Dimensional Arrays | 5.6 Initializing Two-Dimensional Arrays      | 5.9 Case Studies             |

---

## 5.1 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.

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.



## 5.2 Computer Programming

For example,

```
salary [10]
```

represents the salary of 10<sup>th</sup> employee. While the complete set of values is referred to as an array, individual values are called *elements*.

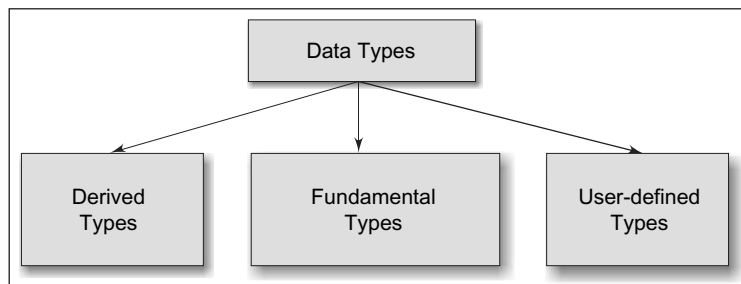
The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, 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

### 5.1.1 Data Structures

C supports a rich set of derived and user-defined data types in addition to a variety of fundamental types as shown:



- |             |                   |                |
|-------------|-------------------|----------------|
| - Arrays    | - Integral Types  | - Structures   |
| - Functions | - Float Types     | - Unions       |
| - Pointers  | - Character Types | - Enumerations |

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

## 5.2 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

**$x[1], x[2], x[3], \dots, x[n]$**

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.*

## 5.3 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



## 5.4 Computer Programming

`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:

- 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 5.1** Write a program using a single-subscripted variable to evaluate the following expressions:

---

$$\text{Total} = \sum_{i=1}^{10} x_i^2$$

The values of  $x_1, x_2, \dots$  are read from the terminal.

Program in Fig. 5.1 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++)
 {
 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. 5.1** Program to illustrate one-dimensional array



**Example 5.2** Write a C program to find the sum marks of n students using arrays.

The program in Fig. 5.2 uses arrays to find the sum marks of n students.

```

Program
#include <stdio.h>
int main()
{
 int marks[10],i,n,sum=0;
 /*marks is an array of int type*/
 clrscr();
 printf("Enter number of students: ");
 scanf("%d",&n);
 for(i=0;i<n;++i)
 {
 printf("Enter marks of student%d: ",i+1);
 scanf("%d",&marks[i]);
 sum+=marks[i];
 }
 printf("Sum= %d",sum);
 getch();
 return 0;
}
Output
Enter number of students: 4
Enter marks of student1: 12
Enter marks of student2: 1
Enter marks of student3: 2
Enter marks of student4: 3
Sum= 18

```

**Fig. 5.2** Program to find the sum marks of n students using arrays

## 5.4 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

### 5.4.1 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 8.)

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.

## 5.4.2 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;
}


```



## 5.8 Computer Programming

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 5.3** 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. 5.3 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. */
 ++ 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. 5.3** Program for frequency counting

Note that we have used an initialization statement.

```
int group [COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
```

which can be replaced by

```
int group [COUNTER] = {0};
```

This will initialize all the elements to zero.

---

**Example 5.4** The program shown in Fig. 5.4 shows the algorithm, flowchart and the complete C program to find the two's complement of a binary number.

---

#### Algorithm

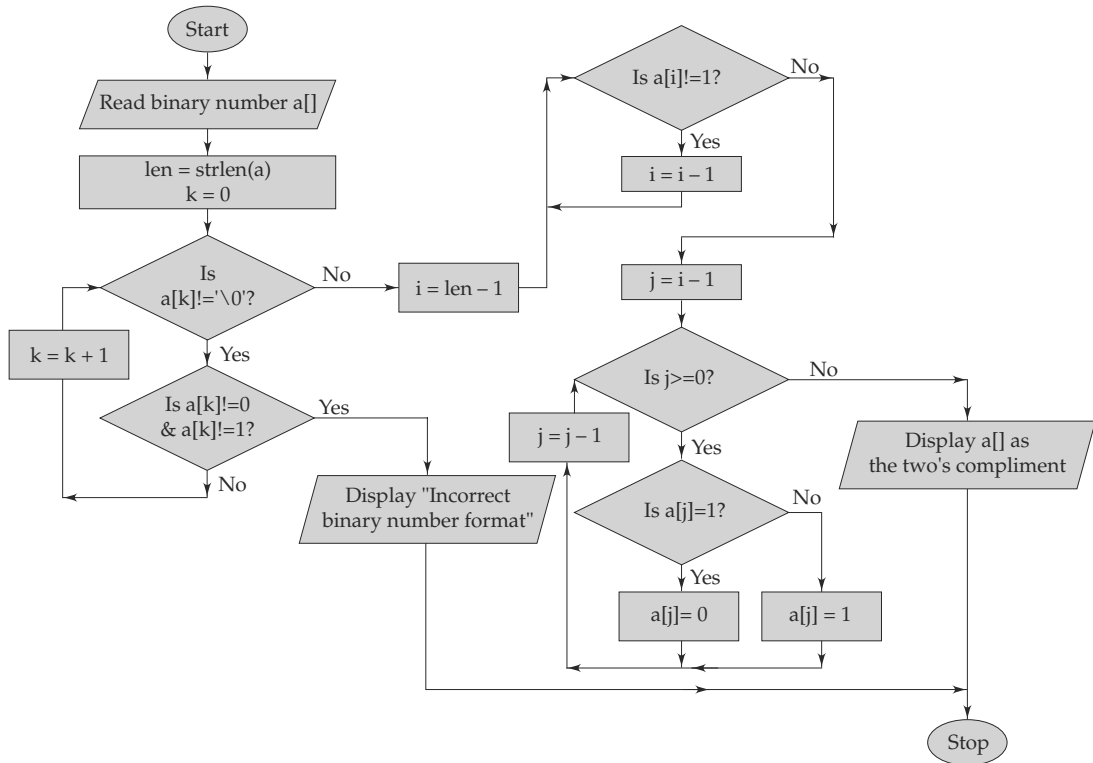
- Step 1 - Start
- Step 2 - Read a binary number string (a[])
- Step 3 - Calculate the length of string str (len)
- Step 4 - Initialize the looping counter k=0
- Step 5 - Repeat Steps 6-8 while a[k] != '\0'
- Step 6 - If a[k] != 0 AND a[k] != 1 goto Step 7 else goto Step 8
- Step 7 - Display error "Incorrect binary number format" and terminate the program
- Step 8 - k = k + 1
- Step 9 - Initialize the looping counter i = len - 1
- Step 10 - Repeat Step 11 while a[i] != '1'



## 5.10 Computer Programming

Step 11 -  $i = i - 1$   
 Step 12 - Initialize the looping counter  $j = i - 1$   
 Step 13 - Repeat Step 14-17 while  $j \geq 0$   
 Step 14 - If  $a[j]=1$  goto Step 15 else goto Step 16  
 Step 15 -  $a[j]='0'$   
 Step 16 -  $a[j]='1'$   
 Step 17 -  $j = j - 1$   
 Step 18 - Display  $a[]$  as the two's compliment  
 Step 19 - Stop

Flowchart



Program

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{
 char a[16];
 int i,j,k,len;

```



```

clrscr();
printf("Enter a binary number: ");
gets(a);
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);
 }
}
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';
}
printf("\n2's compliment = %s",a);
getch();
}

```

Output

```

Enter a binary number: 01011001001
2's compliment = 10100110111

```

**Fig. 5.4** Algorithm, flowchart and C program to find two's compliment of a binary number

### 5.4.3 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.



## 5.12 Computer Programming

*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.

## 5.5 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  $i^{\text{th}}$  row and  $j^{\text{th}}$  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

$v[4][3]$

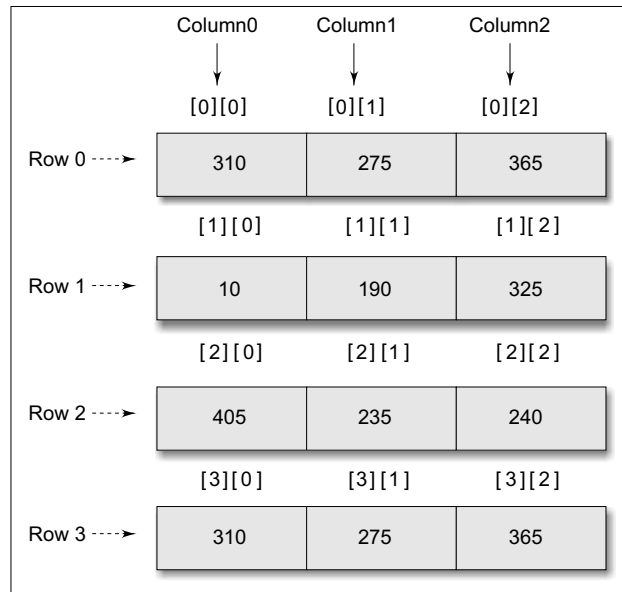
Two-dimensional arrays are declared as follows:

```
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.

Two-dimensional arrays are stored in memory, as shown in Fig. 5.5. 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.





**Fig. 5.5** Representation of a two-dimensional array in memory

**Example 5.5** 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. 5.6 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 ;

```



## 5.14 Computer Programming

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. 5.6** Program to print multiplication table using two-dimensional array

**Example 5.6** 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. 5.7. 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:



$$(a) \text{ Total sales by } m^{\text{th}} \text{ girl} = \sum_{j=0}^2 \text{value}[m][j] \text{ (girl\_total}[m])$$

$$(b) \text{ Total value of } n^{\text{th}} \text{ item} = \sum_{i=0}^3 \text{value}[i][n] \text{ (item\_total}[n])$$

$$\begin{aligned} (c) \text{ Grand total} &= \sum_{i=0}^3 \sum_{j=0}^2 \text{value}[i][j] \\ &= \sum_{i=0}^3 \text{girl\_total}[i] \\ &= \sum_{j=0}^2 \text{item\_total}[j] \end{aligned}$$

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];
 }
}
```



## 5.16 Computer Programming

```
/*.....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. 5.7** Illustration of two-dimensional arrays

## 5.6 INITIALIZING TWO-DIMENSIONAL ARRAYS

A 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 5.7** 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.

---



## 5.18 Computer Programming

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  $5 \times 5$  and all the elements are initialized to zero.

The program shown in Fig. 5.8 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;
 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. 5.8** Program to tabulate a survey data

### 5.6.1 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 as follows:



## 5.20 Computer Programming

|     |   | 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.

---

**Example 5.8** The program in Fig. 5.9 shows how to find the transpose of a matrix.

---

#### Algorithm

```
Step 1 - Start
Step 2 - Read a 3 X 3 matrix (a[3][3])
Step 3 - Initialize the looping counter i = 0
Step 4 - Repeat Steps 5-9 while i<3
Step 5 - Initialize the looping counter j = 0
Step 6 - Repeat Steps 7-8 while j<3
Step 7 - b[i][j]=a[j][i]
Step 8 - j = j + 1
Step 9 - i = i + 1
Step 10 - Display b[][] as the transpose of the matrix a[][]
Step 11 - Stop
```

#### Program

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

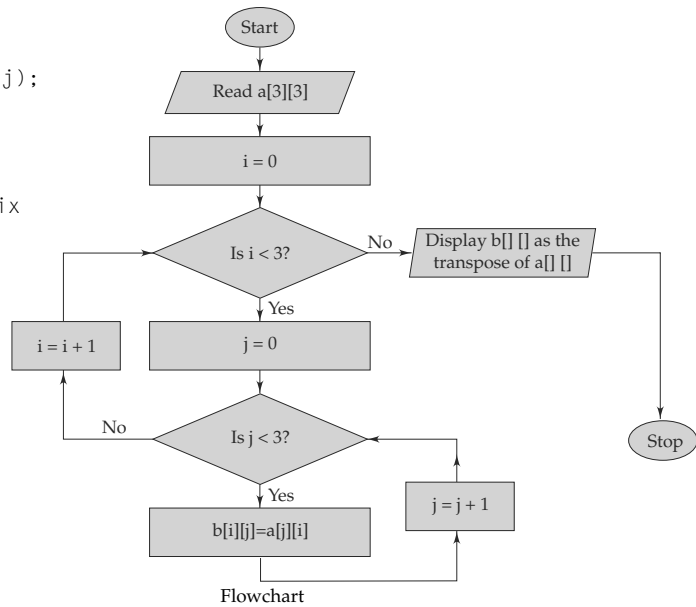


```

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]);
 }
 }
 printf("\nThe entered matrix\n\n");
 for(i=0;i<3;i++)
 {
 printf("\n");
 for(j=0;j<3;j++)
 {
 printf("%d\t",a[i][j]);
 }
 }
 for(i=0;i<3;i++)
 {
 for(j=0;j<3;j++)
 {
 b[i][j]=a[j][i];
 }
 }
 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]);
 }
 getch();
 }
}

```



Output

```

Enter a 3 X 3 matrix:
a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[1][0] = 4

```



## 5.22 Computer Programming

```
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
```

**Fig. 5.9** Program to find transpose of a matrix

---

**Example 5.9** The program in Fig. 5.10 shows how to multiply the elements of two  $N \times N$  matrices.

---

Program

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int a1[10][10],a2[10][10],c[10][10],i,j,k,a,b;
 clrscr();
 printf("Enter the size of the square matrix\n");
 scanf ("%d", &a);
 b=a;
 printf("You have to enter the matrix elements in row-wise fashion\n");
 for(i=0;i<a;i++)
 {
 for(j=0;j<b;j++)
 {
 printf("\nEnter the next element in the 1st matrix=");
 scanf("%d",&a1[i][j]);
 }
 }
 for(i=0;i<a;i++)
 {
 for(j=0;j<b;j++)
 {
 printf("\n\nEnter the next element in the 2nd matrix=");
 scanf("%d",&a2[i][j]);
 }
 }
 printf("\n\nEntered matrices are\n");
 for(i=0;i<a;i++)
```



```

{ printf("\n");
for(j=0;j<b;j++)
printf(" %d ",a1[i][j]);
}
printf("\n");
for(i=0;i<a;i++)
{ printf("\n");
for(j=0;j<b;j++)
printf(" %d ",a2[i][j]);
}
printf("\n\nProduct of the two matrices is\n");
for(i=0;i<a;i++)
 for(j=0;j<b;j++)
 {
 c[i][j]=0;
 for(k=0;k<a;k++)
 c[i][j]=c[i][j]+a1[i][k]*a2[k][j];
 }
 for(i=0;i<a;i++)
 { printf("\n");
 for(j=0;j<b;j++)
 printf(" %d ",c[i][j]);
 }
 getch();
}

```

#### Output

```

Enter the size of the square matrix
2
You have to enter the matrix elements in row-wise fashion
Enter the next element in the 1st matrix=1
Enter the next element in the 1st matrix=0
Enter the next element in the 1st matrix=2
Enter the next element in the 1st matrix=3
Enter the next element in the 2nd matrix=4
Enter the next element in the 2nd matrix=5
Enter the next element in the 2nd matrix=0
Enter the next element in the 2nd matrix=2
Entered matrices are
1 0
2 3
4 5
0 2
Product of the two matrices is
4 5
8 16

```

**Fig. 5.10** Program for  $N \times N$  matrix multiplication



---

**Example 5.10** Write a C program to find sum of two matrix of order  $2 \times 2$  using multidimensional arrays where, elements of matrix are entered by user.

---

Program

```
#include <stdio.h>
int main()
{
 float a[2][2], b[2][2], c[2][2];
 int i,j;
 clrscr();
 printf("Enter the elements of 1st matrix\n");
 for(i=0;i<2;++i)
 for(j=0;j<2;++j)
 {
 printf("Enter a%d%d: ",i+1,j+1);
 scanf("%f",&a[i][j]);
 }
 printf("Enter the elements of 2nd matrix\n");
 for(i=0;i<2;++i)
 for(j=0;j<2;++j)
 {
 printf("Enter b%d%d: ",i+1,j+1);
 scanf("%f",&b[i][j]);
 }
 for(i=0;i<2;++i)
 for(j=0;j<2;++j)
 {
 c[i][j]=a[i][j]+b[i][j];
 /*we can directly add two matrix by '+' operator*/
 }
 printf("\n Addition Of Matrix:");
 for(i=0;i<2;++i)
 for(j=0;j<2;++j)
 {
 printf("%.1f\t",c[i][j]);
 if(j==1)
 printf("\n");
 }
 getch();
 return 0;
}
```

Output

```
Enter the elements of 1st matrix
Enter a11: 1
Enter a12: 2
Enter a21: 3
Enter a22: 4

Enter the elements of 2nd matrix
Enter b11: 4
Enter b12: 3
```



```

Enter b21: 2
Enter b22: 1

Addition Of Matrix:
5.0 5.0
5.0 5.0

```

**Fig. 5.11** C program to find sum of two matrix of order  $2 \times 2$  using multidimensional arrays

## 5.7 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 example 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.

Remember that a three-dimensional array can be represented as a series of two-dimensional arrays as shown below:

| Year 1 | month city | 1 | 2 | ..... | 12 |
|--------|------------|---|---|-------|----|
|        | 1          |   |   |       |    |
|        | .          |   |   |       |    |
|        | .          |   |   |       |    |
|        | .          |   |   |       |    |
|        | 5          |   |   |       |    |
| Year 2 | month city | 1 | 2 | ..... | 12 |
|        | 1          |   |   |       |    |
|        | .          |   |   |       |    |
|        | .          |   |   |       |    |
|        | .          |   |   |       |    |
|        | 5          |   |   |       |    |



ANSI C does not specify any limit for array dimension. However, most compilers permit seven to ten dimensions. Some allow even more.

## 5.8 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.

## 5.9 CASE STUDIES

### 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. 5.12. The sorting algorithm used is as follows:

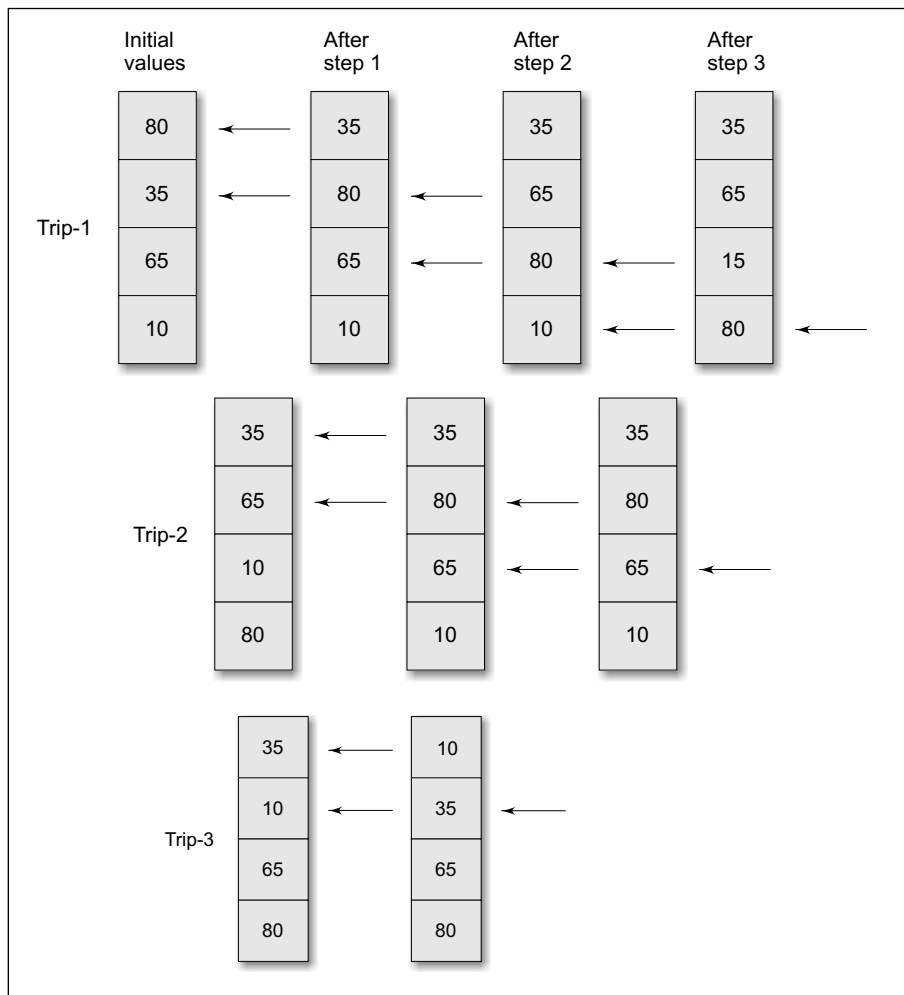
1. Compare the first two elements in the list, say  $a[1]$ , and  $a[2]$ . If  $a[2]$  is smaller than  $a[1]$ , then interchange their values.
2. Compare  $a[2]$  and  $a[3]$ ; interchange them if  $a[3]$  is smaller than  $a[2]$ .
3. Continue this process till the last two elements are compared and 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++)
```



## 5.28 Computer Programming

```
{ /* 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. 5.12** 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. 5.13.

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);
}
```



### 5.30 Computer Programming

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. 5.13** 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:

|                 | 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. 5.14. 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]);
 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]);
 scanf("%c",&response[i]);
 response[i] = '\0';
 for(i=0; i < ITEMS; i++)
 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
abcdabcdabcdabcdabcd
Input responses of student-1

```



### 5.32 Computer Programming

```
abcdabcdabcdabcdabcdabcd
Student-1
Score is 25 out of 25
Response to the following items are wrong
NIL
Input responses of student-2
abcdcdcbabcdabcdcdcdcdcd
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. 5.14** 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:

- (a) Value of weekly production and sales.
- (b) Total value of all the products manufactured.
- (c) Total value of all the products sold.
- (d) 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,

$$Mvalue[i][j] = Mij \times Cj$$

$$Svalue[i][j] = Sij \times Cj$$

A program to generate the required outputs for the review meeting is shown in Fig. 5.15. 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 jth type product manufactured during the month

$$= \sum_{i=1}^4 Mvalue[i][j]$$

Sproduct[j] = Value of jth type product sold during the month

$$= \sum_{i=1}^4 Svalue[i][j]$$

Mtotal = Total value of all the products manufactured during the month

$$= \sum_{i=1}^4 Mweek[i] = \sum_{j=1}^5 Mproduct[j]$$

Stotal = Total value of all the products sold during the month

$$= \sum_{i=1}^4 Sweek[i] = \sum_{j=1}^5 Sproduct[j]$$

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]);
```



### 5.34 Computer Programming

```
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];
 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++)

```



### 5.36 Computer Programming

```
 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");
} /* 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 SALE
 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
 Product(1) 500 450
 Product(2) 1100 940
 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. 5.15 Program for production and sales analysis



## Key Terms

- **Array:** Is a fixed-size sequenced collection part of elements of the same data type.
- **One-dimensional Array:** Is a list of items that has one variable name and one subscript to access the items.
- **Structured Data Types:** Represent data values that have a structure of some sort. For example, arrays, structures, etc.
- **Searching:** Is the process of finding the location of the specified element in the list.
- **Sorting:** Is the process of rearranging elements in the list as per ascending or descending order.
- **Two-dimensional Array:** Is an array of arrays that has two subscripts for accessing its values. It is used to represent table or matrix data.
- **Multi-dimensional Array:** Is an array with more than one dimension. Examples of multi-dimensional arrays are two-dimensional array, three-dimensional array and so on.
- **Dynamic Arrays:** Are the arrays declared using dynamic memory allocation technique.
- **Dynamic Memory Allocation:** Is the process of allocating memory at run time.
- **Static Arrays:** Are the arrays declared using static memory allocation technique.
- **Static Memory Allocation:** Is the process of allocating memory at compile time.



## Just Remember

1. We need to specify three things, namely, name, type and size, when we declare an array.
2. Use of invalid subscript is one of the common errors. An incorrect or invalid index may cause unexpected results.
3. Always remember that subscripts begin at 0 (not 1) and end at size - 1.
4. Defining the size of an array as a symbolic constant makes a program more scalable.
5. 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.
6. Do not forget to initialize the elements; otherwise they will contain “garbage”.
7. Supplying more initializers in the initializer list is a compile time error.
8. 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.



9. 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++)
```
10. Referring a two-dimensional array element like `x[i, j]` instead of `x[i][j]` is a compile time error.
11. Leaving out the subscript reference operator `[ ]` in an assignment operation is compile time error.
12. When initializing character arrays, provide enough space for the terminating null character.
13. Make sure that the subscript variables have been properly initialized before they are used.
14. During initialization of multi-dimensional arrays, it is an error to omit the array size for any dimension other than the first.
15. While using static arrays, choose the array size in such a way that the memory space is efficiently utilized and there is no overflow condition.



### Multiple Choice Questions

1. Which of the following statements hold true?
  - (a) An array is a variable size sequenced collection of the same data type.
  - (b) An array is a fixed size sequenced collection of the same data type.
  - (c) An array is a fixed size sequenced collection of different data types.
  - (d) An array is a fixed size unorganized collection of the same data type.
2. How are elements in an array located?
  - (a) Randomly
  - (b) Sequentially
  - (c) Depends on how an array is defined
  - (d) Depends on the type of array
3. Identify the correct way to initialize an array.
  - (a) `int num{} = 1,2,3,4,5`
  - (b) `int n{}= {5,10,15,20};`
  - (c) `int num[5]= {1,2,3,4,5};`
  - (d) `int n(5)= {1,2,3,4,5};`
4. Suppose `z` is an array. Which of the following operations can be carried out?
  - (a) `++z`
  - (b) `z++`
  - (c) `z*2`
  - (d) `z+1`
5. How many maximum dimensions can an array in C have?
  - (a) 3
  - (b) 8
  - (c) 50
  - (d) Can have any no. of dimensions. Practically the memory size and compiler are the only limiters.
6. What will be the output for the following piece of code?
 

```
void main()
{
 int x[8]={1,2,3,4,5};
 printf ("%d", x[5]);
}
```

  - (a) 0
  - (b) 5
  - (c) 6
  - (d) Garbage value
7. What will be the output for the following piece of code?
 

```
void main()
{
 char str1[] = "Hello";
 char str2[] = "hello";
 if(str1==str2)
 printf(" Strings are Equal");
 else
```



## 5.40 Computer Programming

- ```
printf("This is
not how you compare
strings");
}
```
- (a) Strings are equal
(b) This is not how you compare strings
(c) Runtime error
(d) Garbage value
8. Identify the incorrect way to initialize a two dimensional array amongst the following?
- (a) `int arr[2][3]={0,0,0,1,1,1};`
(b) `int arr[2][3]={{0,0,0},{1,1,1}};`
(c) `int arr[2,3]= {{0,0,0},{1,1,1}};`
(d) `int arr[2][3]={{0,0,0}, {1,1,1}}`
9. What are the arrays created at run time called?
- (a) Multi dimensional array
(b) One dimensional array
(c) Static array
(d) Dynamic array
10. Which of the following will identify the 8th element is an array of 50 elements?
- (a) `Arr[7]` (b) `Arr[8]`
(c) `Arr{7}` (d) `Arr{8}`
11. `int x[5]={3,4,5}`
What will be the value of `x[4]`?
- (a) 4 (b) 6
(c) 0 (d) Garbage value

Answers

- | | | | | |
|---------|--------|--------|--------|---------|
| 1. (b) | 2. (b) | 3. (c) | 4. (d) | 5. (d) |
| 6. (a) | 7. (b) | 8. (c) | 9. (d) | 10. (a) |
| 11. (c) | | | | |



Review Questions

1. State whether the following statements are *true* or *false*.
- (a) An array can store infinite data of similar type.
(b) In declaring an array, the array size can be a constant or variable or an expression.
(c) The declaration `int x[2] = {1,2,3};` is illegal.
(d) When an array is declared, C automatically initializes its elements to zero.
(e) An expression that evaluates to an integral value may be used as a subscript.
(f) In C, by default, the first subscript is zero.
(g) When initializing a multidimensional array, not specifying all its dimensions is an error.
- (h) When we use expressions as a subscript, its result should be always greater than zero.
(i) In C, we can use a maximum of 4 dimensions for an array.
(j) Accessing an array outside its range is a compile time error.
(k) A **char** type variable cannot be used as a subscript in an array.
(l) An unsigned long int type can be used as a subscript in an array.
2. Fill in the blanks in the following statements.
- (a) The variable used as a subscript in an array is popularly known as _____ variable.
(b) An array that uses more than two subscript is referred to as _____ array.

- (c) _____ is the process of arranging the elements of an array in order.
- (d) An array can be initialized either at compile time or at _____.
- (e) An array created using **malloc** function at run time is referred to as _____ array.
3. 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

4. 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];`

5. 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";`
6. What is a data structure? Why is an array called a data structure?
7. What is a dynamic array? How is it created? Give a typical example of use of a dynamic array.
8. 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.
9. Discuss how initial values can be assigned to a multidimensional array.



Debugging Exercises

1. Identify errors, if any, in each of the following array declaration statements, assuming that ROW and COLUMN are declared as symbolic constants:
- (a) `int score (100);`
- (b) `float values [10,15];`
- (c) `char name[15];`
- (d) `float average[ROW],[COLUMN];`
- (e) `double salary [i + ROW]`
- (f) `long int number [ROW]`
- (g) `int sum[];`
- (h) `int array x[COLUMN];`
2. Identify errors, if any, in each of the following initialization statements.
- (a) `int number[] = {0,0,0,0,0};`
- (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;`
3. 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<4; i++)`  
`scanf("%f", B[i]);`
- (b) `for (i=1; i<=5; i++)`



## 5.42 Computer Programming

```

for(j=1; j<=4; j++)
A[i][j] = 0;
(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;

```

4. What is the error in the following program?

```

main ()
{
 int x ;
 float y [] ;

}

```

5. 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) ;
}

```

6. What is the output of the following program?

```

main ()
{
 char string [] = "HELLO WORLD" ;
 int m;
 for (m = 0; string [m] != '\0'; m++)
 if ((m%2) == 0)
 printf("%c", string [m]);
}

```



## Programming Exercises

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  $n$ .

2. The daily maximum temperatures recorded in 10 cities during the month of January (for all 31 days) have been tabulated as follows:

| Day | City |   |       |    |
|-----|------|---|-------|----|
|     | 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.



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.
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.

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:

- Total marks obtained by each student.
  - The highest marks in each subject and the Roll No. of the student who secured it.
  - The student who obtained the highest total marks.
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.

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**.

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

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



#### 5.44 Computer Programming

ordered list of size 2 and an unordered 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.

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 5.9 or use any other sorted list.

11. Write a program that will compute the length of a given character string.

12. Write a program that will count the number occurrences of a specified character in a given line of text. Test your program.

13. Write a program to read a matrix of size  $m \times n$  and print its transpose.

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:

$$\begin{aligned}\text{Sum} &= (1 \times \text{first digit}) + (2 \times \text{second digit}) \\ &\quad + (3 \times \text{third digit}) + \dots \\ &\quad + (9 \times \text{ninth digit}).\end{aligned}$$

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.

15. Write a program to read two matrices A and B and print the following:
  - (a)  $A + B$ ; and
  - (b)  $A - B$ .



---

# 6

# Strings

---

## CHAPTER OUTLINE

|                                                 |                                         |                               |
|-------------------------------------------------|-----------------------------------------|-------------------------------|
| 6.1 Introduction                                | 6.4 Writing Strings to Screen           | 6.7 Comparison of Two Strings |
| 6.2 Declaring and Initializing String Variables | 6.5 Arithmetic Operations on Characters | 6.8 String-Handling Functions |
| 6.3 Reading Strings from Terminal               | 6.6 Putting Strings Together            | 6.9 Table of Strings          |
|                                                 |                                         | 6.10 Case Studies             |

---

## 6.1 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 the following:

- 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.



## 6.2 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 as follows:

```
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.

## 6.3 READING STRINGS FROM TERMINAL

### 6.3.1 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:

|   |   |   |    |   |   |   |   |   |   |
|---|---|---|----|---|---|---|---|---|---|
| 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 6.1** Write a program to illustrate how to read string from terminal.

---

```
Program
#include <stdio.h>
int main()
{
 char name[20];
 clrscr();
 printf("Enter name: ");
 scanf("%s",name);
 /*'%s' is used to read a string*/
 printf("Your name is %s.",name);
 getch();
 return 0;
}
Output
Enter name: Sachin Tendulkar
Your name is Sachin Tendulkar.
```

**Fig. 6.1** Reading string from terminal using C program

---

**Example 6.2** Write a program to read a series of words from a terminal using scanf function.

---

The program shown in Fig. 6.2 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*.

```
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. 6.2** 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, the two following 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 |

### 6.3.2 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. 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.



## 6.6 Computer Programming

### 6.3.3 Using getchar and gets Functions

We can use `getchar` 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 following form:

```
char ch;
ch = getchar();
```

**NOTE:** `getchar` function has no parameters.

---

**Example 6.3** Write a program to read a line of text containing a series of words from the terminal.

---

The program shown in Fig. 6.3 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();
 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. 6.3** 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 6.4** Write a program to copy one string into another and count the number of characters copied.

---

The program is shown in Fig. 6.4. 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);

```



## 6.8 Computer Programming

```
 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. 6.4** Copying one string into another

---

**Example 6.5** The program in Fig. 6.5 shows how to write a program to find the number of vowels and consonants in a text string. Elucidate the program and flowchart for the program.

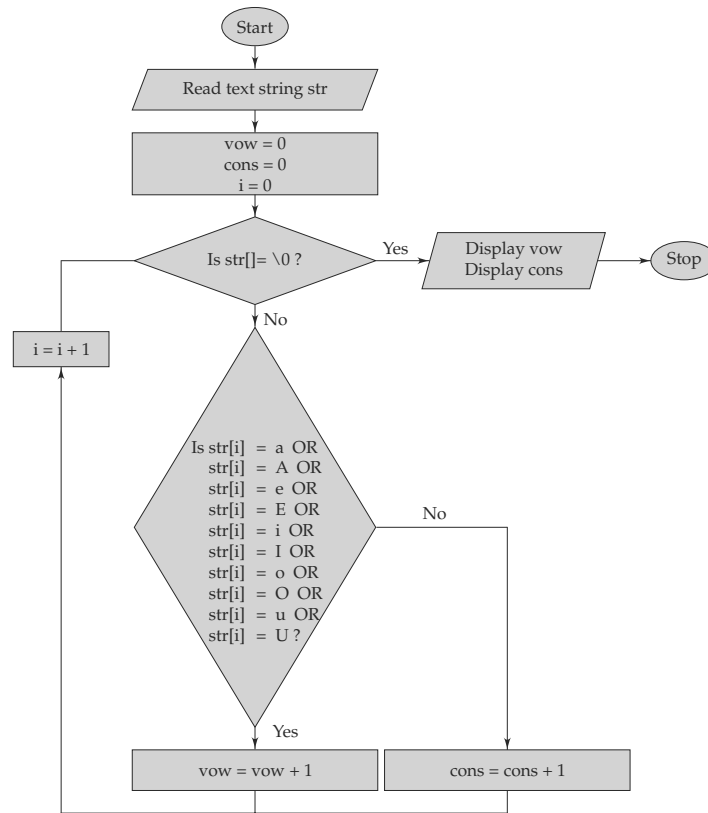
---

Algorithm

```
Step 1 - Start
Step 2 - Read a text string (str)
Step 3 - Set vow = 0, cons = 0, i = 0
Step 4 - Repeat steps 5-8 while (str[i]!='\0')
Step 5 - if str[i] = 'a' OR str[i] = 'A' OR str[i] = 'e' OR str[i] = 'E' OR str[i] = 'i' OR str[i] = 'I' OR str[i] = 'o' OR str[i] = 'O' OR str[i] = 'u' OR str[i] = 'U' goto Step 6 else goto Step 7
Step 6 - Increment the vowels counter by 1 (vow=vow+1)
Step 7 - Increment the consonants counter by 1 (cons=cons+1)
Step 8 - i = i + 1
Step 9 - Display the number of vowels and consonants (vow, cons)
Step 10 - Stop
```



## Flowchart



## Program

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
void main()
{
 char str[30];
 int vow=0,cons=0,i=0;
 clrscr();
 printf("Enter a string: ");
 gets(str);
 while(str[i] != '\0')
 {
 if(str[i]== 'a' || str[i]=='A' || str[i]=='e' || str[i]=='E' || str[i]=='i'
 || str[i]=='I' || str[i]=='o' || str[i]=='O' || str[i]=='u' || str[i]=='U')
 vow++;
 else
 cons++;
 }
}

```



## 6.10 Computer Programming

```
 i++;
 }
 printf("\nNumber of Vowels = %d",vow);
 printf("\nNumber of Consonants = %d",cons);
 getch();
}
```

Output

```
Enter a string: Chennai
Number of Vowels = 3
Number of Consonants = 4
```

**Fig. 6.5** Program to find the number of vowel and consonants in a text string

---

**Example 6.6** Write a C program to remove all special characters in a string except alphabet.

---

Program

```
#include<stdio.h>
int main()
{
 char line[150];
 int i,j;
 clrscr();
 printf("Enter a string: ");
 gets(line);
 /*gets will read a string from the user*/
 for(i=0; line[i]!='\0'; ++i)
 {
 while (!((line[i]>='a'&&line[i]<='z') || (line[i]>='A'&&line[i]<='Z' ||
line[i]=='\0'))))
 {
 for(j=i;line[j]!='\0';++j)
 {
 line[j]=line[j+1];
 }
 line[j]='\0';
 }
 }
 printf("Output String: ");
 puts(line);
 /*puts() will display a string on user screen*/
}
```



```

 getch();
 return 0;
}
Output
Enter a string: c.programming@gmail.com
Output String: cprogramminggmailcom

```

**Fig. 6.6** Removing all special characters in a string except alphabet

## 6.4 WRITING STRINGS TO SCREEN

### 6.4.1 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.

---

**Example 6.7** 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. 6.7. The output illustrates the following features of the `%s` specifications.

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("%.15s\n", country);
}

```



## 6.12 Computer Programming

```
printf("%.3s\n", country);
printf("%s\n", country);
printf("----- \n");
}
```

Output

```
123456789012345

United Kingdom
United Kingdom
United
United
Uni
United Kingdom

```

**Fig. 6.7** 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 6.8** 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. 6.8, for variable specifications **%12.\*s**, **%.\*s**, and **%\*.1s** are shown in Fig. 6.9, 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
CProgram
CProgra
CProgr
CProg
CPro
CPr
CP
C

```

**Fig. 6.8** Illustration of variable field specifications by printing sequences of characters



## 6.14 Computer Programming

|              |              |                                                                           |
|--------------|--------------|---------------------------------------------------------------------------|
| C            | C            | C                                                                         |
| CP           | CP           | C <br>C                                                                   |
| CPr          | CPr          | C <br>C <br>C                                                             |
| CPro         | CPro         | C <br>C <br>C <br>C                                                       |
| CProg        | CProg        | C <br>C <br>C <br>C <br>C                                                 |
| CProgr       | CProgr       | C <br>C <br>C <br>C <br>C <br>C                                           |
| CProgra      | CProgra      | C <br>C <br>C <br>C <br>C <br>C <br>C                                     |
| CProgram     | CProgram     | C <br>C <br>C <br>C <br>C <br>C <br>C <br>C                               |
| CProgramm    | CProgramm    | C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C                         |
| CProgrammi   | CProgrammi   | C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C                   |
| CProgrammin  | CProgrammin  | C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C             |
| CProgramming | CProgramming | C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C <br>C |
| 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. 6.9** Further illustrations of variable specifications

## 6.4.2 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 4 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.

## 6.5 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

```
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 6.9** 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. 6.10. 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. 6.10** Printing of the alphabet set in decimal and character form

## 6.6 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*. Figure 6.11 illustrates the concatenation of three strings.



**Example 6.10** 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. 6.11. 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. 6.11** Concatenation of strings



## 6.7 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");
```

## 6.8 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.

### 6.8.1 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:



|         |   |   |   |   |   |    |   |   |   |   |   |   |
|---------|---|---|---|---|---|----|---|---|---|---|---|---|
| Part1 = | 0 | 1 | 2 | 3 | 4 | 5  | 6 | 7 | 8 | 9 | 0 | 1 |
|         | V | E | R | Y |   | \0 |   |   |   |   |   |   |

|         |   |   |   |   |    |   |   |
|---------|---|---|---|---|----|---|---|
| Part2 = | 0 | 1 | 2 | 3 | 4  | 5 | 6 |
|         | G | O | O | D | \0 |   |   |

|         |   |   |   |    |   |   |   |
|---------|---|---|---|----|---|---|---|
| Part3 = | 0 | 1 | 2 | 3  | 4 | 5 | 6 |
|         | B | A | D | \0 |   |   |   |

Execution of the statement

```
strcat(part1, part2);
```

will result in:

|         |   |   |   |   |   |   |   |   |   |    |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|----|---|---|---|
| Part1 = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 0 | 1 | 2 |
|         | V | E | R | Y |   | G | O | O | D | \0 |   |   |   |

|         |   |   |   |   |    |   |   |
|---------|---|---|---|---|----|---|---|
| Part2 = | 0 | 1 | 2 | 3 | 4  | 5 | 6 |
|         | G | O | O | D | \0 |   |   |

while the statement

will result in:

|         |   |   |   |   |   |   |   |   |    |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|----|---|---|---|---|
| Part1 = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8  | 9 | 0 | 1 | 2 |
|         | V | E | R | Y |   | B | A | D | \0 |   |   |   |   |

|         |   |   |   |    |   |   |   |
|---------|---|---|---|----|---|---|---|
| Part3 = | 0 | 1 | 2 | 3  | 4 | 5 | 6 |
|         | 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**.

## 6.8.2 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**.

### 6.8.3 strcpy() Function

The **strcpy** function works almost like a string-assignment operator. It takes the following form:

**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**.

### 6.8.4 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 6.11** **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. 6.12. 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. 6.12 Illustration of string handling functions



---

**Example 6.12** The program in Fig. 6.13 shows how to write a C program that reads a string and prints if it is a palindrome or not.

---

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 string %s is a palindrome",str);
 else
 printf("\nThe string %s is not a palindrome",str);
 getch();
}
```

Output

```
Enter a string: nitin
The string nitin is a palindrome
```

**Fig. 6.13** Program to check if a string is palindrome or not

### 6.8.5 Other String Functions

The header file **<string.h>** contains many more string manipulation functions. They might be useful in certain situations.

#### **strncpy**

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.

### strncmp

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.

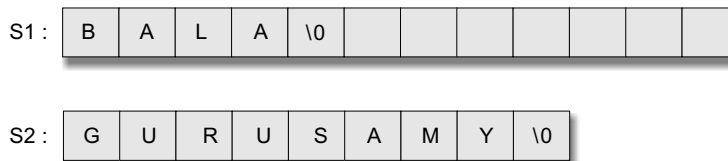
- (a) 0 if they are equal;
- (b) negative number, if **s1** sub-string is less than **s2**; and
- (c) positive number, otherwise.

### strncat

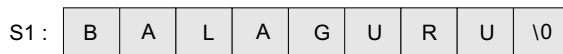
This is another concatenation function that takes three parameters as shown below:

```
strncat (s1, s2, n);
```

This call will concatenate the left-most **n** characters of **s2** to the end of **s1**. Example:



After **strncat** (s1, s2, 4); execution:



### strstr

It is a two-parameter function that can be used to locate a sub-string in a string. This takes the following 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');
```



## 6.24 Computer Programming

will locate the first occurrence of the character 'm' and the call

```
strchr(s1, 'm');
```

will locate the last occurrence of the character 'm' in the string **s1**.

### Warning



- 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.

## 6.9 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 6.13** 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. 6.14. It employs the method of bubble sorting.

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. 6.14** 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.

## 6.10 CASE STUDIES

### 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. 6.15. 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. 6.15** 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           |
| -----                 | -----            |
| -----                 | -----            |



## 6.28 Computer Programming

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 6.16 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)
```



```

 {
 /* 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. 6.16** Program to alphabetize a customer list





## Key Terms

- **String:** Is a sequence of characters that is considered as a single data item.
- **strcat:** Concatenates two strings.
- **strcmp:** Compares two strings and determines whether they are equal or not.
- **strcpy:** Copies one string into another.
- **strstr:** Determines whether one string is a subset of another.



## Just Remember

1. Character constants are enclosed in single quotes and string constants are enclosed in double quotes.
2. Allocate sufficient space in a character array to hold the null character at the end.
3. Avoid processing single characters as strings.
4. It is a compile time error to assign a string to a character variable.
5. The header file `<stdio.h>` is required when using standard I/O functions.
6. The header file `<stdlib.h>` is required when using general utility functions.
7. Using the address operator **&** with a **string** variable in the **scanf** function call is an error.
8. Use `%s` format for printing strings or character arrays terminated by null character.
9. Using a string variable name on the left of the assignment operator is illegal.
10. When accessing individual characters in a string variable, it is logical error to access outside the array bounds.
11. Strings cannot be manipulated with operators. Use string functions.
12. Do not use string functions on an array **char** type that is not terminated with the null character.
13. 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.
14. Be aware the return values when using the functions **strcmp** and **strncmp** for comparing strings.
15. 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.
16. The header file `<ctype.h>` is required when using character handling functions.
17. The header file `<string.h>` is required when using string manipulation functions.



## Multiple Choice Questions

1. Which library function is used to compare two strings?  
 (a) `strcmp()` (b) `strlen()`  
 (c) `strstr()` (d) `strchr()`
2. What does `strcmp()` return if two strings are identical?  
 (a) 1 (b) -1  
 (c) 0 (d) True
3. Identify the library function used to find the last occurrence of a character in a string.  
 (a) `strcmp()` (b) `strlen()`  
 (c) `strstr()` (d) `strchr()`



4. Identify the library function which is used to locate a substring in a string.  
 (a) `strnset()` (b) `strchr()`  
 (c) `strstr()` (d) `strlen()`
5. Identify the function which is more appropriate for reading a multi word string.  
 (a) `scanf()` (b) `printf()`  
 (c) `gets()` (d) `puts()`
6. What will be the output for the following piece of code?  

```
#include<stdio.h>
void main()
{
 printf(6+"Hello world ");
}
```
- (a) W (b) World  
 (c) Hello World (d) Hello
7. Which of the following data type does `sprintf()` operate on?  
 (a) Data file (b) String  
 (c) `Stdin` (d) `Stderr`
8. What is meant by string concatenation?  
 (a) Dividing the string equally into two parts  
 (b) Taking a substring out of a string  
 (c) Combining two strings  
 (d) Adding two strings

## Answers

- |        |        |        |        |        |
|--------|--------|--------|--------|--------|
| 1. (a) | 2. (c) | 3. (d) | 4. (c) | 5. (c) |
| 6. (b) | 7. (b) | 8. (c) |        |        |



## Review Questions

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) The input function **gets** has one string parameter.
  - (e) The function **scanf** cannot be used in any way to read a line of text with the white-spaces.
  - (f) The function **getchar** skips white-space during input.
  - (g) In C, strings cannot be initialized at run time.
  - (h) String variables cannot be used with the assignment operator.
  - (i) We cannot perform arithmetic operations on character variables.
  - (j) The ASCII character set consists of 128 distinct characters.
  - (k) In the ASCII collating sequence, the uppercase letters precede lowercase letters.
  - (l) In C, it is illegal to mix character data with numeric data in arithmetic operations.
  - (m) The function call **strcpy(s2, s1)**; copies string s2 into string s1.
  - (n) The function call **strcmp("abc", "ABC")**; returns a positive number.
  - (o) We can assign a character constant or a character variable to an **int** type variable.



## 6.32 Computer Programming

2. Fill in the blanks in the following statements.

- We can use the conversion specification \_\_\_\_\_ in **scanf** to read a line of text.
- The function \_\_\_\_\_ does not require any conversion specification to read a string from the keyboard.
- The **printf** may be replaced by \_\_\_\_\_ function for printing strings.
- The function **strncat** has \_\_\_\_\_ parameters.
- The function \_\_\_\_\_ is used to determine the length of a string.
- We can initialize a string using the string manipulation function \_\_\_\_\_.
- To use the function **atoi** in a program, we must include the header file \_\_\_\_\_.
- The \_\_\_\_\_ string manipulation function determines if a character is contained in a string.
- The function call **strcat (s2, s1);** appends \_\_\_\_\_ to \_\_\_\_\_.
- The function \_\_\_\_\_ is used to sort the strings in alphabetical order.

3. Describe the limitations of using **getchar** and **scanf** functions for reading strings.

4. Character strings in C are automatically terminated by the *null* character. Explain how this feature helps in string manipulations.

5. Strings can be assigned values as follows:

- During type declaration  

```
char string[] = {"....."};
```
- Using **strcpy** function  

```
strcpy(string, ".....");
```
- Reading using **scanf** function  

```
scanf("%s", string);
```
- Reading using **gets** function

```
gets(string);
```

Compare them critically and describe situations where one is superior to the others.

6. Assuming the variable **string** contains the value "The sky is the limit.", determine what output of the following program segments will be.

- ```
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')
```

7. Which of the following statements will correctly store the concatenation of strings **s1** and **s2** in string **s3**?

- ```
s3 = strcat (s1, s2);
```
- ```
strcat (s1, s2, s3);
```
- ```
strcat (s3, s2, s1);
```
- ```
strcpy (s3, strcat (s1, s2));
```
- ```
strcmp (s3, strcat (s1, s2));
```
- ```
strcpy (strcat (s1, s2), s3);
```

8. What will be the output of the following statement?

```
printf ("%d", strcmp ("push",  
"pull"));
```

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));
```

10. What will be the output of the following segment?

```
char s1[ ] = "Kolkotta" ;  
char s2[ ] = "Pune" ;  
strcpy (s1, s2) ;  
printf("%s", s1) ;
```


11. What will be the output of the following segment?

```
char s1[ ] = "NEW DELHI" ;
char s2[ ] = "BANGALORE" ;
strncpy (s1, s2, 3) ;
printf("%s", s1) ;
```

12. What will be the output of the following code?

```
char s1[ ] = "Jabalpur" ;
char s2[ ] = "Jaipur" ;
printf(strncmp(s1, s2, 2) );
```

13. What will be the output of the following code?

```
char s1[ ] = "ANIL KUMAR GUPTA";
char s2[ ] = "KUMAR";
printf (strstr (s1, s2) );
```

14. Compare the working of the following functions:

- (a) strcpy and strncpy;
- (b) strcat and strncat; and
- (c) strcmp and strncmp.



Debugging Exercises

1. Find errors, if any, in the following code segments:

- (a)

```
char str[10]
strcpy(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);
```



Programming Exercises

- Write a program, which reads your name from the keyboard and outputs a list of ASCII codes, which represent your name.
- 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.
- 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 nth character.

- Write a program which will read a text and count all occurrences of a particular word.
- 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.
- 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."
- A Maruti car dealer maintains a record of sales of various vehicles in the following form:

6.34 Computer Programming

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).

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).
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”.
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 list:

- (a) Alphabetical list of names, roll numbers and marks obtained.
 - (b) List sorted on roll numbers.
 - (c) List sorted on marks (rank-wise list)
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.
 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()**.
 13. Write a program that will copy m consecutive characters from a string s1 beginning at position n into another string s2.
 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.
 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

7

Pointers

CHAPTER OUTLINE

7.1 Introduction	7.7 Chain of Pointers	7.15 Array of Pointers
7.2 Understanding Pointers	7.8 Pointer Expressions	7.16 Dynamic Memory Allocation
7.3 Initialization of Pointer Variables	7.9 Pointer Increments and Scale Factor	7.17 Allocating a Block of Memory: Malloc
7.4 Declaring Pointer Variables	7.10 Pointers as Function Arguments	7.18 Allocating Multiple Blocks of Memory: Calloc
7.5 Accessing the Address of a Variable	7.11 Functions Returning Pointers	7.19 Releasing the Used Space: Free
7.6 Accessing a Variable Through Its Pointer	7.12 Pointers to Functions	7.20 Case Study
	7.13 Pointers and Arrays	
	7.14 Pointers and Character Strings	

7.1 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.

7.2 UNDERSTANDING POINTERS

The computer's memory is a sequential collection of *storage cells* as shown in Fig. 7.1. Each cell, commonly known as a *byte*, has a number called *address* associated with it. Typically, the addresses are numbered 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.

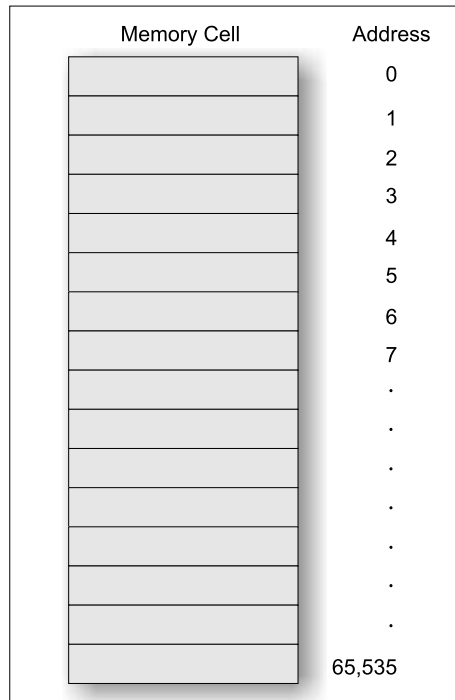


Fig. 7.1 Memory organisation

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. 7.2. (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. 7.3. The address of **p** is 5048.

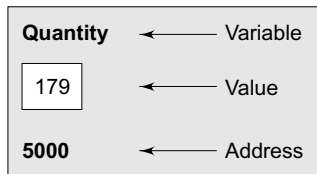


Fig. 7.2 Representation of a variable

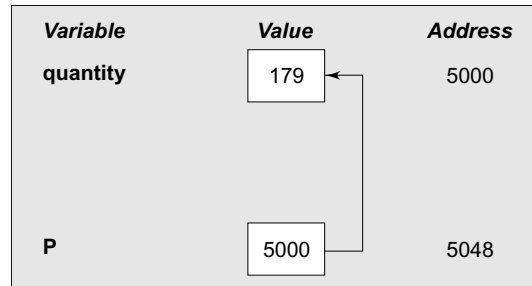
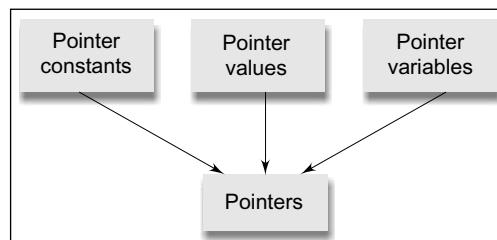


Fig. 7.3 Pointer variable

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**.)

7.2.1 Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated:



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*.

7.3 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.

7.4 Computer Programming

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 */
```

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.

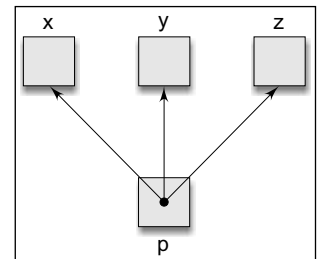
We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued

```
int *p = NULL;  
int *p = 0;
```

7.3.1 Pointer Flexibility

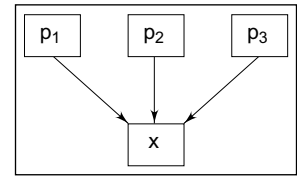
Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
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 */
```

7.4 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,

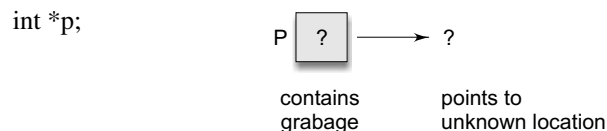
```
int *p; /* integer pointer */
```

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:



7.4.1 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:

7.6 Computer Programming

```
int*      p;           /* style 1 */
int       *p;          /* style 2 */
int       * p;         /* style 3 */
```

However, the style 2 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;
```

7.5 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 use 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 7.1 Write a program to print the address of a variable along with its value.

The program shown in Fig. 7.4, 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.

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. 7.4 Accessing the address of a variable

Example 7.2 Write a C program to demonstrate working of pointers.

Program

```
#include <stdio.h>
int main()
{
    int* pc;
    /*'variable' is syntax to declare a pointer*/
    int c;
    c=22;
    clrscr();
    printf("Address of c:%d\n",&c);
    printf("Value of c:%d\n\n",c);
    pc=&c;
    printf("Address of pointer pc:%d\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);
    c=11;
    printf("Address of pointer pc:%d\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);
    *pc=2;
```


7.8 Computer Programming

```
    printf("Address of c:%d\n",&c);
    printf("Value of c:%d\n\n",c);
    getch();
    return 0;
}
```

Output

```
Address of c:-699793244
Value of c:22
Address of pointer pc:-699793244
Content of pointer pc:22
Address of pointer pc:-699793244
Content of pointer pc:11
Address of c:-699793244
Value of c:2
```

Fig. 7.5 Demonstrating the working of pointers

7.6 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 7.3 illustrates the distinction between pointer value and the value it points to.

Example 7.3 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. 7.6. 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. 7.6 Accessing a variable through its pointer

The actions performed by the program are illustrated in Fig. 7.7. 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,

7.10 Computer Programming

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*.

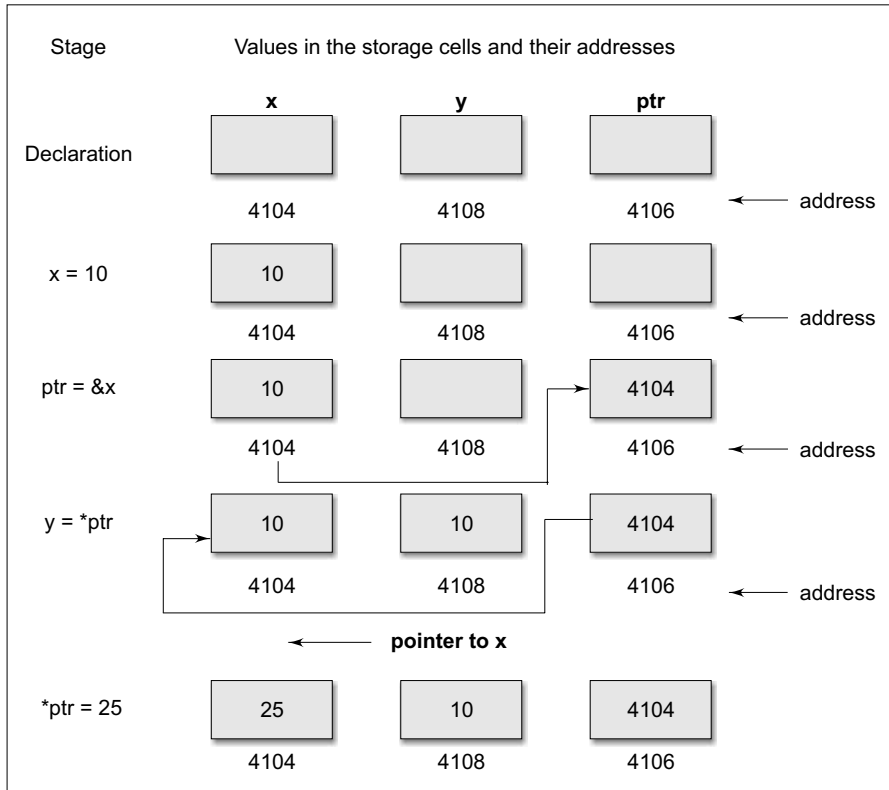
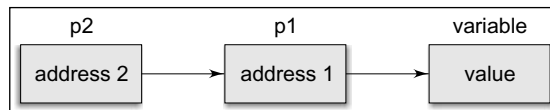


Fig. 7.7 Illustration of pointer assignments

7.7 CHAIN OF POINTERS

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown.



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.

7.8 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 7.4 Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig. 7.8 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

7.12 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. 7.8 Evaluation of pointer expressions

7.9 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.)

7.9.1 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.
5. 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).

7.10 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)
```


7.14 Computer Programming

```
{  
    *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. See Appendix 1.

Example 7.5 Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig. 7.9 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. 7.9 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 can also use this technique in designing user-defined functions discussed in Chapter 9. Let us consider the problem sorting an array of integers.

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. 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.

7.16 Computer Programming

Each character, after it has been copied, is compared with ‘\0’ and therefore, copying is terminated as soon as the ‘\0’ is copied.

Example 7.6 The program of Fig. 7.10 shows how to calculate the sum of two numbers which are passed as arguments using the call by reference method.

Program

```
#include<stdio.h>
#include<conio.h>
void swap (int *p, *q);
main()
{
    int x=0;
    int y=20;
    clrscr();
    printf("\nValue of X and Y before swapping are X=%d and Y=%d", x,y);
    swap(&x, &y);
    printf("\n\nValue of X and Y after swapping are X=%d and Y=%d", x,y);
    getch();
}

void swap(int *p, int *q)//Value of x and y are transferred using call by reference
{
    int r;
    r=*p;
    *p=*q;
    *q=r;
}
```

Output

```
Value of X and Y before swapping are X=10 and Y=20
Value of X and Y after swapping are X=20 and Y=10
```

Fig. 7.10 Program to pass the arguments using call by reference method

7.11 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.

7.12 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 *gptra();
```

would declare **gptra** as a function returning a pointer to *type*.

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 7.7 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. 7.11. 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)();
```


7.18 Computer Programming

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) = 2*x*x-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. 7.11 Use of pointers to functions

7.12.1 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.

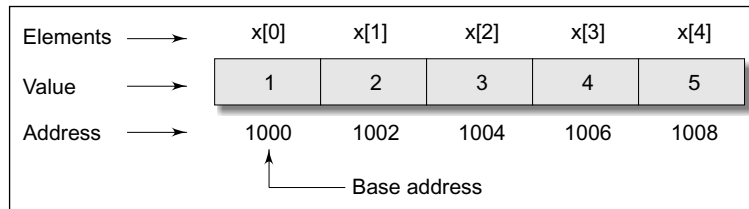
7.13 POINTERS AND ARRAYS

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:

7.20 Computer Programming



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:

$$\begin{aligned} p &= \&x[0] (= 1000) \\ p+1 &= \&x[1] (= 1002) \\ p+2 &= \&x[2] (= 1004) \\ p+3 &= \&x[3] (= 1006) \\ p+4 &= \&x[4] (= 1008) \end{aligned}$$

You may notice that the address of an element is calculated using its index and the scale factor of the data type. For instance,

$$\begin{aligned} \text{address of } x[3] &= \text{base address} + (3 \times \text{scale factor of int}) \\ &= 1000 + (3 \times 2) = 1006 \end{aligned}$$

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that ***(p+3)** gives the value of **x[3]**. The pointer accessing method is much faster than array indexing.

Example 7.8 Illustrates the use of pointer accessing method.

Example 7.8 Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig. 7.12 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 **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. 7.12 Accessing one-dimensional array elements using the pointer

It is possible to avoid the loop control variable **i** as shown:

```

.....
p = x;
while(p <= &x[4])
{
    sum += *p;
    p++;
}
.....

```

Here, we compare the pointer **p** with the address of the last element to determine when the array has been traversed.

Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array **x**, the expression

$*(x+i)$ or $*(p+i)$

represents the element **x[i]**. Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

$*(*(a+i)+j)$ or $*(*(p+i)j)$

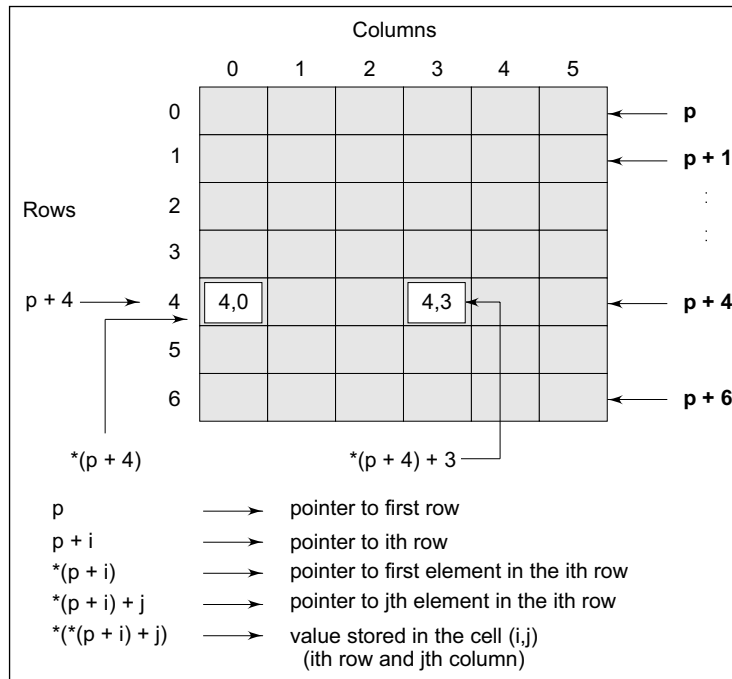
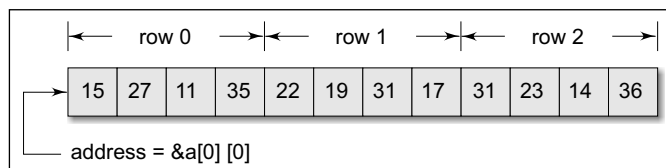


Fig. 7.13 *Pointers to two-dimensional arrays*

Figure 7.13 illustrates how this expression represents the element **a[i][j]**. The base address of the array **a** is **&a[0][0]** and starting at this address, the compiler allocates contiguous space for all the elements *row-wise*. That is, the first element of the second row is placed immediately after the last element of the first row, and so on. Suppose we declare an array **a** as follows:

```
int a[3][4] = { {15,27,11,35},
                 {22,19,31,17},
                 {31,23,14,36}
               };
```

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 $*(p+4 \times i+j)$

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.

7.14 POINTERS AND CHARACTER STRINGS

We have seen in Chapter 8 that strings are treated like character arrays and therefore, they are declared and initialized as follows:

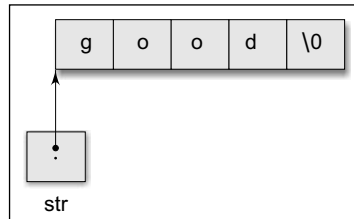
```
char str [5] = "good";
```

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 the Example 7.9.

Example 7.9 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. 7.14. 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')
```

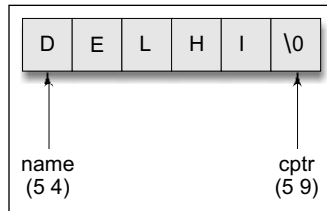
is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr - name;
```


7.24 Computer Programming

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. 7.14 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";
```

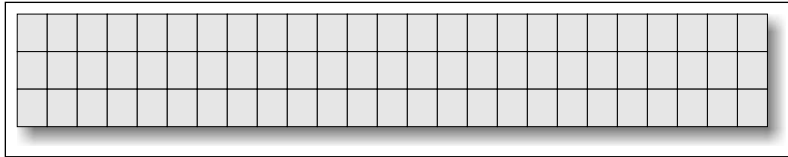
do not work.

7.15 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.



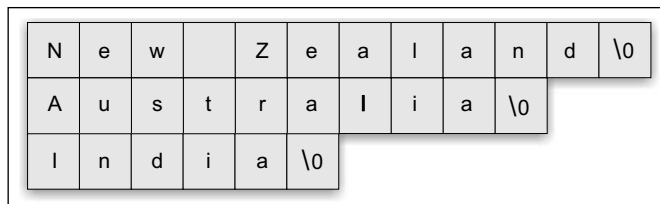
We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. For example,

```
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:

```
name [0] ———> New Zealand
name [1] ———> Australia
name [2] ———> India
```

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown



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.

7.16 DYNAMIC MEMORY ALLOCATION

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.

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 7.1. These functions help us build complex application programs that use the available memory intelligently.

7.16.1 Memory Allocation Process

Before we discuss these functions, let us look at the memory allocation process associated with a C program. Figure 7.15 shows the conceptual view of storage of a C program in memory.

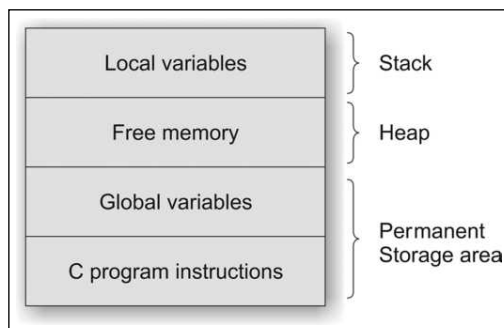


Fig. 7.15 Storage of a C program

TABLE 7.1 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).

7.17 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*.

Example:

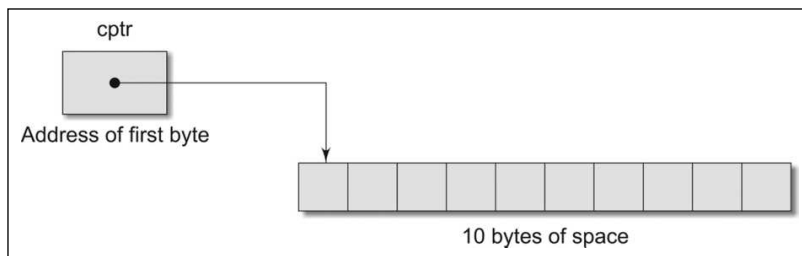
```
x = (int *) malloc (100 *sizeof(int));
```

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. 7.17.

Example 7.11 Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig. 7.16. 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. 7.16 Memory allocation with malloc

Example 7.12 Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

Program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    clrscr();
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));
    /*the malloc function is used to allocate memory on the heap*/
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    getch();
    return 0;
}
```

Output

```
Enter number of elements: 4
Enter elements of array: 1
2
3
4
Sum=10
```

Fig. 7.17 Sum of n elements after allocating memory dynamically using malloc() function

7.18 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);
```


7.30 Computer Programming

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);
}
```

Example 7.13 Write a program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using **calloc()** function.

Program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    clrscr();
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    /*it is same as malloc but calloc sets allocated memory to zero*/
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
```



```

        for(i=0;i<n;++i)
        {
            scanf("%d",ptr+i);
            sum+=*(ptr+i);
        }
        printf("Sum=%d",sum);
        free(ptr);
    getch();
    return 0;
}

```

Output

```

Enter number of elements: 4
Enter elements of array: 1
2
3
4
Sum=10

```

Fig. 7.18 Finding sum of n elements after allocating memory dynamically using `calloc()` function

Example 7.14 Write a program to swap two numbers using: (i) call by value method and (ii) call by reference method.

(i) Call by Value Method

```

Program
#include<stdio.h>
#include<conio.h>
main()
{
    int a, ,b, c;
    printf ("Please enter the value of a");
    scanf ("%d",&a);
    printf ("Please enter the value of b");
    scanf ("%d",&b);
    printf ("The value of a and b before swapping is");
    printf ("The value of a is=%d", a);
    printf ("The value of b is=%d", b);

    /*Swapping*/
    c=b;
    a=b;
    b=c;

    /*Disply of Output*/
    printf ("The value of a and b after swapping is");
    printf ("The value of a is=%d", a);
    printf ("The value of b is=%d", b);
}

```


7.32 Computer Programming

```
getch();  
}
```

Output

```
Please enter the value of a: 20  
Please enter the value of b: 10  
The value of a and b before swapping is:  
The value of a is: 20  
The value of b is: 10  
The value of a and b after swapping is:  
The value of a is: 10  
The value of b is: 20
```

(ii) Call by Reference Method

Program

```
#include<stdio.h>  
#include(conio.h)  
main ()  
{  
    int a=20, b=10;  
    void swap ( int *a, int *b); // Function Prototype  
    swap ( &a, &b); // Function Call  
    /*Disply of Output*/  
    printf ("The value of a and b after swapping is");  
    printf ("The value of a is=%d", a);  
    printf ("The value of b is=%d", b);  
    getch();  
}  
  
Void swap ( int *a, int *b)  
{  
    /*Swapping*/  
    int c;  
    c=*b;  
    *a=*b;  
    *b=c;  
}
```

Output

```
The value of a and b after swapping is:  
The value of a is: 10  
The value of b is: 20
```


7.19 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.

7.20 CASE STUDIES

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. 7.19 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.

7.34 Computer Programming

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 [ ],
               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];
    }
}

```


7.36 Computer Programming

```
        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	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. 7.19 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. 7.20 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**.

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. 7.20 Use of structure pointers as function parameters



Key Terms

- **Memory:** This is a sequential collection of storage cells with each cell having an address value associated with it.
- **Pointer:** It is used to store the memory address as value.
- **Pointer variable:** It is a variable that stores the memory address of another variable.
- **Call by reference:** It is the process of calling a function using pointers to pass the addresses of variables.
- **Call by value:** It is the process of passing the actual value of variables.



Just Remember

1. Only an address of a variable can be stored in a pointer variable.
2. Do not store the address of a variable of one type into a pointer variable of another type.
3. The value of a variable cannot be assigned to a pointer variable.
4. 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.
5. Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.
6. 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.
7. It is an error to assign a numeric constant to a pointer variable.
8. It is an error to assign the address of a variable to a variable of any basic data types.
9. 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.
10. Be careful while using indirection operator with pointer variables. A simple pointer uses single indirection operator (*ptr) while a pointer to a pointer uses additional indirection operator symbol (**ptr).
11. 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.

12. 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.
13. When we pass a parameter by address, the corresponding formal parameter must be a pointer variable.
14. 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).
15. When using memory allocation functions malloc and calloc, test for a NULL pointer return value. Print appropriate message if the memory allocation fails.
16. Never call memory allocation functions with a zero size.
17. Release the dynamically allocated memory when it is no longer required to avoid any possible "memory leak".
18. Using a pointer after its memory has been released is an error.
19. It is an error to assign the return value from malloc or calloc to anything other than a pointer.



Multiple Choice Questions

1. Which of the following statements does not hold true for pointers?
 - (a) Pointers help in dynamic memory management.
 - (b) Pointers are more efficient in handling arrays and data tables.
 - (c) Pointers can return only single value from a function.
 - (d) The use of pointers helps save data storage space in memory.
2. Which of the following operators is used to access the value of the variable using the pointer?
 - (a) % (b) & (c) ^ (d) *
3. What will the output of the following piece of code be?


```
int main()
{
    int x=100;
    *p=&x;
    printf("%d",(int)*p);
    return 0;
}
```

 - (a) 100
 - (b) Compile time error
 - (c) Undefined behavior
 - (d) Run time error
4. What does the following statement represent?


```
(void*)0
```

 - (a) A void pointer (b) A null pointer
 - (c) Error (d) Invalid statement
5. Which of the following operators is used to access data members of the structure through the pointer variable in a case where the variable is a pointer to the structure?
 - (a) & (b) * (c) % (d) ->
6. What is the data type of the address stored in a pointer variable?
 - (a) float (b) integer
 - (c) character (d) array
7. Identify the correct syntax which can be used to send an array as a parameter to function.
 - (a) Func(&arr); (b) Func(*array);
 - (c) Func(array); (d) Func(array[size]);
8. What will the output of the following piece of code be?


```
void main()
{
    int a = 2;
    int *p = &a;
    int **q = &p;
    printf("%d%d%d\n", a, *p,
    **q);
}
```

 - (a) 2 2 0 (b) 2 2 2
 - (c) 2 2 junk value (d) Run time error
9. How is a pointer to a function declared?
 - (a) Type (*fptr); (b) Type*(fptr)();
 - (c) Type(*fptr) (); (d) Type(fptr*);

7.40 Computer Programming

10. Which of the following is a constructing data type used to collect different data types?
(a) String (b) Structures
(c) Char (d) Union
11. Which of the following cannot be a member of a structure?
(a) Another structure
(b) Array
(c) Function
(d) Pointer arrays
12. Which of the following functions is used to release allocated memory which is no longer required?
(a) dropmem() (b) free()
(c) release() (d) dealloc()
13. Which of the following functions allocates request size of bytes and returns a pointer to the first byte of allocated space?
(a) malloc (b) calloc
(c) free (d) realloc
14. The machine instructions and data are physically placed onto the main memory by:
(a) Linker (b) Loader
(c) Code generator (d) Interpreter
15. Which of the following data structures is used by malloc() for random memory allocation?
(a) Tree (b) Stack
(c) Heap (d) Queue
16. Which of the following header files should be included in order to use functions like Malloc() and calloc()?
(a) stdio.h (b) stdlib.h
(c) string.h (d) memory.h

Answers

- | | | | | |
|---------|---------|---------|---------|---------|
| 1. (c) | 2. (d) | 3. (b) | 4. (b) | 5. (d) |
| 6. (b) | 7. (c) | 8. (b) | 9. (c) | 10. (b) |
| 11. (c) | 12. (b) | 13. (a) | 14. (b) | 15. (c) |
| 16. (b) | | | | |



Review Questions

1. State whether the following statements are *true* or *false*.
 - (a) Pointer constants are the addresses of memory locations.
 - (b) The underlying type of a pointer variable is void.
 - (c) Pointer variables are declared using the address operator.
 - (d) It is possible to cast a pointer to float as a pointer to integer.
 - (e) Pointers to pointers is a term used to describe pointers whose contents are the address of another pointer.
 - (f) A pointer can never be subtracted from another pointer.
 - (g) An integer can be added to a pointer.
 - (h) Pointers cannot be used as formal parameters in headers to function definitions.
 - (i) When an array is passed as an argument to a function, a pointer is passed.
 - (j) Value of a local variable in a function can be changed by another function.
2. Fill in the blanks in the following statements:
 - (a) A pointer variable contains as its value the _____ of another variable.
 - (b) The _____ operator returns the value of the variable to which its operand points.
 - (c) The _____ operator is used with a pointer to de-reference the address contained in the pointer.
 - (d) The pointer that is declared as _____ cannot be de-referenced.
 - (e) The only integer that can be assigned to a pointer variable is _____.
3. What is a pointer? How can it be initialized?
4. A pointer in C language is

- (a) address of some location
 (b) useful to describe linked list
 (c) can be used to access elements of an array
 (d) All of the above.
5. Explain the effects of the following statements:
- (a) `int a, *b = &a;`
 (b) `int p, *p;`
 (c) `char *s;`
 (d) `a = (float *) &x;`
 (e) `double(*f)();`
6. Distinguish between `(*m)[5]` and `*m[5]`.
7. Given the following declarations:
- ```
int x = 10, y = 10;
int *p1 = &x, *p2 = &y;
```
- What is the value of each of the following expressions?
- (a) `(*p1)++`                      (b) `--(*p2)`  
 (c) `*p1 + (*p2) --`              (d) `++(*p2) - *p1`
8. Describe typical applications of pointers in developing programs.
9. What are the arithmetic operators that are permitted on pointers?
10. What is printed by the following program?
- ```
int m = 100';
int * p1 = &m;
int **p2 = &p1;
printf("%d", **p2);
```
11. 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).`
12. What is the output of the following segment?
- ```
int m[2];
*(m+1) = 100;
```
- ```
*m = *(m+1);
printf("%d", m [0]);
```
13. 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);
```
14. 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);
}
```
15. 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.
16. Do the declarations
`char s [5] ;`
`char *s;`
 represent the same? Explain.
17. 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);`



Debugging Exercises

1. 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.
- (a) `p1 = &m;` (b) `p2 = n;`
 (c) `*p1 = &n;` (d) `p2 = &*&m;`
 (e) `m = p2-p1;` (f) `p1 = &p2;`
 (g) `m = *p1 + *p2++;`

7.42 Computer Programming

2. Find the error, if any, in each of the following statements:

- (a) `int x = 10;`
- (b) `int *y = 10;`
- (c) `int a, *b = &a;`

- (d) `int m;`
`int **x = &m;`

3. What is wrong with the following code?

```
int **p1, *p2;  
p2 = &p1;
```



Programming Exercise

1. Write a program using pointers to read in an array of integers and print its elements in reverse order.

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.

- 3. Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place.
- 4. Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.
- 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.
- 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.

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.

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.)

- 9. Write a function (using a pointer parameter) that reverses the elements of a given array.
- 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.

8

Structures and Unions

CHAPTER OUTLINE

8.1 Introduction	8.5 Arrays of Structures	8.9 Bit Fields
8.2 Defining a Structure	8.6 Structures and Functions	8.10 Typedef
8.3 Declaring Structure Variables	8.7 Pointers and Structures	8.11 Command Line Arguments
8.4 Structure Initialization	8.8 Unions	8.12 Case Study

8.1 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.

8.2 DEFINING A STRUCTURE

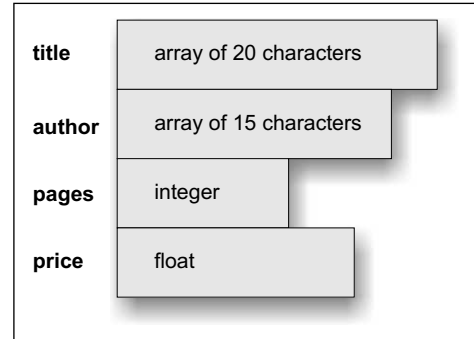
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

8.2 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.

8.3 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;
    flat 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.

8.3.1 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 8.1 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. 8.1. 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. 8.1 Defining and accessing structure members

8.4 STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```

main()
{
    struct
    {
        int weight;
        float height;
    }
}

```


8.6 Computer Programming

```
        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  
    {  
        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.

Example 8.2 Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet. (Note: 12 inches = 1 foot)

Program

```
#include <stdio.h>
/* definition to declare a structure */
struct Distance
{
    int feet;
    float inch;
}
d1,d2,sum;
int main()
{
    clrscr();
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d",&d1.feet);
    printf("Enter inch: ");
    scanf("%f",&d1.inch);
    printf("2nd distance\n");
    printf("Enter feet: ");
    scanf("%d",&d2.feet);
    printf("Enter inch: ");
    scanf("%f",&d2.inch);
    sum.feet=d1.feet+d2.feet;
    sum.inch=d1.inch+d2.inch;
    if (sum.inch>12)
    {
        ++sum.feet;
        sum.inch=sum.inch-12;
    }
    printf("Sum of distances=%d\'-%.1f\'",sum.feet,sum.inch);
```


8.8 Computer Programming

```
    getch();
    return 0;
}
Output
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances= 15'-5.7"
```

Fig. 8.2 Adding two distances

8.5 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. 8.3.

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. 8.3 The array student inside memory

Example 8.3 Write a C program to use structure's member through pointer using malloc() function.

Program

```
#include <stdio.h>
#include<stdlib.h>
struct name
{
    int a;
    float b;
    char c[30];
};
int main()
{
    struct name *ptr;
    int i,n;
    clrscr();
    printf("Enter n: ");
    scanf("%d",&n);
    ptr=(struct name*)malloc(n*sizeof(struct name));
    /*method to use structures*/
    for(i=0;i<n;++i)
    {
        printf("Enter string, integer and floating number respectively:\n");
        scanf("%s%d%f",&(ptr+i)->c,&(ptr+i)->a,&(ptr+i)->b);
    }
    printf("Displaying Infromation:\n");
    for(i=0;i<n;++i)
        printf("%s\t%d\t%.2f\n", (ptr+i)->c, (ptr+i)->a, (ptr+i)->b);
    getch();
    return 0;
}
```


8.10 Computer Programming

Output

```
Enter n: 2
Enter string, integer and floating number  respectively:
Programming
2
3.2
Enter string, integer and floating number respectively:
Structure
6
2.3
Displaying Information
Programming    2      3.20
Structure      6      2.30
```

Fig. 8.4 Usage of structure's member through pointer using malloc() function

Example 8.4 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. 8.5. 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. 8.5 Arrays of structures: Illustration of subscripted structure variables

8.5.1 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 8.5 Rewrite the program of Example 8.4 using an array member to represent the three subjects.

8.12 Computer Programming

The modified program is shown in Fig. 8.6. 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++)
        {
            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. 8.6 Use of subscripted members arrays in structures

8.5.2 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:

```
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
```


8.14 Computer Programming

```
        {
            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
{
    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.

8.6 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 8.6 Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

A program to update an item is shown in Fig. 8.7. 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. 8.7 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.

8.6.1 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");
}

```


8.18 Computer Programming

```
printf("%s",my_struct.fname);
printf("%s",my_struct.lname);
my_struct.age = 63;
show_name(st_ptr); /* pass the pointer */
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);
}
```

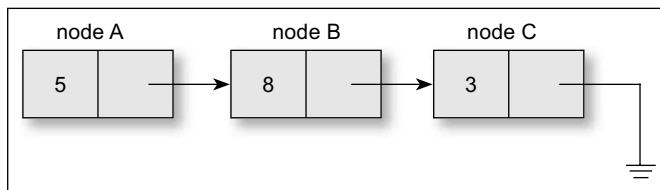
8.6.2 Self Referential Structure

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;
};
```



8.7 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 8.7 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. 8.8. 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)
        {

```


8.20 Computer Programming

```
        printf("%-20s %5d %10.2f\n",
               ptr->name,
               ptr->number,
               ptr->price);
        ptr++;
    }
}
```

Output

```
INPUT
Washing_machine   5      7500
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. 8.8 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.

*ptr->p	Fetches whatever p points to.
*ptr->p++	Increments p after accessing whatever it points to.
(*ptr->p)++	Increments whatever p points to.
*ptr++->p	Increments ptr 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**.

8.8 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 8.9 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

```

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).

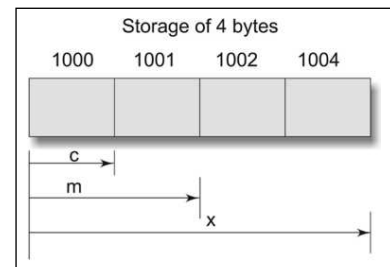


Fig. 8.9 Sharing of a storage locating by union members

8.22 Computer Programming

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.

Example 8.8 Write a C program to store the name and salary of an employee using unions.

Program

```
#include <stdio.h>
/* definition to declare a union */
union job
{
    char name[32];
    float salary;
    int worker_no;
}
u;
int main()
{
    clrscr();
    printf("Enter name:\n");
    scanf("%s",&u.name);
    printf("Enter salary: \n");
    scanf("%f",&u.salary);
    printf("Displaying\nName:%s\n",u.name);
    printf("Salary: %.1f",u.salary);
    getch();
    return 0;
}
```

Output

```
Enter name:
Sam
Enter salary:
24000
Displaying
Name:
Salary: 24000
```

Fig. 8.10 Storing the name and salary of an employee using unions

8.9 BIT FIELDS

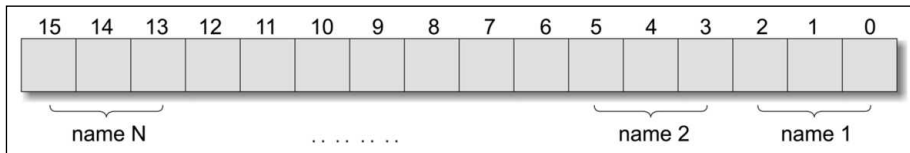
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a 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.

8.24 Computer Programming

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;
    . . . . .
    . . . . .
}
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.'

8.10 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;
```

Example 8.9 Write a program to print the marks obtained by two students using structures.

Program

```
#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();
```


8.26 Computer Programming

```
/*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
```

Fig. 8.11 Marks obtained by two students using structures.

Example 8.10 Write a program that uses a simple structure for storing different students' details.

Program

```
#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

```


8.28 Computer Programming

```
student 2
Name Arun
Roll No. 6
Total Marks 325
```

Fig. 8.12 Simple structure for storing different students' details.

Example 8.11 Write a simple program to demonstrate the process of defining a structure variable and assigning values to its members.

Program

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

```

Fig. 8.13 Defining a structure variable and assigning values to its members

Example 8.12 Write a program to realize the concept of complex numbers using structures and also print the sum of two complex number variables.

Program

```

#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();
}

```


8.30 Computer Programming

```
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
Imaginary Part of Second Number: 4

1.00+(2.00)i + 3.00+(4.00)i = 4.00+(6.00)i
```

Fig. 8.14 *Printing the sum of 2 complex numbers using structures.*

Example 8.13 Write a program to read the marks obtained by a student in three subjects and compute its sum and average.

```
Program
#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

```

Fig. 8.15 Marks obtained by a student in three subjects and compute its sum and average.

Example 8.14 Implement the problem in Example 1 using pointer notation.

Program

```

#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();
}

```


8.32 Computer Programming

Output

Marks obtained by 1st student: 55, 66 and 80

Marks obtained by 2nd student: 60, 85 and 78

Fig. 8.16 Demonstrating structure member initialization using pointers

Example 8.15 Implement the problem in Example 8.12 using pointer notation.

Program

```
#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

Fig. 8.17 Demonstrating operations on individual structure members using pointer notation

Example 8.16 Implement the following student information fields using structures:

Roll No
Name (First, Middle, Last)
DOB (Day, Month, Year)
Course (Elective1, Elective2)

```
Program
#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");
```


8.34 Computer Programming

```
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");
/*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.
elective2);

    getch();
}
```

Output

```
Roll No.: 21
Name: Tulsi K Shanta
Date of Birth (DD MM YYYY): 2 2 1981
Course Electives: Mechanics & Animation
```

Fig. 8.18 Demonstrating nesting of structure.

Example 8.17 Implement the following employee information fields using structures:

Employee ID
Name (First, Middle, Last)
DOJ,
Gross Salary (HRA, BASIC, Special Allowance).

Program

```
#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

```

Fig. 8.19 Implementing employee information fields using structures

Example 8.18 Write a simple program to demonstrate the use of unions.

Program

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

Fig. 8.20 Program to demonstrate the use of unions

Example 8.19 Write a program that uses the size of operator to differentiate between structures and unions.

```

Program
#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();
    /*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

```

Fig. 8.21 Program for differentiating between structure & union

8.11 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

8.38 Computer Programming

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 8.20 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 8.11 shows the use of command line arguments. The command line is

F14_7 TEXT AAAAAA BBBB BB CCCCC DDDDD EEEEE FFFFF GGGGG

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 BBBBBB CCCCCC DDDDDD EEEEE EFFFFFF GGGGG
No. of arguments in Command line = 9

Contents of TEXT file

AAAAAA BBBBBB CCCCCC DDDDDD EEEEE EFFFFFF GGGGGG

C:\C\F12_7.EXE
TEXT
    AAAAAA
      BBBBBB
        CCCCCC
          DDDDDD
            EEEEE
              FFFFFFF
                GGGGGG

```

Fig. 8.22 Use of command line arguments

Example 8.21 Write a simple program to open and close a file; or print an error message in case of unsuccessful operation.

Program

```
/*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 incase 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
```

Fig. 8.23 Opening & closing a file

Example 8.22 Write a program to copy the contents of one file into another.

Program

```
/*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();
}

```

Output

```

File copy operation performed successfully
You can confirm the same by checking the Destination.txt file

```

Fig. 8.24 Copy contents of a file into another

Example 8.23 Write a program to count the number of characters in a file.

Program

```

/*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();

```


8.42 Computer Programming

```
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
```

Fig. 8.25 Counting number of characters in a file

Example 8.24 Write a program to read the contents of one file and write them in reverse order in another file.

```
Program
/*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

```

Fig. 8.26 Reading the contents of one file and writing them in reverse order in another file

Example 8.25 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()
{

```


8.44 Computer Programming

```
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
```

Fig. 8.27 Demonstrating the use of *getw* and *putw*

Example 8.26 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);
    }
    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 com-
        puted and copied to the target file. You can open the Target.txt file for vali-
        dation");
        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

Fig. 8.28 Reading integer from a file & writing square of those integers in another file

Example 8.27 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
```

Fig. 8.29 Reading list of integers from two different files & merging & storing then in a single file

Example 8.28 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/ location*/
#include <stdio.h>
#include <conio.h>

void main(int argc, char *argv[])
{
    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...

```

Fig. 8.30 Open a file using command line arguments and display its contents

8.11.1 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 8.29 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

Fig. 8.31 Copying the contents of one file into another using command line arguments

8.12 CASE STUDY

1. 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. 8.12. 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.

Program

```
#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);
    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");
}

```


8.52 Computer Programming

```
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. 8.32 Program of bookshop inventory



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.



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.
- 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.

25. The size of a structure variable can be computed by adding the storage space of each of its member variables.
26. 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.
27. The uninitialized structure members are assigned the value zero for integer and float and '\0' for characters and strings.
28. Nesting of structures is possible by declaring a structure inside another structure.



Multiple Choice Questions

1. What amongst the following determines the size of a union?
 - (a) Sum of the sizes of all its members
 - (b) First member of the union
 - (c) Last member of the union
 - (d) Biggest member of the union
2. Which of the following is a wrong element for compile time initialization of a structure variable?
 - (a) The keyword structure
 - (b) Structure tag name
 - (c) Logical operator &&
 - (d) The name of the variable to be declared
3. Which of the following statements is false?
 - (a) The members of a union share the same memory area.
 - (b) At a particular instant of time, only one of the members of a union can be assigned a value.
 - (c) The compiler keeps a track of the kind of information currently stored.
 - (d) The size allocated to a union corresponds to the size of its member that needs the maximum storage.
4. Which of the following makes the statement "A->B" syntactically correct?
 - (a) Both A and B are structures
 - (b) A is a pointer to the structure in which B is a field.
 - (c) A is a structure whereas B is a pointer to the structure.
 - (d) A is a pointer to the structure B.
5. Determine the size of the following union declaration.


```
#include <stdio.h>
union uTemp
{
    double x;
    int y[5];
    char c;
}u;
```

 (Assume that size of double: 8; size of int=4; size of char=1)
 - (a) 8
 - (b) 4
 - (c) 20
 - (d) 29
6. What will the output for the following piece of code be?


```
void main()
{
    struct classroom
    {
        int no;
        char name[20];
    };
    struct classroom s;
    s.no = 8;
    printf("%d", s.no);
}
```

 - (a) Garbage value
 - (b) Runtime error
 - (c) Nothing
 - (d) 8

Answers

- | | | | | |
|--------|--------|--------|--------|--------|
| 1. (d) | 2. (c) | 3. (c) | 4. (b) | 5. (c) |
| 6. (d) | | | | |



Review Questions

1. State whether the following statements are *true* or *false*.
 - (a) A **struct** type in C is a built-in data type.
 - (b) The tag name of a structure is optional.
 - (c) Structures may contain members of only one data type.
 - (d) A structure variable is used to declare a data type containing multiple fields.
 - (e) It is legal to copy a content of a structure variable to another structure variable of the same type.
 - (f) Structures are always passed to functions by pointers.
 - (g) Pointers can be used to access the members of structure variables.
 - (h) We can perform mathematical operations on structure variables that contain only numeric type members.
 - (i) The keyword **typedef** is used to define a new data type.
 - (j) In accessing a member of a structure using a pointer *p*, the following two are equivalent:
 $(*p).member_name$ and $p \rightarrow member_name$
 - (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.
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 \rightarrow requires the use of a _____ to access the members of a structure.
 - (e) The variables declared in a structure definition are called its _____.
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";`
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. 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];`

8.56 Computer Programming

- (b) `struct items[10] item_bank;`
- (c) `struct item_bank items (10);`
- (d) `struct item_bank items [10];`
- (e) `struct items item_bank [10];`

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];`
7. How does a structure differ from an array?
8. Explain the meaning and purpose of the following:
- (a) Template
 - (b) **struct** keyword
 - (c) **typedef** keyword
 - (d) **sizeof** operator
 - (e) Tag name
9. Explain what is wrong in the following structure declaration:

```
struct
{
    int number;
    float price;
}
main( )
{
    . . . . .
    . . . . .
}
```

10. When do we use the following?
- (a) Unions
 - (b) Bit fields
 - (c) The **sizeof** operator

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.

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) ...`

13. Describe with examples, the different ways of assigning values to structure members.
14. State the rules for initializing structures.
15. What is a 'slack byte'? How does it affect the implementation of structures?
16. Describe three different approaches that can be used to pass structures as function arguments.
17. What are the important points to be considered when implementing bit-fields in structures?
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.

19. What is the error in the following program?

```
typedef struct product
{
    char name [ 10 ];
    float price ;
} PRODUCT products [ 10 ];
```

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);
}
```



Debugging Exercises

1. Identify the error in the following code, if any:

```
#include<stdio.h>

void main()
{
    int a;
    int *ptr;
    a=50;
    ptr=&a;

    printf("address of a = %u\tvalue
of a = %d\n",ptr,**&ptr);
    printf("address of ptr = %u\
tvalue of ptr = %u",&ptr,ptr);
}
```

2. Identify the error in the following code, if any:

```
#include<stdio.h>

void main()
{
    int i;
    char *name[3] =
    {
        "New Zealand",
        "Australia",
```

```
        "India"
    };
    for(i=0;i<3;i++)
        printf("Address of name [%d] =
        %u, which points to the string
        \"%s\\n\",i,name[i],*name[i]);
}
```

3. Identify the error in the following code, if any:

```
#include<stdio.h>

void main()
{
    int i;
    int num[3] = {22,34,88};
    void print(int *num);
    print(num);
}

void print(int *num1)
{
    int i;
    printf("The values stored in the
    array are: ");
    for(i=0;i<3;i++)
        printf("%d\t",*num1[i]);
}
```




Programming Exercise

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

2. Modify the above program such that a function is used to input values to the members and another function to display the time.
3. Design a function **update** that would accept the data structure designed in Exercise 8.1 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.)
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. Design a function **update** that accepts the **date** structure designed in Exercise 8.4 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
6. Modify the input function used in Exercise 8.4 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.
 7. Add a function called **nextdate** to the program designed in Exercise 8.4 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.

8. Use the **date** structure defined in Exercise 8.4 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
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,)
10. Add a function to the program of Exercise 8.9 that accepts two vectors as input parameters and return the addition of two vectors.
 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.
 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
 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
 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.
 15. Design a structure **student_record** to contain name, date of birth and total marks obtained. Use the **date** structure designed in Exercise 8.4 to represent the date of birth.
Develop a program to read data for 10 students in a class and list them rank-wise.

9

Data Files

CHAPTER OUTLINE

9.1 Introduction

9.2 Defining and Opening a File

9.3 Closing a File

9.4 Input/Output Operations on Files

9.5 Error Handling during I/O Operations

9.6 Random Access to Files

9.1 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 9.1.

TABLE 9.1 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.

9.2 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 the following:

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:

```
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.

9.3 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");
```


9.4 Computer Programming

```
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.

9.4 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 9.1.

9.4.1 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 9.1 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. 9.1. 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 *fp1;
```



```

        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);
        /* 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. 9.1 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.

Example 9.2 Program in Fig. 9.2 takes an input from the user and write it in a file. If file doesn't exists then also create a file with same name.

```
Program
#include <stdio.h>
int main()
{
    int n;
    FILE *fptr;
    /*FILE is used to declare a variable of file type*/
    clrscr();
    fptr=fopen("input.txt","w");
    /*fopen() function is used to open a file*/
    if(fptr==NULL)
    {
        printf("Error!");
    }
    printf("Enter n: ");
    scanf("%d",&n);
    fprintf(fptr,"%d",n);
    printf("Data has been successfully written...");
    fclose(fptr);
    /*fclose() function is used to close a file*/
    getch();
    return 0;
}
Output
Enter n: 45

Data has been successfully written...
```

Fig. 9.2 Illustration of create/write input in a file

Example 9.3 Write a C program to delete a specific line from a text file.

```
Program
#include <stdio.h>
int main()
{
    FILE *fileptr1, *fileptr2;
    char filename[40];
    char ch;
    int delete_line, temp = 1;
    clrscr();
    printf("Enter file name: ");
    scanf("%s", filename);
    fileptr1 = fopen(filename, "r");
    ch = getc(fileptr1);
    while (ch != EOF)
        /*condition to use the end of a file*/
        {
            printf("%c", ch);
```



```

        ch = getc(fileptr1);
    }
    rewind(fileptr1);
    printf(" \n Enter line number of the line to be deleted:");
    scanf("%d", &delete_line);
    fileptr2 = fopen("replica.c", "w");
    ch = getc(fileptr1);
    while (ch != EOF)
    {
        ch = getc(fileptr1);
        /*getc()function is used to read a character from a file*/
        if (ch == '\n')
            temp++;
            if (temp != delete_line)
            {
                putc(ch, fileptr2);
            }
    }
    fclose(fileptr1);
    fclose(fileptr2);
    remove(filename);
    /*remove()function is used to remove an element from a file*/
    rename("replica.c", filename);
    printf("\n The contents of file after being modified are as follows:\n");
    fileptr1 = fopen(filename, "r");
    ch = getc(fileptr1);
    while (ch != EOF)
    {
        printf("%c", ch);
        ch = getc(fileptr1);
    }
    fclose(fileptr1);
}
getch();
return 0;
}

```

Output

```

Enter file name: input.txt
hi
this
is
file
handling
example
Enter line number of the line to be deleted:2

The contents of file after being modified are as follows:
i
is
file
handling
example

```

Fig. 9.3 Deletion of a specific line from a text file

9.4.2 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 as follows:

```
putw(integer, fp);
getw(fp);
```

Worked-Out Problem 9.2 illustrates the use of **`putw`** and **`getw`** functions.

Example 9.4 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. 9.4. 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. 9.4 Operations on integer data

9.4.3 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 9.5 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. 9.5. 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. 9.5 Operations on mixed data types

9.5 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 the following:

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 9.6 Write a program to illustrate error handling in file operations.

The program shown in Fig. 9.6 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. 9.6 Illustration of error handling in file operations

9.6 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 9.2 illustrate the operations of the **fseek** function:

TABLE 9.2 Operations of **fseek** function

Statement	Meaning
<code>fseek(fp,0L,0);</code>	Go to the beginning. (Similar to <code>rewind</code>)
<code>fseek(fp,0L,1);</code>	Stay at the current position. (Rarely used)
<code>fseek(fp,0L,2);</code>	Go to the end of the file, past the last character of the file.
<code>fseek(fp,m,0);</code>	Move to (m+1)th byte in the file.
<code>fseek(fp,m,1);</code>	Go forward by m bytes.
<code>fseek(fp,-m,1);</code>	Go backward by m bytes from the current position.
<code>fseek(fp,-m,2);</code>	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 9.7 Write a program that uses the functions **ftell** and **fseek**.

A program employing **ftell** and **fseek** functions is shown in Fig. 9.7. 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()
{

```


9.16 Computer Programming

```
FILE *fp;
long n;
char c;
fp = fopen("RANDOM", "w");
while((c = getchar()) != EOF)
    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. 9.7 Illustration of *fseek* and *ftell* functions

Example 9.8 Write a program to append additional items to the file INVENTORY created in Program 9.3 and print the total contents of the file.

The program is shown in Fig. 9.8. 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");
```


9.18 Computer Programming

```
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. 9.8 Adding items to an existing file

Example 9.9 Write a C program to reverse the first *n* character in a file. The file name and the value of *n* are specified on the command line. Incorporate validation of arguments, that is, the program should check that the number of arguments passed and the value of *n* that are meaningful.

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 souce 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*/
            j=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

```

Fig. 9.9 Program to reverse *n* characters in a file



Key Terms

- **Filename:** It is a string of characters that make up a valid filename for the operating system.
- **fseek:** It is a function that sets the position to a desired point in the file.
- **ftell:** It is a function that returns the current position in the file.
- **Rewind:** It is a function that sets the position to the beginning of the file.



Just Remember

1. Do not try to use a file before opening it.
2. Remember, when an existing file is open using 'w' mode, the contents of file are deleted.
3. When a file is used for both reading and writing, we must open it in 'w+' mode.
4. It is an error to omit the file pointer when using a file function.
5. It is an error to open a file for reading when it does not exist.
6. It is an error to access a file with its name rather than its file pointer.
7. It is a good practice to close all files before terminating a program.
8. EOF is integer type with a value -1. Therefore, we must use an integer variable to test EOF.
9. It is an error to try to read from a file that is in write mode and vice versa.
10. To avoid I/O errors while working with files, it is a good practice to include error handling code in programs by using functions such as feof and ferror.
11. It is an error to attempt to place the file marker before the first byte of a file.



Multiple Choice Questions

- Which of the following functions reads an integer from a file?
 - putw()
 - fseek()
 - getw()
 - fscanf()
- What does the function ftell() do?
 - Sets the position to the beginning of the file
 - Writes an integer to the file
 - Reads a character from a file
 - Sets the position to a desired point in the file
- What does the following function return?
fopen("name", "r")
 - A pointer to FILE name
 - Values 0 or 1
 - Pointer to a new file
 - Nothing
- When a file is opened in r+ mode, which of the following is possible?
 - Reading
 - Writing
 - Both reading and writing
 - Nothing
- Opening a file in which of the following modes results in the loss of contents of a file?
 - a
 - w
 - a+
 - w+

Answers

- (c)
- (d)
- (a)
- (c)
- (d)



Review Questions

- 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.
- Fill in the blanks in the following statements.
 - The mode _____ is used for opening a file for updating.
 - The function _____ is used to write data to randomly accessed file.
 - The function _____ gives the current position in the file.
 - The function _____ may be used to position a file at the beginning.
- Describe the use and limitations of the functions **getc** and **putc**.
- What is the significance of EOF?
- 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?
- Distinguish between the following functions:
 - getc and getchar
 - printf and fprintf
 - feof and ferror
- How does an append mode differ from a write mode?
- What are the common uses of **rewind** and **ftell** functions?
- Explain the general format of **fseek** function?
- What is the difference between the statements **rewind(fp);** and **fseek(fp,0L,0);**?
- What does the following statement mean?
FILE(*p) (void)
- What does the following statement do?
while ((c = getchar() != EOF)
 putc(c, fl);
- What does the following statement do?
while ((m = getw(fl)) != EOF)
 printf("%5d", m);

9.22 Computer Programming

14. What does the following segment do?

```
    . . . . .
    for (i = 1; i <= 5; i++ )
    {
        fscanf(stdin, "%s", name);
        fprintf(fp, "%s", name);
    }
    . . . . .
```

15. What is the purpose of the following functions?
(a) feof () (b) ferror ()

16. Give examples of using **feof** and **ferror** in a program.
17. Can we read from a file and write to the same file without resetting the file pointer? If not, why?
18. When do we use the following functions?
(a) free () (b) rewind ()
19. Describe an algorithm that will append the contents of one file to the end of another file.



Debugging Exercise

1. Find error, if any, in the following statements:

```
FILE fptr;
fptr = fopen ("data", "a+");
```



Programming Exercise

- Write a program to copy the contents of one file into another.
- 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.
- Write a program that compares two files and returns 0 if they are equal and 1 if they are not.
- Write a program that appends one file at the end of another.
- Write a program that reads a file containing integers and appends at its end the sum of all the integers.
- 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.

- 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.

- 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.
- Write a program to read the file created in Exercise 9.8 and compute and print the total value of all the five products.
- Rewrite the program developed in Exercise 9.8 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.

1

C99/C11 Features

C99 Features

A.1 INTRODUCTION

C, as developed and standardized by ANSI and ISO, is a powerful, flexible, portable, and elegant language. Due to its suitability for both systems and applications programming, it has become an industry-standard, general-purpose language to-day.

The standardization committee working on C language has been trying to examine each element of the language critically and see any change or enhancement is necessary in order to continue to maintain its lead over other competing languages. The committee also interacted with many user groups and elicited suggestions on improvements that are required from the point-of-view of users. The result was the new version of C, called C99.

The C99 standard incorporates enhancements and new features that are desirable for any modern computer language. Although it has borrowed some features from C++ (a progeny of C) and modified a few constructs, it retains almost all the features of ANSI C and thus continues to be a true C language.

In this appendix, we will highlight the important changes and new features added to C by the 1999 standard.

A.2 KEYWORDS

ANSI C has defined 32 keywords. C99 has added five more keywords. They are as follows:

_Bool
_Complex
_Imaginary
inline
restrict

Addition of these keywords is perhaps the most significant feature of C99. The use of these keywords are highlighted later in this appendix.

A.3 COMMENT

C99 adds what is known as the single-line comment, a feature borrowed from C++. Single-line comments begin with // (two back slashes) and end at the end of the line. Examples:

```
                                // A comment line
if (x > y)                       // Testing
    printf(.....);             // Printing
int m;                           // Declaration
```

Single-line comments are useful when brief, line-by-line comments are needed.

A.4 DATA TYPES

C defines five basic data types, namely, **char**, **int**, **float**, **double**, and **void**. C99 adds three new built-in data types. They are as follows:

_Bool
_Complex
_Imaginary

C99 also allows **long** to modify **long** thus creating two more modified data types, namely, **long long int** and **unsigned long long int**.

_Bool Type

_Bool is an integer type which can hold the values 1 and 0. Example:

```
_Bool x, y;
x = 1;
y = 0;
```

We know that relational and logical expressions return 0 for false and 1 for true. These values can be stored in **_Bool** type variables. For example,

```
_Bool b = m > n;
```

The variable **b** is assigned 1 if **m** is greater than **n**, otherwise 0.

_Complex and _Imaginary Types

C99 adds two keywords **_Complex** and **_Imaginary** to provide support for complex arithmetic that is necessary for numerical programming. The following complex types are supported:

float_Complex	float_Imaginary
double_Complex	double_Imaginary
long double_Complex	long double_Imaginary

The long long Types

The **long long int** has range of at least $-(2^{63}-1)$ to $2^{63}-1$. Similarly, the **unsigned long long int** has a range of 0 to $2^{64}-1$.

A.5 DECLARATION OF VARIABLES

In C, we know that all the variables must be declared at the beginning of a block or function before any executable statements. However, C99 allow us to declare a variable at any point, just before its use. For example, the following code is legal in C99.

```
main()
{
    int m;
    m = 100;
    . . . . .
    . . . . .
    int n;                /* Legal in C99*/
    n = 200;
    . . . . .
}
```


C99 extends this concept to the declaration of control variables in **for** loops. That is, C99 permits declaration of one or more variables within the initialization part of the loop. For example, the following code is legal.

```
main()
{
    . . . . .
    . . . . .
    for (int i = 0; i<5; i++)
    {
        . . . . .
        . . . . .
    }
    . . . . .
    . . . . .
}
```

A variable declared inside a **for** loop is local to that loop only. The value of the variable is lost, once the loop ends. (This concept is again borrowed from C++.)

A.6 I/O FORMATS

In order to handle the new data types with **long long** specification, C99 adds a new format modifier **ll** to both **scanf()** and **printf()** format specifications. Examples: **%lld**, **%llu**, **%lli**, **%llo**, and **%llx**.

Similarly, C99 adds **hh** modifier to d, i, o, u, and x specifications when handling **char** type values.

A.7 HANDLING OF ARRAYS

C99 introduces some features that enhance the implementation of arrays.

Variable-Length Arrays

In ANSI C, we must declare array dimensions using integer constants and therefore the size of an array is fixed at compile time. C99 permits declaration of array dimensions using integer variables or any valid integer expressions. The values of these variables can be specified just before they are used. Such arrays are called *variable-length arrays*.

Example:

```
main()
{
    int m, n;
    scanf("%d %d", &m, &n);
    float matrix [ m ] [ n ];      /* variable-length array */
    . . . . .
    . . . . .
}
```

We can specify the values of m and n at run time interactively thus creating the matrix with different size each time the program is run.

Type Specification in Array Declaration

When we pass arrays as function arguments, we can qualify the dimension parameters with the keyword **static**. For example:

A1.4 Computer Programming

```
void array (int x [ static 20 ])
{
    . . . . .
    . . . . .
}
```

The qualifier **static** guarantees that the array **x** contains at least the specified number of elements.

Flexible Arrays in Structures

When designing structures, C99 permits declaration of an array without specifying any size as the last member. This is referred to as a **flexible array member**. Example:

struct find

```
{
    float x;
    int number;
    float list [ ];      /* flexible array */
};
```

A.8 FUNCTIONS IMPLEMENTATION

C99 has introduced some changes in the implementation of functions. They include the following:

- Removal of “default to **int**” rule
- Removal of “implicit function declaration”
- Restrictions on **return** statement
- Making functions **inline**

Default to int Rule

In ANSI C, when the return type of a function is not specified, the return type is assumed to be **int**. For example:

```
prod(int a, int b)      /* return type is int by default */
{
    return (a*b);
}
```

is a valid definition. The return type is assumed to be **int** by default. The implicit **int** rule is not valid in C99. It requires an explicit mention of return type, even if the function returns an integer value. The above definition must be written as:

```
int prod(int a, int b)  /* explicit type specification */
{
    return (a*b);
}
```

Another place where we use implicit **int** rule is when we declare function parameters using qualifiers. For example, function definitions such as

```
fun1( const a)          /* a is int by default */
{
    . . . . .
}
```


and

```
fun2 (register x, register y)      /* x and y are int */
{
    . . . . .
}
```

are not acceptable in C99. The parameters a, x and y must be explicitly declared as **int**, like:

```
const int a
const register x
```

Explicit Function Declaration

Although prior explicit declaration of function is not technically required in ANSI C, it is required in C99 (like in C++).

Restrictions on Return Statement

In ANSI C, a non-void type function can include a **return** statement without including a value. For example, the following code is valid in ANSI C.

```
float value (float x, float y)
{
    . . . . .
    . . . . .
    return;                      /* no value included */
}
```

But, in C99, if a function is specified as returning a value, its return statement must have a return value specified with it. Therefore, the above definition is not valid in C99. The **return** statement for the above function may take one of the following forms:

```
return(p);          /* p contains float value */
return(p);
return 0.0;         /* when no value to be returned*/
```

Making Functions Inline

The new keyword **inline** is used to optimize the function calls when a program is executed. The **inline** specifier is used in function definition as follows:

```
inline mul (int x, int y)
{
    return (x*y);
}
```

Such functions are called *inline functions*. When an inline function is invoked, the function's code is expanded inline, rather than called. This eliminates a significant amount of overhead that is required by the calling and returning mechanisms thus reducing the execution time considerably. However, the expansion "inline" may increase the size of the object code of the program when the function is invoked many times. Due to this, only small functions are made inline.

A.9 RESTRICTED POINTERS

The new keyword **restrict** has been introduced by C99 as a type qualifier that is applied only to pointers. A pointer qualified with **restrict** is referred to as a *restricted pointer*. Restricted pointers are declared as follows.

```
int *restrict p1;
void *restrict p2;
```

A pointer declared “restricted” is the only means of accessing the object it points to. (However, another pointer derived from the restricted pointer can also be used to access the object.)

Pointers with **restrict** specifier are mainly used as function parameters. They are also used as pointers that point to memory created by **malloc ()** function.

C99 has added this feature to the prototype of many library functions, both existing and new. For details, you must refer to the functions defined in the C standard library.

A.10 COMPILER LIMITATIONS

All language compilers have limitations in terms of handling some features such as the length of significant characters, number of arguments in functions, etc. C99 has enhanced many of such limitations. They are listed below:

- Significant characters in identifiers: increased from 6 to 31
- Levels of nesting of blocks : Increased from 15 to 127
- Arguments in a function : Increased from 31 to 127
- Members in a structure : Increased from 127 to 1023

C11 Features

A.11 INTRODUCTION

C11 is the current C standard that replaces the previous standard i.e. C99. C11 mainly aims at standardizing features that are common to most of the present-day compilers. To overcome one of the limitations of C99 where there was a delay in compilers conforming to the C99 standard, C11 makes several of its features optional. This makes it easier for the compilers to conform to the C11 standard.

A.12 LANGUAGE AND LIBRARY SPECIFICATION CHANGES

Let's now take a look at some of the key language and library specification changes that C11 brings to the C99 standard:

_Noreturn Function Specifier

It is used in the function declaration statement to indicate that the function does not return either by a return statement or by reaching the end of the function body. The specifier is provided in the `stdnoreturn.h` header file.

Some of the standard library functions that are declared using noreturn specifier include `abort()`, `exit()`, `longjmp()`, etc.

_Generic Keyword

It provides a method for choosing one of the given expressions at the time of compilation.

Syntax: `_Generic (controlling-expression, association-list)`

Here, controlling-expression is any expression, except the comma operator, and association-list is a comma-separated list of associations.

_Thread_local Specifier

The variables declared with `_Thread_local` specifier are given thread-specific storage duration. That is, the variables are allocated when the thread begins and deallocated when the thread ends. Further, conforming to the thread-specific behavior, each thread has its own instance of the variable. The new header file `<threads.h>` contains `_Thread_local` specifier.

Enhanced Unicode Support

C11 supports certain enhancements related to Unicode support. For instance,

- New header file `uchar.h` comprising of UTF-16 and UTF-32 related character utilities
- `u` and `U` string literal prefixes
- `u8` prefix for UTF-8 encoded literals

Anonymous Structures and Unions

C11 supports implementation of anonymous structures and unions, which are specifically useful in situations where unions and structures are nested. The following example code illustrates this:

```
struct S
{
    char c;
    union
    {
        int a; float b;
    };
};
```

Note that in the above code union is anonymous, as it has not been named.

ADDITION OF `quick_exit` FUNCTION

It is a new function for quick program termination. It terminates the program without cleaning the system resources.

REPLACEMENT OF `gets` WITH `gets_s`

The `gets()` function has been completely removed with C11 as it does not perform bounds checking resulting in buffer overflow situations. The new `gets_s()` function overcomes this limitation by reading at most characters from `stdin`. It keeps reading until a new line or end-of-file is reached and returns only a part of the read characters to the input buffer.

ALIGNMENT SPECIFICATION

C11 provides four new macros for alignment of objects. These are:

```
_alignas  
_alignof  
_alignas_is_defined  
_alignof_is_defined.
```

These macros are provided in the new stdalign.h header.

NOTE: For a complete list of C11 standard features, you may refer the ISO/IEC 9899:2011 specification.

Solved Question Paper

Jan-Feb 2015 Set 1

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supplementary Examinations Jan/Feb - 2015

COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem E, Aero E, AME, Min E, PE, & Metal E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**
Answering the question in **Part-A** is Compulsory,
Three Questions should be answered from **Part-B**

PART-A

1. (i) C is a structured programming language. Explain.
- (ii) Discuss about nested if with example.
- (iii) What are header files? Explain.
- (iv) Differentiate between pointer variable and normal variable.
- (v) Explain about bit fields.
- (vi) Write a short notes on files.

[3+4+4+4+3+4]

- Sol.** (i) Refer Section 2.2
(ii) Refer Section 3.2.3
(iii) Refer Section 2.13
(iv) A variable has 3 things when you make them:

```
int <- type  
x <- name  
=3 <- data
```

A pointer doesn't have any data in themselves, they "point" to a variable. Like this:

```
int x = 3;  
int *x = x
```

Now the *x is 3, because it "points" to the x variable (actually it points to its memory address)
*x = 6 means that where the pointer is pointing (the memory part) will be filled with this data.

Incrementing (i++) a pointer can do two different things:

(*x) ++ adds +1 to the value , while *x++ will move the "pointing" to the next thing in the memory (if we are not talking about an array, then it's usually memory garbage)

Normal variable stores a value of the given datatype whereas the pointer variable stores the address of a variable.

```
for example int n=10;  
int *p;  
p=&n;
```


here p is a pointer variable and n is a normal variable. p stores the address of n whereas n stores an integer value. *p prints the value of n, p prints the address of n.

(v) Refer Section 8.9

(vi) Refer Section 9.1

PART-B

2. (a) List and explain different types of operators in C.

(b) Write a program to find primes in the given range.

[8+8]

Sol. (a) Refer Section 2.9

(b) `#include<stdio.h>`

```
int main()
{
    int n1, n2, i, flag;
    printf("Enter two numbers.");
    scanf("%d",&n1, &n2);

    printf("Prime numbers between %d and %d are: "n1, n2);

    while(n1<n2)
    {
        flag=0;
        for(i=2;i<=n1/2;++i)
        {
            if(n1%i ==0)
            {
                flag=1;
                break;
            }
        }

        if(flag==0)
            printf("%d",n1);

        ++n1;
    }

    return 0;
}
```

3. (a) Explain any two iterative statements with examples.

(b) Write a program for calculating the length of a string without using string handling functions.

[6+10]

Sol. (a) Refer Section 3.6

(b) A program to count the length of a string is shown in Fig. 1. 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

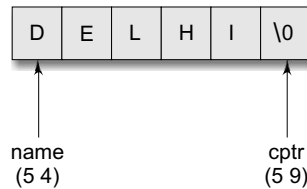
```
while(*cptr != '\0')
```

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. 1 String handling by pointers

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

SQP4 Computer Programming

```
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.

4. (a) **What is the difference between recursive and non-recursive functions? Give their merits and demerits.**

(b) **Write a recursive function for finding the factorial value of a given number.** [8+8]

Sol. (a) For definition of recursion refer Section 4.10

Difference between recursive and non-recursive functions:

- A recursive function in general has an extremely high time complexity while a non-recursive one does not.
- A recursive function generally has smaller code size whereas a non-recursive one is larger.
- In some situations, only a recursive function can perform a specific task, but in other situations, both a recursive function and a non-recursive one can do it.
- A recursive function is a function which calls itself.

Advantages of recursive functions:

- Avoidance of unnecessary calling of functions.
- A substitute for iteration where the iterative solution is very complex. For example to reduce the code size for Tower of Hanoi application, a recursive function is best suited.
- Extremely useful when applying the same solution

Disadvantages of recursive functions:

- A recursive function is often confusing.
- The exit point must be explicitly coded.
- It is difficult to trace the logic of the function.

(b) Refer Section 4.10

5. (a) **How pointers can be used for declaring of multidimensional arrays? Discuss.**

(b) **Write a program to multiply two matrices using pointer.** [8+8]

Sol. (a) Refer Section 7.9

```
(b) #include<stdio.h>  
#include<stdlib.h>  
int main(void)  
{  
    int a[10][10],b[10][10],c[10][10],n=0,m=0,i=0,j=0,p=0,q=0,k=0;  
    int *pt,*pt1,*pt2;  
    printf("Enter size of 1st 2d array : ");  
    scanf("%d %d",&n,&m);
```



```

    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            printf("Enter element no. %d %d :",i,j);
            scanf("%d",&a[i][j]);
        }
    }
    printf("Enter size of 2nd 2d array : ");
    scanf("%d %d",&p,&q);
    for(i=0;i<p;i++)
    {
        for(j=0;j<q;j++)
        {
            printf("Enter element no. %d %d :",i,j);
            scanf("%d",&b[i][j]);
        }
    }
    if(m!=p)
    {
        printf("Multiplication cannot be done\n");
        exit (0);
    }
    pt=&a[0][0];
    pt1=&b[0][0];
    pt2=&c[0][0];
    for(i=0;i<n;i++)
    {
        for(k=0;k<q;k++)
        {
            *(pt2+(i*10+k))=0;
            for(j=0;j<m;j++)
            {
                *(pt2+(i*10+k))+=*(pt+(i*10+j))**(pt1+(j*10+k));
            }
        }
    }
    for(i=0;i<n;i++)
    {
        for(j=0;j<q;j++)
        {
            printf("%d ",c[i][j]);
        }
    }
    printf("\n");
    return 0;
}

```


6. (a) Write a short note on unions within structures.

(b) Write a program to create a linked list.

[8+8]

Sol. (a) Refer Section 8.2, 8.8

Example:

```
#include <string.h>
```

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

```
typedef union {  
    int units;  
    float kgs;  
} amount ;
```

```
typedef struct {  
    char selling[15];  
    float unitprice;  
    int unittype;  
    amount howmuch;  
} product;
```

```
int main() {
```

```
    product dieselmotorbike;  
    product apples;  
    product * myebaystore[2];
```

```
    int nitems = 2; int i;
```

```
    strcpy(dieselmotorbike.selling,"A Diesel Motor Cycle");  
    dieselmotorbike.unitprice = 5488.00;  
    dieselmotorbike.unittype = 1;  
    dieselmotorbike.howmuch.units = 4;
```

```
    strcpy(apples.selling,"Granny duBois");  
    apples.unitprice = 0.78;  
    apples.unittype = 2;  
    apples.howmuch.kgs = 0.5;
```

```
    myebaystore[0] = &dieselmotorbike;  
    myebaystore[1] = &apples;
```

```
    for (i=0; i<nitems; i++) {  
        printf("\n%s\n",myebaystore[i]->selling);  
        switch (myebaystore[i]->unittype) {  
            case 1:  
                printf("We have %d units for sale\n",  
                    myebaystore[i]->howmuch.units);  
                break;  
            case 2:  
                printf("We have %f kgs for sale\n",
```



```

                                myebaystore[i]->howmuch.kgs);
                                break;
                                }
                                }
}

```

- (b) The program first allocates a block of memory dynamically for the first node using the statement

```
head = (node *)malloc(sizeof(node));
```

which returns a pointer to a structure of type node that has been type defined earlier. The linked list is then created by the function create. The function requests for the number to be placed in the current node that has been created. If the value assigned to the current node is -999, then null is assigned to the pointer variable next and the list ends. Otherwise, memory space is allocated to the next node using again the malloc function and the next value is placed into it. Not that the function create calls itself recursively and the process will continue until we enter the number -999.

The items stored in the linked list are printed using the function **print**, which accept a pointer to the current node as an argument. It is a recursive function and stops when it receives a NULL pointer. Printing algorithm is as follows;

1. Start with the first node.
2. While there are valid nodes left to print
 - (a) print the current item; and
 - (b) advance to next node.

Similarly, the function count counts the number of items in the list recursively and return the total number of items to the main function. Note that the counting does not include the item -999 (contained in the dummy node).

```

Program
#include <stdio.h>
#include <stdlib.h>
#define NULL 0
struct linked_list
{
    int number;
    struct linked_list *next;
};
typedef struct linked_list node; /*node type defined*/
main()
{
    node *head;
    void create(node *p);
    int count(node *p);
    void print(node *p);
    head = (node *)malloc(sizeof(node));
    create(head);
    printf("\n");
    printf(head);
    printf("\n");
}

```



```

        printf("\nNumber of items = %d \n", count(head));
    }
    void create(node *list)
    {
        printf("Input a number\n");
        printf("(type -999 at end): ");
        scanf("%d", &list->number); /*create current node*/

        if(list->number == -999)
        {
            list->next = NULL;
        }
        else /*create next node*/
        {
            list->next = (node *)malloc(sizeof(node));
            create(list->next); /* Recursion occurs*/
        }
        return;
    }
    void print(node *list)
    {
        if(list->next != NULL)
        {
            printf("%d-->", list->number); /*print current item*/

            if(list->next->next == NULL)
                printf("%d", list->next->number);
            print(list->next); /*move to next item*/
        }
        return;
    }
    int count(node *list)
    {
        if(list->next == NULL)
            return (0);
        else
            return(1+ count(list->next));
    }

```

Output

```

Input a number
(type -999 to end); 60
Input a number
(type -999 to end); 20
Input a number
(type -999 to end); 10
Input a number
(type -999 to end); 40
Input a number
(type -999 to end); 30

```



```

Input a number
(type -999 to end); 50
Input a number
(type -999 to end); -999
60 -->20 -->10 -->40 -->30 -->50 --> -999
Number of items = 6

```

Fig. 2 *Creating a linear linked list***7. (a) Discuss about unformatted I/O with suitable examples.****(b) Write a program to print file contents in reverse order.****[8+8]****Sol. (a) Unformatted I/O in C**

Unformatted I/O functions works only with character datatype (char).

The unformatted input functions used in C are getch(), getche(), getchar(), gets().

Syntax for getch () in C :

```
variable_name = getch();
```

getch() accepts only single character from keyboard. The character entered through getch() is not displayed in the screen (monitor).

Syntax for getche() in C :

```
variable_name = getche();
```

Like getch(), getche() also accepts only single character, but unlike getch(), getche() displays the entered character in the screen.

Syntax for getchar() in C :

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

getchar() accepts one character type data from the keyboard.

Syntax for gets() in C :

```
gets(variable_name);
```

gets() accepts any line of string including spaces from the standard Input device (keyboard). gets() stops reading character from keyboard only when the enter key is pressed.

The unformatted output statements in C are putch, putchar and puts.

Syntax for putch in C :

```
putch(variable_name);
```

putch displays any alphanumeric characters to the standard output device. It displays only one character at a time.

Syntax for putchar in C :

```
putchar(variable_name);
```

putchar displays one character at a time to the Monitor.

Syntax for puts in C :

```
puts(variable_name);
```

puts displays a single / paragraph of text to the standard output device.

Example

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


SQP10 *Computer Programming*

```
#include<conio.h>
void main()
{
char a[20];
gets(a);
puts(a);

getch();
}
```

(b)

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 souce 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*/
            j=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
```

Fig. 3 Program to reverse *n* characters in a file

Solved Question Paper

Jan-Feb 2015 Set 2

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supplementary Examinations Jan/Feb - 2015

COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem E, Aero E, AME, Min E, PE, & Metal E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**
Answering the question in **Part-A** is Compulsory,
Three Questions should be answered from **Part-B**

PART-A

1. (i) What is pseudo code? Explain.
- (ii) Differentiate between break and exit.
- (iii) Explain about block structure.
- (iv) What is indirect pointer? Discuss.
- (v) What is union? Explain.
- (vi) Write a program to read a text file and to print the count the no of tab characters in a given file. [2+4+4+4+4+4]

Sol. (i) A pseudo code is an outline of a program, written in a form that can easily be converted into real programming statements. For example, the pseudocode for a bubble sort routine might be written:

```
while not at end of list
  compare adjacent elements
  if second is greater than first
    switch them
  get next two elements
  if elements were switched
  repeat for entire list
```

Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules. It is simply one step, an important one, in producing the final code. The benefit of pseudocode is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, you can write pseudocode without even knowing what programming language you will use for the final implementation.

Pseudocode is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programming language. Pseudocode is sometimes used as a detailed step in the process of developing a program. It allows

designers or lead programmers to express the design in great detail and provides programmers a detailed template for the next step of writing code in a specific programming language.

- (ii) The differences between break and exit are mentioned as follows:

break	exit()
break is a keyword in C.	exit() is a standard library function.
break causes an immediate exit from the switch or loop (for, while or do).	exit() terminates program execution when it is called.
break is a reserved word in C; therefore it can't be used as a variable name.	exit() can be used as a variable name.
No header files needs to be included in order to use break statement in a C program.	stdlib.h needs to be included in order to use exit().
break transfers the control to the statement follows the switch or loop (for, while or do) in which break is executed.	exit() returns the control to the operating system or another program that uses this one as a sub-process.
Example of break <pre>// some code here before while loop while(true) { ... if(condition) break; } // some code here after while loop</pre>	Example of exit() <pre>// some code here before while loop while(true) { ... if(condition) exit(-1); } // some code here after while loop</pre>
In the above code, break terminates the while loop and <i>some code here after while loop</i> will be executed after breaking the loop.	In the above code, when if(condition) returns <i>true</i> , exit(-1) will be executed and the program will get terminated. Upon call of exit(-1); -1 will be returned to the calling program that is operating system most of the time. The <i>some code here after while loop</i> will never be executed in this case.
Conclusively, break is a program control statement which is used to alter the flow of control upon a specified conditions.	exit() is a library function, which causes immediate termination of the entire program, forcing a return to the operating system.

- (iii) In computer programming, a block or code block is a section of code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called a block-structured programming language. Blocks are fundamental to structured programming, where control structures are formed from blocks.

The function of blocks in programming is to enable groups of statements to be treated as if they were one statement, and to narrow the lexical scope of variables, procedures and functions declared in a block so that they do not conflict with variables having the same name used elsewhere in a program for different purposes. In a block-structured programming language, the names of variables and other objects such as procedures which are declared in outer blocks are visible inside other inner blocks, unless they are shadowed by an object of the same name.

(iv) Refer Section 7.7

(v) Refer Section 8.8

(vi)

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    char ch;
    int not=0;
    clrscr();
    fp=fopen("animesh.txt","r");
    while(1)
    {
        ch=fgetc(fp);
        if(ch==EOF)
            break;
        if(ch=='\t')
            not++;
    }
    fclose(fp);
    printf("\n Number of tabs = %d",not);
    getch();
}
```

PART-B

2. (a) What is a data type? Discuss about the range for different data types.

(b) Write a program to find the sum of the series: $1+2^2+3^2+$

[8+8]

Sol. (a) Refer Section 1.4

(b)

```
#include <stdio.h>
int main()
{
    int number, i;
    int sum = 0;

    printf("Enter maximum values of series number: ");
    scanf("%d", &number);
    sum = (number * (number + 1) * (2 * number + 1)) / 6;
    printf("Sum of the above given series : ");
    for (i = 1; i <= number; i++)
    {
        if (i != number)
            printf("%d^2 + ", i);
        else
            printf("%d^2 = %d ", i, sum);
    }
    return 0;
}
```


3. (a) **Differentiate between iteration and branching.**

(b) **Write a program to print the day of the week using switch and else-if and also give the comparison between using of switch and else-if.** [8+8]

Sol. (a) Iteration

When a process or sequence in a computer program is repeated, this is an iteration. In a computer program, a common form of iterations is a loop, which repeats code to determine values for multiple variables or sometimes just a single variable

Iteration, in the context of computer programming, is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met. When the first set of instructions is executed again, it is called an iteration. When a sequence of instructions is executed in a repeated manner, it is called a loop.

- The for statement
- The while statement
- The do-while statement

Branching

In programming the order of execution of instructions may have to be changed depending on certain conditions. 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 instructions accordingly.

The C language programs follow a sequential form of execution of statements. Many times it is required to alter the flow of sequence of instructions. C language provides statements that can alter the flow of a sequence of instructions. These statements are called as control statements. To jump from one part of the program to another, these statements help. The control transfer may be unconditional or conditional. Branching Statement are of following categories:

- If Statement
- The If else Statement
- Compound Relational tests
- Nested if Statement
- Switch Statement

(b) **Advantages of switch over if-else ladder:**

- A switch statement works much faster than equivalent if-else ladder. It is because compiler generates a jump table for a switch during compilation. Consequently, during execution, instead of checking which case is satisfied, it only decides which case has to be executed.
- It is more readable and in compare to if-else statements.
- It is more manageable for having higher level of indentation than if. For instance check below two source codes (solving same problem one using else if the other using switch) to check error messages.

Where to use switch over if-else ladder:

- If there are large number of compares for a condition in your program, use switch over if-else ladder.
- For more complex comparisons.

Program using Switch:

```
/*Program to print the day of the week using switch*/
#include <stdio.h>
int main()
{
    int week;
    printf("Enter week number(1-7): ");
    scanf("%d", &week);
    switch(week)
    {
        case 1: printf("MONDAY");
                break;
        case 2: printf("TUESDAY");
                break;
        case 3: printf("WEDNESDAY");
                break;
        case 4: printf("THURSDAY");
                break;
        case 5: printf("FRIDAY");
                break;
        case 6: printf("SATURDAY");
                break;
        case 7: printf("SUNDAY");
                break;
        default: printf("Invalid input! Please enter week number between 1-7");
    }
    return 0;
}
```

Program using else if:

```
/*Program to print the day of the week using else if*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    clrscr();
    printf("\n Enter Day of week as Number 1 to 7 ");
    scanf("%d",&n);
    if(n == 1)
        printf("\n MONDAY");
    else if(n == 2)
        printf("\n TUESDAY");
    else if(n == 3)
        printf("\n WEDNESDAY");
    else if(n == 4)
        printf("\n THURSDAY");

    else if(n == 5)
        printf("\n FRIDAY");
```



```

else if(n == 6)
printf("\n SATURDAY");

else if(n == 7)
printf("\n SUNDAY");
else
printf("\n WRONG INPUT ");
getch();
}

```

4. (a) What is user defined functions? Discuss with an example.

(b) Write a recursive function to find GCD value.

[8+8]

Sol. (a) Refer Section 4.2

(b) Ref Example 4.7

5. (a) What is copy by value and copy by address? Discuss.

(b) Write a program to illustrate passing by address example.

[8+8]

Sol. (a) Refer Section 4.12.1

(b) The program in Fig. 1 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. 1 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 11.4. We can also use this technique in designing user-defined functions discussed in Chapter 10. Let us consider the problem sorting an array of integers discussed in Example 10.6.

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 11.10). 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.

6. (a) Explain about the bitwise operators with examples.

(b) Write a program to find the one's complement for the given number.

[8+8]

Sol. (a) Refer Section 2.9.7

```
(b) #include <stdio.h>
#include <string.h>
#define SIZE 8
int main()
{
    char binary[SIZE + 1], onesComp[SIZE + 1];
    int i, error=0;
    printf("Enter any %d bit binary value: ", SIZE);
    gets(binary);

    for(i=0; i<SIZE; i++)
    {
        if(binary[i]=='1')
        {
            onesComp[i] = '0';
        }
        else if(binary[i]=='0')
        {
            onesComp[i] = '1';
        }
        else
        {
            printf("Invalid Input");
            error = 1;
            break;
        }
    }
    onesComp[SIZE] = '\0';
    if(error==0)
    {
        printf("\nOriginal binary = %s\n", binary);
        printf("Ones complement = %s", onesComp);
    }
    return 0;
}
```


7. (a) How to read from and write to a file? Explain with examples.

(b) Write a program to find the n^{th} occurrence of a given word in a given file.

[8+8]

Sol. (a) Refer Section 9.4

```
(b) #include<stdio.h>
#include<conio.h>
#include<string.h>
int main() {
    FILE * f;
    int count = 0, i;
    char buf[50], read[100];
    printf("Which file to open\n");
    fgets(buf, 50, stdin);
    buf[strlen(buf) - 1] = '\0';
    if (!(f = fopen(buf, "rt"))) {
        printf("Wrong file name");
    } else printf("File opened successfully\n");
    for (i = 0; fgets(read, 100, f) != NULL; i++) {
        if (read[i] == 'if') count++;
    }
    printf("Result is %d", count);
    getch();
    return 0;
}
```

Solved Question Paper

Jan-Feb 2015 Set 3

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supplementary Examinations Jan/Feb - 2015

COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem E, Aero E, AME, Min E, PE, & Metal E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**
Answering the question in **Part-A** is Compulsory,
Three Questions should be answered from **Part-B**

PART-A

1. (i) Explain about enum data type.
 - (ii) Differentiate between do-while and for loop.
 - (iii) Explain about C preprocessor with an example.
 - (iv) What are actual and former parameters.
 - (v) What is left shift? How is it different from right shift?
 - (vi) Write a program to read a text file and to count the no of uppercase letters in a given file.
- [4+4+3+4+3+4]

Sol. (i) Refer Section 2.12.2

(ii) Refer Section 3.8, 3.9

- (iii) C is a unique language in many respects. We have already seen features such as structures and pointers. Yet another unique feature of the C language is the *preprocessor*. The C preprocessor provides several tools that are unavailable in other high-level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable, and more efficient.

The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines or directives*. 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. We have already used the directives #define and #include to a limited extent. A set of commonly used preprocessor directives and their functions is given in Table 1.

TABLE I 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.

These directives can be divided into three categories:

1. Macro substitution directives.
2. File inclusion directives.
3. Compiler control directives.

(iv) Refer Section 4.8

(v) Refer Table 2.8

(vi) #include<stdio.h>

```
void main()
{
    FILE *f1;
    char c;
    f1=fopen("file1","r");
    while(c=getc(f1)!=EOF)
    {
        if(c>='A' || c<='Z')
            count++;
    }
    printf("number of upper case charaters are: %d",count);
    getch();
}
```

PART-B

2. (a) **What is algorithm? Write an algorithm and flowchart for the finding the given no is Armstrong no or not?**

(b) **Write a C program to calculate the total of the series: $1+(1/2^2)+(1/3^2)+.....$ [8+8]**

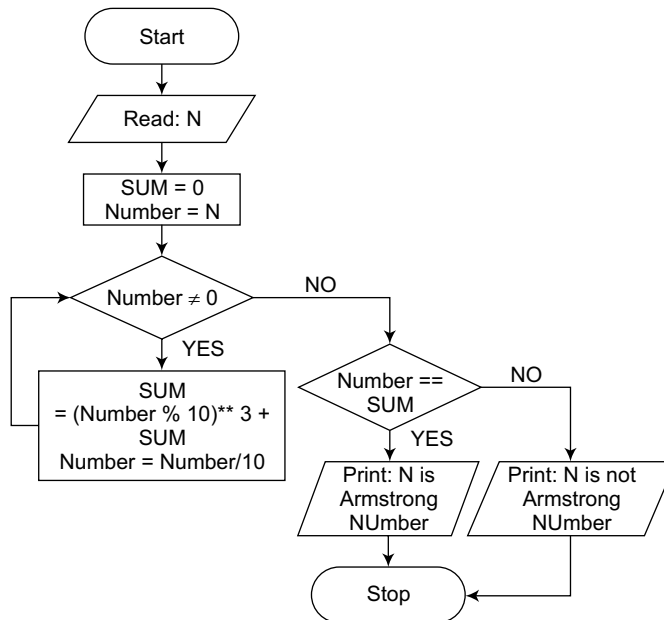
Sol. (a) Refer Section 1.6

Algorithm to Check Armstrong Number

```
Step 1: Start
Step 2: Take Input N as Integer
Step 3: [ initializing SUM ] Set: SUM = 0
Step 4: Set: Number = N
Step 5: Repeat While Number ≠ 0
    SUM = (Number%10)**3 + SUM
    Number = Number/10
[ End of While Loop ]
```


Step 6: [Checking ?]
 If SUM == N then
 Print: N is an Armstrong Number.
 Else
 Print: N is not an Armstrong Number.
 [End of If-Else Structure]
 Step 7: Exit

Flowchart to Check Armstrong Number



(b) `#include <stdio.h>`
`#include <math.h>`
`void main()`
`{`
`int i, n;`
`float sum = 0, term = 0;`
`printf("\nEnter number: ");`
`scanf("%d", &n);`

`for(i = 1; i <= n; i++)`
`{`
`term = 1.0 / i;`
`term = pow(term, i);`
`sum = sum + term;`
`printf("\nTerm: %-5d Value = %.4f Sum = %.6f", i, term, sum);`
`}`

`printf("\n\n");`
`}`

3. (a) Differentiate between string and array? What are the applications of an array? Discuss.

(b) Write a program to find the 4^{3^2} value. [8+8]

Sol. (a) Difference between Strings and Arrays:

- String can hold only char data whereas an array can hold any data type.
- An array size cannot be changed whereas a string size can be changed if it is a char pointer
- The last element of an array is an element of the specific type. The last character of a string is a null – ‘\0’ character.
- The length of an array is to specified in [] at the time of declaration (except char[]). The length of the string is the number of characters + one (null character).
- Whenever we require a collection of data objects of the same type and want to process them as a single unit, an array can be used, provided the number of data items is constant or fixed.

Arrays have a wide range of applications ranging from business data processing to scientific calculations to industrial projects.

(b)

```
#include<stdio.h>
#include<math.h>
void main()
{
    int a,b,c,val,result;
    clrscr();
    printf("Enter 3 numbers");
    scanf("%d%d%d",&a,&b,&c);
    val=pow(a,b);
    result=pow(val,c);
    printf("%d"result);
    getch();
}
```

4. (a) Explain about call by value and call by reference mechanisms.

(b) Write a non recursive program for calculating the factorial of a number using functions. [8+8]

Sol. (a) Refer Section 4.12.1 , 7.10

(b)

```
#include<conio.h>
int main()
{
    int num;
    printf("\n >> PROGRAM TO FIND FACTORIAL OF GIVEN NUMBER
        using Function <<\n");
    printf("\n Enter the Number whose Factorial you want: ");
    scanf("%d",&num);
    printf("\n The factorial of %d is %d.\n\n",
        num,factorial(num));

    return 0;
}
```



```

factorial(num1)
{
    int i,fact=1;
    for(i=num1; i>=2 ; i--)
    {
        fact = fact * i;
    }
    return fact;
}

```

5. (a) What is the importance of * and **? Explain about the initialization and declaration of pointer variables.

(b) Write a program for illustrating the dynamic memory allocation. [8+8]

Sol. (a) Refer Section 7.3, 7.4

(b) Refer Section 7.16

6. (a) Explain about structures and functions.

(b) Write a program to illustrate structures and functions. [8+8]

Sol. (a) Refer Section 8.6

(b) Refer Example 8.6

7. (a) What is file? Explain about the input and output functions of files.

(b) Write a program to illustrate file operations. [8+8]

Sol. (a) Refer Section 9.1, 9.4

(b) Refer Example 9.1

Solved Question Paper

Jan-Feb 2015 Set 4

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supplementary Examinations Jan./Feb. - 2015

COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem E, Aero E, AME, Min E, PE, & Metal E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**
Answering the question in **Part-A** is Compulsory,
Three Questions should be answered from **Part-B**

PART-A

1. (i) How to execute C program in Linux? Explain with example.
- (ii) Differentiate between do-while and while-do.
- (iii) Differentiate between 1D and 2D arrays.
- (iv) Explain about character pointer.
- (v) Discuss about rotation.
- (vi) What is binary file? Discuss.

[4+4+3+4+3+4]

- Sol.** (i) You can type your C program using any of the editors that are available under Linux such as *vi* or *emacs* or any other editor.

Once you have written and saved your C program using any editor return to the prompt. An *ls* command should display your C program. It should have the .c extension. Now at the prompt type the following:

```
$ gcc -o firstprogram firstprogram.c
```

If your file is named firstprogram.c then type '-o firstprogram' as the parameter to gcc. This is basically your suggested name for the executable file that gcc would create. In case you typed something like the following:

```
$ gcc firstprogram.c
```

You would be having an a.out in the same directory as the source C file. This is the default name of the executable that gcc creates. This would create problems when you compile many programs in one directory. So you override this with the -o option followed by the name of the executable

```
$ gcc -o hello secondprogram.c
```

Would create an executable by the name *hello* for your source code named secondprogram.c
Running the executable that you created is as simple as typing the following at the prompt.

```
$ ./firstprogram
```


- (ii) 1. In While loop the condition is tested first and then the statements are executed if the condition turns out to be true.

In do while the statements are executed for the first time and then the conditions are tested, if the condition turns out to be true then the statements are executed again.

2. A do while is used for a block of code that must be executed at least once.

These situations tend to be relatively rare, thus the simple while is more commonly used.

3. A do while loop runs at least once even though the the condition given is false while loop do not run in case the condition given is false

4. In a while loop the condition is first tested and if it returns true then it goes in the loop

In a do-while loop the condition is tested at the last.

5. While loop is entry control loop
where as do while is exit control loop.

6. Syntax:

```
while loop :
while (condition)
{
    Statements;
}
```

```
do while loop :
do
{
    Statements;
}while(condition);
```

- (iii) Refer Section 5.2, 5.5

- (iv) Since text strings are represented in C by arrays of characters, and since arrays are very often manipulated via pointers, character pointers are probably the most common pointers in C.

However, pointers only hold an address, they cannot hold all the characters in a character array. This means that when we use a `char *` to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

Below is how you might use a *character pointer* to keep track of a string.

```
char label[] = "Single";
char label2[10] = "Married";
char *labelPtr;
```

```
labelPtr = label;
```

- (v) Refer Table 2.8

- (vi) Binary files are very similar to arrays of structures, except the structures are in a disk-file rather than an array in memory. Binary files have two features that distinguish them from text files:

- You can instantly use any structure in the file.
- You can change the contents of a structure anywhere in the file.

After you have opened the binary file, you can read and write a structure or seek a specific position in the file. A file position indicator points to record 0 when the file is opened.

A read operation reads the structure where the file position indicator is pointing to. After reading the structure the pointer is moved to point at the next structure.

A write operation will write to the currently pointed-to structure. After the write operation the file position indicator is moved to point at the next structure.

The fseek function will move the file position indicator to the record that is requested.

Remember that you keep track of things, because the file position indicator can not only point at the beginning of a structure, but can also point to any byte in the file.

The fread and fwrite function takes four parameters:

- A memory address
- Number of bytes to read per block
- Number of blocks to read
- A file variable

fread(&my_record,sizeof(struct rec),1,ptr_myfile);

PART- B

2. (a) Differentiate between hardware and software.
- (b) Explain about the history of C programming language.
- (c) Write a C program that illustrates the unary operators.

[4+6+6]

Sol. (a) Refer Section 1.2

(b) Refer Section 1.3

(c) #include<stdio.h>

#include<conio.h>

void main()

```
{
    int a;
    int b;
    clrscr();
    a=-100;
    printf("\nThe value of a is %d",a); //-100
    a=(10+20);
    printf("\nThe value of a is %d",a); //30
    a++;
    printf("\nThe value of a is %d",a); //31
    printf("\nThe value of a is %d",++a); //32
    b=sizeof(a);
    printf("\nThe size of a is %d",b); //2
    getch();
}
```

3. (a) What is an array? What are the disadvantages of an array? Discuss.

- (b) Write a program to print the following matrix on the screen.

```
a b c d e
f g h i j
k l m n o
p q r s t
```

[8+8]

Sol. (a) Refer Section 5.1

Disadvantages of array:

- We must know in advance that how many elements are to be stored in array.
- Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or reduced.
- Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.
- The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

(b)

```
#include<stdio.h>
void main()
{
    int i,count;
    clrscr();
    count=0;
    for(i=1;i<20;i++)
    {
        printf("%c",i);
        count++;
        if(count%5==0)
            printf("\n");
    }
    getch();
}
```

4. (a) Explain about different storage classes with examples along with scope rules.

(b) Write a program to print Pascal triangle using functions.

[8+8]

Sol. (a) Refer Section 4.13

(b)

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

```
long factorial(int);

int main()
{
    int i, n, c;

    printf("Enter the number of rows you wish to see in pascal triangle\n");
    scanf("%d",&n);

    for (i = 0; i < n; i++)
    {
        for (c = 0; c <= (n - i - 2); c++)
            printf(" ");

        for (c = 0 ; c <= i; c++)
            printf("%ld ",factorial(i)/(factorial(c)*factorial(i-c)));
    }
}
```



```
        printf("\n");
    }

    return 0;
}

long factorial(int n)
{
    int c;
    long result = 1;

    for (c = 1; c <= n; c++)
        result = result*c;

    return result;
}
```

5. (a) Differentiate between direct and indirect pointers with examples.

(b) Write a program to illustrate pointers.

[8+8]

Sol. (a) Refer Section 7.4, 7.7

(b) Refer Figure 7.4

6. (a) What is union? How to declare and initialize unions? Discuss.

(b) Write a program to find two's complement for the given no.

[8+8]

Sol. (a) Refer Section 8.8

(b) Refer Figure 5.4

7. (a) What are different types of operating modes of files? Explain with an example.

(b) Write a program to copy one file contents into another file in reverse order.

[6+10]

Sol. (a) Refer Section 9.2, Example 9.1

```
(b) #include<stdio.h>
#include<conio.h>
void main()
{
    FILE *in,*out;
    char ch,f1[80],f2[80];
    long loc;
    clrscr();

    printf("\n Enter Name of Source File:");
    scanf("%s",&f1);

    printf("\n Enter Name of Target File:");
    scanf("%s",&f2);
```



```
in=fopen(f1,"w");
if(f1==NULL)
{
printf("\nCannot open file");
exit(0);
}
else
{
printf("\nEnter data in file %s(Press q to stop):",f1);
while(1)
{
ch=getchar();
if(ch=='q')
break;
else
fputc(ch,in);
}
fclose(in);
}
in=fopen(f1,"r");
if(f1==NULL)
{
printf("\nfile does not exist");
exit(0);
}
out=fopen(f2,"w");
if(f2==NULL)
{
printf("\ncannot open file");
exit(0);
}
fseek(in, 0, SEEK_END);
loc = ftell(in);

loc = loc-1;

while(loc >= 0)
{
fseek(in, loc, SEEK_END);
ch = fgetc(in);
fputc(ch, out);
loc--;
}
printf("\nfile copied in reverse order successfully");

fcloseall();
getch();
}
```

Solved Question Paper

Nov-Dec 2015 Set 1

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supple. Examinations Nov/Dec - 2015

COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem. E, Aero E, AME, Min E, PE, Metal E, Textile Engg.)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**
Answering the question in **Part-A** is Compulsory,
Three Questions should be answered from **Part-B**

PART-A

1. (a) C is a structured language? Justify the statement.
- (b) Differentiate between while-do and do while.
- (c) What is the importance of #include? explain
- (d) What is pointer? Is *p is similar to &p? explain.
- (e) Discuss about the masks.
- (f) Differentiate between text file and binary file.

[3+4+3+4+4+4]

Sol. (a) Refer Section 2.2

- (b) 1. In While loop the condition is tested first and then the statements are executed if the condition turns out to be true.
In do while the statements are executed for the first time and then the conditions are tested, if the condition turns out to be true then the statements are executed again.
2. A do while is used for a block of code that must be executed at least once.
These situations tend to be relatively rare, thus the simple while is more commonly used.
3. A do while loop runs at least once even though the the condition given is false while loop do not run in case the condition given is false
4. In a while loop the condition is first tested and if it returns true then it goes in the loop
In a do-while loop the condition is tested at the last.
5. While loop is entry control loop
where as do while is exit control loop.
6. Syntax:

```
while loop :  
while (condition)  
{  
    Statements;  
}
```



```

    }

    do while loop :
do
{
    Statements;
}while(condition);

```

- (c) In the C programming language, the `#include` directive tells the preprocessor to insert the contents of another file into the source code at the point where the `#include` directive is found. Include directives are typically used to include the C header files for C functions that are held outside of the current source file.

Syntax

The syntax for the `#include` directive in the C language is:

```
#include <header_file>
```

OR

```
#include "header_file"
```

header_file

The name of the header file that you wish to include. A header file is a C file that typically ends in `“.h”` and contains declarations and macro definitions which can be shared between several source files.

The difference between these two syntaxes is subtle but important. If a header file is included within `<>`, the preprocessor will search a predetermined directory path to locate the header file. If the header file is enclosed in `“”`, the preprocessor will look for the header file in the same directory as the source file.

- (d) Refer Section 7.2, 7.3
- (e) **Masks:** Masking is the process or operation to set bit on to off or off to on in a byte or word. If you need to check whether a specific bit is set, then you need to create an appropriate mask and use bitwise operators.

For example:

```

x =1111 0111 , m=0000 1000=>x&m
Creating a mask
Unsigned char mask=1<I;
Causes ith bit to be set to 1.Since we are testing if bit I is set.
To use the mask on 8-bit data,
Unsigned char is BitIset(unsigned char ch, int i)
{
    Unsigned char mask=1<<i;
    Return mask & ch
}

```

- (f) **Text File:** It is a file in which data are stored using only characters; a text file is written using text stream. Non-character data types are converted to sequence of characters before they are stored in this file. In the text format data are organized into lines terminated by new line character. The text files are human readable form and they can be created and read using any text editor. Text

files are read and written using input/output functions that convert characters to data types: scanf, printf, getchar, putchar.

Binary File: A binary file is a collection of data stored in the internal format of the computer. The binary files are not in human-readable form. There are no lines or new line characters. Binary files are read and written using binary streams known as block input/output functions.

PART-B

2. (a) **Is there any difference between the pre decrement and post decrement operators? Explain.**
 (b) **Write a program for performing the arithmetic calculation.** **[8+8]**

Sol. (a) Refer Section 2.9.5

Yes, computer instructions have the provision of directly subtracting unity to a variable. So, if we use the facility, the operation is quicker than the operation for the statement $A=A+1$; therefore, for increasing and decreasing the values of integral objects by unity, it is better to make use of increment operator (++) and if decrement operator (--), respectively. Another provision with these operators is that the operators may be placed before or after the identifier of the variable. When the operators ++ and -- are placed before the variable name, these are called pre increment and post increment operators respectively. For instance;

```
Int A=10, B=5;
++A* ++B;
```

When the operators are placed after the names they are called post-increment and post-decrement operators. In this case, the increment is carried out after the current use. For example;

```
A++* B++;
```

In case of pre increment/pre decrement operators the present value of the variable is first increased/decreased by unity and this changed value is in application. On the other hand, in case of post increment and post decrement operators, the present value of the variable is used in the application and then its value is incremented/ decremented. Table 1 provides a list of all possible increment and decrement operators along with their applications.

Pre or post	Operator	Applications
Pre-increment	++B	Int B=5,A=6,z; Z=++B* A;
Post-increment	B++	Int B=5,A=6,z; Z=B++* A;
Pre or post	Operator	Applications
Pre-decrement	--B	Int B=5,A=6,z; Z=--B* A;
Post-decrement	B--	Int B=5,A=6,z; Z=B--* A;

- (b) Arithmetic operators on integers and floating point variables.

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


SQP4 Computer Programming

```
int a=4,b=6,k=32,B=-150,E,F;
double d=-1.5,c=-2.5,D,F1;

E=B/b;
D=b/B;
F1=c/d;
F=b%a;
Printf("k/b=%d\n",k/b);
Printf("b/c=%f\n", b/c);
Printf("F=%d \n",F);
Printf("F1= %f\n",F1);
}
```

Output

```
k/b=5
b/c=-2.400000
E=-25
F=2
F1=1.666667
```

3. (a) Explain about the switch statement? What is the importance of the break and continue? Give examples.

- (b) Write a program to sum the digits in a given number.

[8+8]

Sol. (a) Refer Section 3.3

Break and Continue

To exit a loop you can use the break statement at any time. This can be very useful if you want to stop running a loop because a condition has been met other than a loop end condition.

```
#include<stdio.h>
Int main()
{
    Int I;
    I=0;
    While(i<20)
    {
        I++;
        If(i==10)
        Break;
    }
    Return 0;
}
```

In the above example, the while loop will run as long I is smaller than twenty. In the while loop there is an if statement.

- (b) #include<stdio.h>
- ```
Int main()
{
 int n,sum=0,r;
 printf("Enter a number:");
 scanf("%d",&n);
 while(n>0)
 {
```



```

 r=n%10;
 n=n/10;
 sum=sum +r;
}
printf("Sum of digits of number: %d",sum);
return 0;}

```

**4. Explain about different storage classes with examples.**

**[16]**

**Sol.** Refer Section 4.13

**5. (a) How to initialize and access pointer variables? Discuss.**

**(b) Write a program to swap two numbers using pass by address.**

**[8+8]**

**Sol.** (a) Refer Section 7.3

(b) Refer Example 7.6

**6. (a) How to pass structure variable to functions? Explain with example.**

**(b) Write a program to add two complex numbers.**

**[8+8]**

**Sol.** (a) Refer Section 8.6

(b) Refer Example 8.12

**7. (a) Explain about the functions for reading and writing data from a file.**

**(b) Write a program to print command line arguments on screen.**

**[8+8]**

**Sol.** (a) Refer Section 9.4

(b) #include<stdio.h>

```

int main(int args, char *argv[]) {
 int i = 0;
 for (i = 0; i < args; i++)

 printf("\n%s", argv[i]);
 return 0;
}

```

**Output:** Carry out following Steps to Run

1. Save Program
2. Compile Program.
3. Run Program.
4. Now Open Command Prompt.
5. Move to the directory where program is saved.
6. Type following command.



---

# Solved Question Paper

## Nov-Dec 2015 Set 2

---

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supple. Examinations Nov/Dec - 2015

### COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem. E, Aero E, AME, Min E, PE, Metal E, Textile Engg.)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**

Answering the question in **Part-A** is Compulsory,

Three Questions should be answered from **Part-B**

\*\*\*\*\*

### PART-A

1. (a) C is a middle-level language. Justify the statement.
- (b) How to access array elements? Explain.
- (c) What is the importance of auto and register?
- (d) Differentiate between direct and indirect pointers with examples.
- (e) Explain about shift keyword.
- (f) Discuss about the formatted I/O.

[3+4+4+4+3+4]

**Sol.** (a) C is often called a middle-level computer language as it combines the elements of high-level languages with the functionalism of assembly language or in simple words (i) it gives or behaves as high-level language through functions—gives a modular programming and breakup, increased efficiency for reusability; (ii) it gives access to the low level memory through pointers. Moreover, it does support the low-level programming, i.e. assembly language.

- (b) Refer Section 5.2
- (c) Refer Section 4.13.1, 4.13.5
- (d) Refer Section 7.4, 7.7
- (e) Bitwise Right Shift Operator in C
  1. It is denoted by >>
  2. Bit pattern of the data can be shifted by specified number of positions to right
  3. When data is shifted right, leading zero's are filled with zero.
  4. Right shift operator is binary operator [Bi – two]
  5. Binary means operator that require two arguments

Bitwise Left Shift Operator in C

1. It is denoted by <<
2. Bit pattern of the data can be shifted by specified number of positions to left



3. When data is shifted left, trailing zero's are filled with zero.
  4. Left shift operator is binary operator [Bi – two]
  5. Binary means operator that require two arguments
- (f) Refer Section 2.14.3, 2.14.5

## PART- B

2. (a) **What is flow chart? How it is useful in writing the programs? Explain about different symbols in flow chart?**

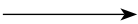



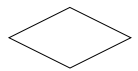



(b) **Draw the flow chart for the Armstrong number and write the program.** **[6+10]**

**Sol.** (a) Flowchart is a diagrammatic representation of an algorithm. Flowcharts are very helpful in writing program and explaining program to others.

Flowcharts are used to analyze, design, document or manage a process or a diagram in different fields. Similar to other kinds of diagrams they help visualize what is going on. This helps to understand a program and find any flaws and bottle necks in it.

### Symbols Used In Flowchart

Different symbols are used for different states in flowchart, for example: input/output and decision making has different symbols. The table below describes all the symbols that are used in making flowchart.

| Symbol                                                                              | Purpose                     | Description                                                                          |
|-------------------------------------------------------------------------------------|-----------------------------|--------------------------------------------------------------------------------------|
|    | Flow line                   | Used to indicate the flow of logic by connecting symbols.                            |
|  | Terminal(Stop/Start)        | Used to represent start and end of flowchart.                                        |
|  | Input/output                | Used for input and output operation.                                                 |
|  | Processing                  | Used for arithmetic operations and data-manipulations.                               |
|  | Decision                    | Used to represent the operation in which there are two alternatives, true and false. |
|  | On-page Connector           | Used to join different flowline                                                      |
|  | Off-page Connector          | Used to connect flowchart portion on different page.                                 |
|  | Predefined Process/Function | Used to represent a group of statements performing one processing task.              |



(b) For flowchart of Armstrong number refer SQP May 2016, Set 3, Q 2 (b).

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int n,sum=0,temp,rem;
 clrscr();
 printf("enter the number you want to check:\n");
 scanf("%d",&n);
 temp=n;
 while(temp!=0)
 {
 Rem=temp%10;
 Sum=sum+rem*rem*rem;
 temp = temp/10;
 }
 if(n==sum)
 printf("number is armstrong number \n");
 else
 printf("number is not an armstrong number \n");
 getch();
}
```

**3. (a) Explain about different iterative statements with examples.**

**(b) Write a program to calculate the ncr value.**

**[8+8]**

**Sol. (a)** Refer Section 3.7, 3.8, 3.9

**Iterative statements:** The term iteration, means repetitive execution of the same set of instructions for a given number of time or until a specified result is obtained. The statements that are executed repetitively are called as iterative statements.

Iterative statements are also popularly known as loop constructs, which repeatedly execute a block of code based on a condition expression, as long as the condition evaluates to true. After that the control transfers to next statement that follows the loop.

(b) #include<stdio.h>  
#include<conio.h>  
int fact(int);  
void main()  
{  
 int n,r,ncr;  
 clrscr();  
 printf("\n enter n and r:");  
 scanf("%d%d",&n,&r);  
 ncr=fact(n)/(fact(n-r)\*fact(r));  
 printf("\n %dc%d: %d",n,r,ncr);  
 getch();  
}  
Int fact(int i)  
{



```

if(i==0)
return 1;
else
 return(i*fact(i-1));
}

```

**4. (a) Explain about the static keyword.**

**(b) Discuss about user defined functions.**

**(c) Write a recursion program for the towers of hanoi.**

**[4+6+6]**

**Sol. (a) Refer Section 4.13.4**

**(b) Refer Section 4.2**

```

(c) #include<stdio.h>
void TowersOfHanoi(int n , char x , char y, char z);
int main()
{
 int n;
 printf("\n enter number of plates:");
 scanf("%d",&n);
 TowersOfHanoi(n-1, 'A', 'B', 'c');
 Return 0;
}
Void TowersOfHanoi(int n, char x, char y, char z)
{
 If(n>0)
 {
 TowersOfHanoi(n-1,x,z,y);
 printf("\n %c ->%c", x,y);
 TowersOfHanoi(n-1,z,y,x);
 }
}

```

**5. (a) How pointers are used to declare single and multi dimension arrays with examples.**

**(b) Write a program for printing the command line arguments.**

**[8+8]**

**Sol. (a) Refer Section 7.13**

**(b) #include<stdio.h>**

```

int main(int args, char *argv[]) {
 int i = 0;
 for (i = 0; i < args; i++)

 printf("\n%s", argv[i]);
 return 0;
}

```

**Output : Carry out following Steps to Run**

1. Save Program
2. Compile Program.



3. Run Program.
4. Now Open Command Prompt.
5. Move to the directory where program is saved.
6. Type following command.

**6. (a) Is it possible to nest the structures? Explain with example.**

**(b) Write a program for reversing a linked list.**

**[8+8]**

**Sol. (a)** Refer Section 8.5.2

**(b) C Program to reverse a linked list**

This C Program reverses the sequence elements in a linked list.

Here is a source code of the C Program to reverse a linked list. The C program is successfully compiled and run on a Linux system. The program output is also shown as follows:

```
/*
 * C Program to Reverse a Linked List
 */
#include <stdio.h>
#include <stdlib.h>

struct node
{
 int num;
 struct node *next;
};

void create(struct node **);
void reverse(struct node **);
void release(struct node **);
void display(struct node *);

int main()
{
 struct node *p = NULL;
 int n;

 printf("Enter data into the list\n");
 create(&p);
 printf("Displaying the nodes in the list:\n");
 display(p);
 printf("Reversing the list...\n");
 reverse(&p);
 printf("Displaying the reversed list:\n");
 display(p);
 release(&p);

 return 0;
}
```



```

void reverse(struct node **head)
{
 struct node *p, *q, *r;

 p = q = r = *head;
 p = p->next->next;
 q = q->next;
 r->next = NULL;
 q->next = r;

 while (p != NULL)
 {
 r = q;
 q = p;
 p = p->next;
 q->next = r;
 }
 *head = q;
}

void create(struct node **head)
{
 int c, ch;
 struct node *temp, *rear;

 do
 {
 printf("Enter number: ");
 scanf("%d", &c);
 temp = (struct node *)malloc(sizeof(struct node));
 temp->num = c;
 temp->next = NULL;
 if (*head == NULL)
 {
 *head = temp;
 }
 else
 {
 rear->next = temp;
 }
 rear = temp;
 printf("Do you wish to continue [1/0]: ");
 scanf("%d", &ch);
 } while (ch != 0);
 printf("\n");
}

void display(struct node *p)
{
 while (p != NULL)

```



**SQP12** *Computer Programming*

```
 {
 printf("%d\t", p->num);
 p = p->next;
 }
 printf("\n");
 }

void release(struct node **head)
{
 struct node *temp = *head;
 *head = (*head)->next;
 while ((*head) != NULL)
 {
 free(temp);
 temp = *head;
 (*head) = (*head)->next;
 }
}
```

**7. What is file? Explain about different file operations with example.**

**[16]**

**Sol.** Refer Sections 9.1, 9.2, 9.3, and 9.4



---

# Solved Question Paper

## Nov-Dec 2015 Set 3

---

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supple. Examinations Nov/Dec - 2015

### COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem. E, Aero E, AME, Min E, PE, Metal E, Textile Engg.)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**  
Answering the question in **Part-A** is Compulsory,  
Three Questions should be answered from **Part-B**

\*\*\*\*\*

### PART-A

1. (a) What is compilation? Is it different from the compiler?
- (b) What is the importance of break in switch statement? If break is not given what happens? Explain with example.
- (c) What is the importance of #define? explain
- (d) What is pointer? Is \*p is similar to \*\*p? explain.
- (e) Discuss about bit fields.
- (f) What are different types of files? Explain. [3+4+3+4+4+4]

**Sol. (a) Program Development in C**

The steps that need to be followed for compilation and execution of a C program are listed below.  
Load one of the compilers, or better an IDE (integrated development environment), in the computer.  
According to the instruction of the compiler, open it for typing the text of the program.  
Click or give command for compiling the program. It will indicate errors if any.  
Correct the indicated errors and again click or give command for compiling.  
If it is indicated that there are no more errors in the text, click or give command to run /execute the program.

The various processes that takes place in an IDE are internal to the software. The visible components are errors that occur during compiling and also during the running.

- (b) If you don't use a break statement, the program will just continue into the next case block, sometimes this behavior is desirable, usually it's not. Here's an example where it would be:

```
int days;
int year = rand() % 100 + 2000; // select a random year
int month = rand() % 12 + 1; // select a random month

switch(month)
{
case 2:
```



## SQP14 Computer Programming

```
days = 28;
if (year % 4 == 0)
days = 29;
break;

case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
days = 31;
break; // need break here, otherwise days will be set to 30

default:
days = 30; // no need to break here, switch block is done.
}
```

### (c) Preprocessor Directives in C

There are a number of predefined preprocessor directives in C standard library which may be used by a programmer.

#### Applications of Some Preprocessor Directives

# define and #undef

The directive #define is used to create symbolic constants and macros (small function type entities). The directive #define may be used in the following manner:

```
#define Identifier Replacement_text
```

Note that there is no semicolon at the end of line and there is at least one space between define and Identifier, and between Identifier and Replacement\_text. During the preprocessor action whenever the identifier occurs in the program listing it is replaced with replacement\_text. For instance:

```
#define X 7.86
```

In this code, X is defined to be equal to constant value 7.86. Wherever X occurs in the source code, it is replaced by 7.86. The above expression is in fact equivalent to the following:

```
Const float X=7.86;
```

Once defined as above, the value of X cannot be changed in the program unless X is first undefined to shed the previous value and is then redefined to a new value. The code is written as given below:

```
#define X 7.86
#undef X
#define X 9.87
```

(d) Refer Section 7.3, 7.7

(e) Refer Section 8.9

(f) Refer SQP Nov/Dec 2015, Set 1, Q 1 (f)

## PART-B

2. (a) What is algorithm? What are the criteria for an algorithm?

(b) Explain about relational operators.

(c) Write an algorithm for sum of digits in a given number.

[5+5+6]



**Sol. (a) Refer Section 1.6**

**(b) Refer Section 2.9.2**

**(c)** Step 1: Start  
 Step 2: Input N  
 Step 3: Sum=0  
 Step 4: While(N!=0)  
     Rem=N%10;  
     Sum=Sum+Rem;  
     N=N/10;  
 Step 5: Print Sum  
 Step 6: Stop

**3. (a) Explain about the selection statements with examples.**

**(b) Write a program for sum of series.**

**[8+8]**

**Sol. (a) Refer Section 3.2**

**(b)** `#include<stdio.h>`  
`int main(){`  
  
`int n,i;`  
`int sum=0;`  
  
`printf("Enter the n i.e. max values of series: ");`  
`scanf("%d",&n);`  
  
`sum = (n * (n + 1)) / 2;`  
  
`printf("Sum of the series: ");`  
  
`for(i =1;i <= n;i++){`  
`if (i!=n)`  
`printf("%d + ",i);`  
`else`  
`printf("%d = %d ",i,sum);`  
`}`  
  
`return 0;`  
`}`

**4. (a) What is recursion? How it is different from the normal function.**

**(b) What are the importance of the extern and register storage classes.**

**(c) Write a recursion program for the factorial function**

**[4+6+6]**

**Sol. (a) Refer Section 4.10**

**(b) Refer Section 4.13.2, 4.13.5**

**(c)** `#include<stdio.h>`  
`int fact(int);`  
`void main()`  
`{`



```

 int num;
 printf("Enter the number to calculate Factorial:");
 scanf("%d",&num);
 printf("\n Factorial:%d,fact(num));
 }
 Int fact(int n)
 {
 If(n==1)
 Return 1;
 Else
 Return n* fact(n-1);
 }

```

**5. (a) How variables are passed to functions using pointers? Discuss.**

**(b) Write a program for matrix multiplication.**

**[8+8]**

**Sol. (a)** C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```

#include <stdio.h>
#include <time.h>
void getSeconds(unsigned long *par);
int main () {
 unsigned long sec;
 getSeconds(&sec);
 /* print the actual value */
 printf("Number of seconds: %ld\n", sec);
 return 0;
}
void getSeconds(unsigned long *par) {
 /* get the current number of seconds */
 *par = time(NULL);
 return;
}

```

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

Number of seconds :1294450468

**(b) Refer Example 5.9**

**6. Explain about different bit wise operators with examples.**

**[16]**

**Sol.** Refer Section 2.9.7

**7. (a) Explain about the fprintf and fscanf functions with examples.**

**(b) Write a program to copy one file contents into another file.**

**[8+8]**

**Sol. (a)** Refer Section 9.4.3

**(b)** Refer Example 8.22



---

# Solved Question Paper

## Nov-Dec 2015 Set 4

---

Subject Code: R13105/R13

I B. Tech I Semester Regular/Supple. Examinations Nov/Dec - 2015

### COMPUTER PROGRAMMING

(Common to CE, ME, CSE, PCE, IT, Chem. E, Aero E, AME, Min E, PE, Metal E, Textile Engg.)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**  
Answering the question in **Part-A** is Compulsory,  
Three Questions should be answered from **Part-B**

\*\*\*\*\*

### PART-A

1. (a) What is interpreter? How to compile a C program?
- (b) Discuss about conditional operator.
- (c) What is the importance of #typedef? explain
- (d) Discuss about character pointer.
- (e) Explain the importance of the bit-wise operators.
- (f) Explain about the functions for reading data from a file.

[3+4+3+4+4+4]

Sol. (a) **Interpreter**

- Interpreter takes single instruction as input.
- No intermediate object code is generated
- Conditional control statements are executed slower
- Memory requirement is less every time higher level program is converted into lower level program
- Errors are displayed for every instruction interpreted (if any)
- The interpreter can immediately execute high-level programs, thus interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly.
- In addition, interpreters are often used in education because they allow students to program interactively.

### Compilation

There are many C compilers around. The cc being the default Sun compiler. The GNU C compiler gcc is popular and available for many platforms. PC users may also be familiar with the Borland bcc compiler.

There are also equivalent C++ compilers which are usually denoted by CC (note upper case CC. For example Sun provides CC and GNU GCC. The GNU compiler is also denoted by g++

Other (less common) C/C++ compilers exist. All the above compilers operate in essentially the same manner and share many common command line options. Below and in Appendix \_\_\_\_\_



we list and give example uses many of the common compiler options. However, the best source of each compiler is through the online manual pages of your system: e.g. `man cc`.

For the sake of compactness in the basic discussions of compiler operation we will simply refer to the `cc` compiler—other compilers can simply be substituted in place of `cc` unless otherwise stated.

To compile your program simply invoke the command `cc`. The command must be followed by the name of the (C) program you wish to compile. A number of compiler options can be specified also. Thus, the basic compilation command is:

```
cc program.c
```

where `program.c` is the name of the file.

If there are obvious errors in your program (such as mistyping, misspelling one of the key words or omitting a semi-colon), the compiler will detect and report them.

There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations.

When the compiler has successfully digested your program, the compiled version, or executable, is left in a file called `a.out` or if the compiler option `-o` is used: the file listed after the `-o`.

It is more convenient to use a `-o` and filename in the compilation as in

```
cc -o program program.c
```

which puts the compiled program into the file `program` (or any file you name following the “`-o`” argument) instead of putting it in the file `a.out`.

- (b) Refer Section 2.9.6
- (c) Refer Section 2.12.2, 8.10
- (d) The string literal merely sets the initial contents of the array. In the definition, `char*pmessag = “now is the time”`; on the other hand, the string literal is used to create a little block of characters somewhere in memory which the pointer `pmessag` is initialized to point to.
- (e) Refer Section 2.9.7
- (f) Refer Section 9.4

## PART-B

2. (a) Explain about the program development steps.

(b) Write an algorithm for generating Fibonacci series.

[8+8]

**Sol.** (a) The development of a C program involves the use of the following programs in the order of their usage.

### Editor

This program is used for writing the source code, the first thing that any programmer writing a program in any language would be doing.

### Debugger

This program helps us identify syntax errors in the source code.

### Pre Processor

There are certain special instructions within the source code identified by the `#` symbol that are carried on by a special program called a preprocessor.



**Compiler**

The process of converting the C source code to machine code and is done by a program called Compiler.

**Linker**

The machine code relating to the source code you have written is combined with some other machine code to derive the complete program in an executable file. This is done by a program called the linker.

**(b) Fibonacci series algorithm:**

```

Start
Declare variables i, a,b , show
Initialize the variables, a=0, b=1, and show =0
Enter the number of terms of Fibonacci series to be printed
Print First two terms of series
Use loop for the following steps
-> show=a+b
-> a=b
-> b=show
-> increase value of i each time by 1
-> print the value of show
End

```

**3. (a) How strings are represented using strings? Explain about different string manipulations.****(b) Write a program to calculate the following series:  $1+1/2+1/3+1/4+1/5+....$  [8+8]**

**Sol. (a)** Ref Section 6.3 , 6.8

```

(b) #include<stdio.h>
#include<conio.h>
void main()
{
 double n,sum=0,i;
 clrscr();

 printf("\n Please Give The Value of N: ");
 scanf("%lf",&n);
 for(i=1;i<=n;i++)
 {
 sum = sum + (1/i);
 if(i==1)
 printf("\n 1 +");
 elseif(i==n)
 printf(" (1/%d) ",i);
 else
 printf(" (1/%d) + ",i);
 }
 printf("\n\n THE SUM OF THIS SERIES IS %.2lf",sum);
 getch();
}

```



**4. Explain about different parameter passing algorithms with examples.****[16]****Sol.** Refer Section 4.12.1**Passing Argument to Function:**

In C Programming we have different ways of parameter passing schemes such as Call by Value and Call by Reference.

Function is good programming style in which we can write reusable code that can be called whenever require.

Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called argument.

**Two Ways of Passing Argument to Function in C Language:**

Call by Reference

Call by Value

Let us discuss different ways one by one:

**1. Call by Value:**

```
#include<stdio.h>

void interchange(int number1,int number2)
{
 int temp;
 temp = number1;
 number1 = number2;
 number2 = temp;
}

int main() {

 int num1=50,num2=70;
 interchange(num1,num2);

 printf("\nNumber 1 : %d",num1);
 printf("\nNumber 2 : %d",num2);

 return(0);
}
```

Output :

Number 1 : 50

Number 2 : 70

***Explanation: Call by Value***

While Passing Parameters using call by value, xerox copy of original parameter is created and passed to the called function.

Any update made inside method will not affect the original value of variable in calling function.

In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1, number2 variable of sum function respectively.

As their scope is limited to only function so they cannot alter the values inside main function.



**2. Call by Reference/Pointer/Address:**

```
#include<stdio.h>

void interchange(int *num1,int *num2)
{
 int temp;
 temp = *num1;
 *num1 = *num2;
 *num2 = temp;
}

int main() {

 int num1=50,num2=70;
 interchange(&num1,&num2);

 printf("\nNumber 1 : %d",num1);
 printf("\nNumber 2 : %d",num2);

 return(0);
}
```

Output :  
 Number 1 : 70  
 Number 2 : 50

*Explanation : Call by Address*

While passing parameter using call by address scheme , we are passing the actual address of the variable to the called function.

Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location.

**Summary of Call By Value and Call By Reference:**

| Point        | Call by Value                                                         | Call by Reference                                                              |
|--------------|-----------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Copy         | Duplicate Copy of Original Parameter is Passed                        | Actual Copy of Original Parameter is Passed                                    |
| Modification | No effect on Original Parameter after modifying parameter in function | Original Parameter gets affected if value of parameter changed inside function |

**5. (a) Explain about the functions which are used for allocating memory dynamically.**

**(b) Write a program for calculating the sum of values in a given array using pointers. [8+8]**

**Sol. (a)** Refer Sections 7.16, 7.17, 7.18, 7.19

**(b)** Refer Example 7.8

**6. (a) What is meant by self-referential structures? Give examples.**

**(b) Write a program for creating a linked list.**

**[8+8]**



**SQP22** *Computer Programming*

- Sol.** (a) Refer Section 8.6.2  
(b) Refer SQP May 2016, Set 4, Q 6 (b).

- 7. (a) Explain about the fopen and fclose functions with examples.**  
**(b) Write a program to merge two files into single file.**

**[8+8]**

- Sol.** (a) Refer Section 9.2, 9.3  
(b) Refer Example 8.27



---

# Solved Question Paper

## May 2016 Set 1

---

Subject Code: R13205/R13

I.B.Tech II Semester Regular/Supplementary Examinations May - 2016

COMPUTER PROGRAMMING

(Common to ECE, EEE, EIE, Bio-Tech, E Com E, Agri E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**  
Answering the question in **Part-A** is Compulsory,  
Three Questions should be answered from **Part-B**

### PART-A

1. (a) Define high level language and low level language.
- (b) Explain about the break and continue with example.
- (c) Define function.
- (d) What is dangling memory?
- (e) What are the applications of structures?
- (f) Differentiate between binary file and text file.

[3 + 4 + 3 + 4 + 4 + 4]

Sol. (a) Refer Section 1.5

(b) Refer Section 3.10.1, Example 3.21

With “continue;” it is possible to skip the rest of the commands in the current loop and start from the top again. (the loop variable must still be incremented). Take a look at the example below:

```
#include<stdio.h>

int main()
{
 int i;

 i = 0;
 while (i < 20)
 {
 i++;
 continue;
 printf("Nothing to see\n");
 }
 return 0;
}
```

(c) Refer Section 2.2

(d) Dangling pointers in computer programming are pointers that do not point to a valid object of the appropriate type. These are special cases of memory safety violations. More generally,



dangling references and wild references are references that do not resolve to a valid destination, and include such phenomena as link rot on the internet.

Dangling pointers arise during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. As the system may reallocate the previously freed memory to another process, if the original program then dereferences the (now) dangling pointer, unpredictable behavior may result, as the memory may now contain completely different data. This is especially the case if the program writes data to memory pointed to by a dangling pointer, a silent corruption of unrelated data may result, leading to subtle bugs that can be extremely difficult to find, or cause segmentation faults (UNIX, Linux) or general protection faults (Windows). If the overwritten data is book-keeping data used by the system's memory allocator, the corruption can cause system instabilities. In object-oriented languages with garbage collection, dangling references are prevented by only destroying objects that are unreachable, meaning they do not have any incoming pointers; this is ensured either by tracing or reference counting. However, a finalizer may create new references to an object, requiring object resurrection to prevent a dangling reference.

- (e) Refer Section 8.6
- (f) All files can be categorized into one of two file formats—binary or text. The two file types may look the same on the surface, but they encode data differently. While both binary and text files contain data stored as a series of bits (binary values of 1s and 0s), the bits in text files represent characters, while the bits in binary files represent custom data.

**\*\*While text files contain only textual data, binary files may contain both textual and custom binary data.\*\***

### **Binary Files**

Binary files typically contain a sequence of bytes, or ordered groupings of eight bits. When creating a custom file format for a program, a developer arranges these bytes into a format that stores the necessary information for the application. Binary file formats may include multiple types of data in the same file, such as image, video, and audio data. This data can be interpreted by supporting programs, but will show up as garbled text in a text editor.

For example if a .PNG image file is opened in an image viewer and a text editor then the image viewer recognizes the binary data and displays the picture. When the image is opened in a text editor, the binary data is converted to unrecognizable text. However, some of the text is readable. This is because the PNG format includes small sections for storing textual data. The text editor, while not designed to read this file format, still displays this text when the file is opened. Many other binary file types include sections of readable text as well. Therefore, it may be possible to find out some information about an unknown binary file type by opening it in a text editor.

Binary files often contain headers, which are bytes of data at the beginning of a file that identifies the file's contents. Headers often include the file type and other descriptive information. For example, in the image above, the "PNG" text indicates the file is a PNG image. If a file has invalid header information, software programs may not open the file or they may report that the file is corrupted.

### **Text Files**

Text files are more restrictive than binary files since they can only contain textual data. However, unlike binary files, they are less likely to become corrupted. While a small error in a binary file may make it unreadable, a small error in a text file may simply show up once the file has been



opened. This is one of the reasons Microsoft switched to a compressed text-based XML format for the Office 2007 file types.

Text files may be saved in either a plain text (.TXT) format or rich text (.RTF) format. A typical plain text file contains several lines of text that are each followed by an End-of-Line (EOL) character. An End-of-File (EOF) marker is placed after the final character, which signals the end of the file. Rich text files use a similar file structure, but may also include text styles, such as bold and italics, as well as page formatting information. Both plain text and rich text files include a (character encoding| characterencoding) scheme that determines how the characters are interpreted and what characters can be displayed.

Since text files use a simple, standard format, many programs are capable of reading and editing text files. Common text editors include Microsoft Notepad and WordPad, which are bundled with Windows, and Apple TextEdit, which is included with Mac OS X.

## PART-B

2. (a) **What is the difference between the post increment and pre increment operations? What is the output of the following if  $k = 5$ ,**

$i = ++k, \quad j = k++, \quad k++, \quad ++k;$

- (b) **What are variables and constants? What are the rules for declaring the variables? [8 + 8]**

**Sol.** (a) Refer Section 2.9.5

(b) Refer Sections 2.10, 2.11, 2.12

3. (a) **Explain about different string functions which can be performed on strings.**

- (b) **Write an algorithm to find whether the given string is palindrome or not.**

- (c) **Write a program for adding two matrices.**

**[3 + 8 + 5]**

**Sol.** (a) Refer Section 6.8

(b) Refer Example 1.5

(c) 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++)
```



## SQP4 Computer Programming

```
{
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*/
}
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

```

**4. Explain about different parameter passing mechanisms with examples.**

**[16]**

**Sol.** Refer Section 7.10

**5. (a) What is pointer and indirect pointer? Explain with examples.**

**(b) Write a program to swap two numbers using pointers.**

**[8 + 8]**

**Sol. (a)** Refer Sections 7.2 and 7.6

```

(b) #include <stdio.h>
int main()
{
 int x, y, *a, *b, temp;

 printf("Enter the value of x and y\n");
 scanf("%d%d", &x, &y);

 printf("Before Swapping\nx = %d\ny = %d\n", x, y);

 a = &x;
 b = &y;

 temp = *b;
 *b = *a;
 *a = temp;

 printf("After Swapping\nx = %d\ny = %d\n", x, y);

 return 0;
}

```

**6. (a) Explain about the logical and shift operators with examples.**

**(b) Write a program to sort the elements in a linked list.**

**[8 + 8]**

**Sol. (a)** Refer Sections 2.9.3 and 2.9.7

The bitwise shift operators are the right-shift operator (>>), which moves the bits of shift\_expression to the right, and the left-shift operator (<<), which moves the bits of shift\_expression to the left.

Syntax:

```

shift-expression << additive-expression
shift-expression >> additive-expression

```



**Bitwise Left Shift Operator**

1. It is denoted by <<
2. Bit pattern of the data can be shifted by specified number of positions to left
3. When data is shifted left, trailing zeros are filled with zero.
4. Left shift operator is binary operator [Bi: two]
5. Binary means operator that require two arguments

|                   |                                  |
|-------------------|----------------------------------|
| Original Number A | 0000 0000 0011 1100              |
| Left Shift        | 0000 0000 1111 0000              |
| Trailing Zero's   | Replaced by 0<br>(Shown in Bold) |

Direction of Movement of Data      <<<<=====Left

Program

```
#include<stdio.h>

int main()
{
 int a = 60;

 printf("\nNumber is Shifted By 1 Bit : %d",a << 1);
 printf("\nNumber is Shifted By 2 Bits : %d",a << 2);
 printf("\nNumber is Shifted By 3 Bits : %d",a << 3);

 return(0);
}
```

Output

```
Number is Shifted By 1 Bit : 120
Number is Shifted By 2 Bits : 240
Number is Shifted By 3 Bits : 480
```

**Bitwise Right Shift Operator**

1. It is denoted by >>
2. Bit pattern of the data can be shifted by specified number of positions to right
3. When data is shifted right, leading zeros are filled with zero.
4. Right shift operator is binary operator [Bi: two]
5. Binary means operator that require two arguments

|                               |                              |
|-------------------------------|------------------------------|
| Original Number A             | 0000 0000 0011 1100          |
| Right Shift by 2              | 0000 0000 0000 1111          |
| Leading 2 Blanks              | Replaced by 0 ,Shown in Bold |
| Direction of Movement of Data | Right =====>>>>>             |

(b)

```
Program
#include <stdio.h>
#include <stdlib.h>
#define NULL 0
```



```

struct linked_list
{
 int number;
 struct linked_list *next;
};
typedef struct linked_list node;

main ()
{
 int n;
 node *head = NULL;
 void print(node *p);
 node *insert_Sort(node *p, int n);

 printf("Input the list of numbers.\n");
 printf("At end, type -999.\n");
 scanf("%d",&n);

 while(n != -999)
 {
 if(head == NULL) /*create 'base' node*/
 {
 head = (node *)malloc(sizeof(node));
 head ->number = n;
 head->next = NULL;

 }

 else /*insert next item*/
 {
 head = insert_sort(head,n);
 }

 scanf("%d", &n);
 }
 printf("\n");
 print(head);
 print("\n");
}

node *insert_sort(node *list, int x)
{
 node *p1, *p2, *p;
 p1 = NULL;
 p2 = list; /*p2 points to first node*/

 for(; p2->number < x ; p2 = p2->next)
 {
 p1 = p2;
 }

```



```

 if(p2->next == NULL)
 {
 p2 = p2->next; /*p2 set to NULL*/
 break; /*insert new node at end*/
 }

 /*key node found*/
 p = (node *)malloc(sizeof(node)); /*space for new node*/
 p->number = x; /*place value in the new node*/
 p->next = p2; /*link new node to key node*/
 if (p1 == NULL)
 list = p; /*new node becomes the first node*/
 else
 p1->next = p; /*new node inserted after 1st node*/
 return (list);
}

void print(node *list)
{
 if (list == NULL)
 printf("NULL");
 else
 {
 printf("%d-->",list->number);
 print(list->next);
 }
 return;
}

```

Output

```

Input the list of number.
At end, type -999.
80 70 50 40 60 -999
40-->50-->60-->70-->80 -->NULL
Input the list of number.
At end, type -999.
40 70 50 60 80 -999
40-->50-->60-->70-->80-->NULL

```

**7. (a) Explain about different file operations that can be performed on files.**

**(b) Write a program to copy one file into another file.**

**[8 + 8]**

**Sol. (a)** Refer Sections 9.2 to 9.5

**(b)** Program

```

#include<stdio.h>
#include<process.h>
void main() {

```



```
FILE *fp1, *fp2;
char a;
clrscr();
fp1 = fopen("test.txt", "r");
if (fp1 == NULL) {
 puts("cannot open this file");
 exit(1);
}
fp2 = fopen("test1.txt", "w");
if (fp2 == NULL) {
 puts("Not able to open this file");
 fclose(fp1);
 exit(1);
}
do {
 a = fgetc(fp1);
 fputc(a, fp2);
} while (a != EOF);

fcloseall();
getch();
}
```

Output

Content will be written successfully to file



---

# Solved Question Paper

## May 2016 Set 2

---

Subject Code: R13205/R13

I.B.Tech II Semester Regular/Supplementary Examinations May - 2016

COMPUTER PROGRAMMING

(Common to ECE, EEE, EIE, Bio-Tech, E Com E, Agri E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**  
Answering the question in **Part-A** is Compulsory,  
Three Questions should be answered from **Part-B**

### PART-A

1. (a) Differentiate between compiler and interpreter.
- (b) What is nested if? Give examples.
- (c) Write short notes on standard library functions.
- (d) What are command line arguments?
- (e) Define nested structures.
- (f) How to read data from and write data to a file?

[3 + 4 + 3 + 4 + 4 + 4]

Sol. (a) The differences between an interpreter and a compiler are as follows:

| Interpreter                                                                                                       | Compiler                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| Translates program statement by statement at any given time.                                                      | Scans the entire program and translates it as a whole into machine code.                                         |
| It takes lesser time to analyze the source code but the overall execution time is slower.                         | It takes large amount of time to analyze the source code but the overall execution time is comparatively faster. |
| No intermediate object code is generated, hence it is memory efficient.                                           | Generates intermediate object code which further requires linking, hence requires more memory.                   |
| Continues translating the program until the first error is met, in which case it stops. Hence, debugging is easy. | It generates the error message only after scanning the whole program. Hence, debugging is comparatively hard.    |
| Programming languages like Python, Ruby use interpreters.                                                         | Programming languages like C, C++ use compilers.                                                                 |

- (b) Refer Section 3.2.3
- (c) Refer Section 2.13
- (d) Refer Section 8.11



- (e) Refer Section 8.5.2  
 (f) Refer Section 9.4

### PART-B

2. (a) Explain about relational and logical operations with examples.  
 (b) Write a program for printing the following pattern on the screen.

```

1
1 2
1 2 3
1 2 3 4

```

**[8 + 8]**

**Sol.** (a) Refer 2.9.2, 2.9.3

(b) 

```
#include <stdio.h>
int main()
{
 int i, j, n;

 printf("Enter the number of rows: ");
 scanf("%d", &n);

 for(i=1; i<=n; ++i)
 {
 for(j=1; j<=i; ++j)
 {
 printf("%d",j);
 }
 printf("\n");
 }
 return 0;
}
```

3. (a) Differentiate between switch and else-if.  
 (b) Write a program to find smallest and largest numbers in a given array.  
 (c) Write a program to multiply two matrices.

**[3 + 8 + 5]**

**Sol.** (a) Refer Section 3.2.2

(b) Program

```
#include <stdio.h>
#define n 11
main()
{
 int i, j, m, k, max, min ;
 int Array[n];

 printf("Enter %d integers:", n);
 for(i=0; i<n; i++)
```



## SQP12 Computer Programming

```
scanf("%d", & Array [i]);
printf("\nyou have entered the following numbers: \n");
for(j =0;j<n; j++)
printf("%d ", Array [j]);
printf("\n");

max = Array [0];
{for (k =0; k<n; k++)
if (Array [k]>max)
max= Array [k];
else max = max;}

printf("Maximum number is = %d\n",max);
min = Array [0];
{for(m =0;m<n;m++)
if(Array [m]<min)
min= Array [m];
else min = min;}
printf("Minimum number = %d\n", min);
}
```

Output

```
Enter 11 integers: 43 54 6 37 2 67 84 43 80 95 4
you have entered the following numbers:
43 54 7 37 2 67 84 43 80 95 4
Maximum number is = 95
Minimum number = 2
```

(c) #include<stdio.h>  
#include<stdlib.h>  
int main(void)  
{  
int a[10][10],b[10][10],c[10][10],n=0,m=0,i=0,j=0,p=0,q=0,k=0;  
int \*pt,\*pt1,\*pt2;  
printf("Enter size of 1st 2d array : ");  
scanf("%d %d",&n,&m);  
for(i=0;i<n;i++)  
{  
for(j=0;j<m;j++)  
{  
printf("Enter element no. %d %d :",i,j);  
scanf("%d",&a[i][j]);  
}  
}  
printf("Enter size of 2nd 2d array : ");  
scanf("%d %d",&p,&q);  
for(i=0;i<p;i++)  
{



```

for(j=0;j<q;j++)
{
printf("Enter element no. %d %d :",i,j);
scanf("%d",&b[i][j]);
}
}
if(m!=p)
{
printf("Multiplication cannot be done\n");
exit (0);
}
pt=&a[0][0];
pt1=&b[0][0];
pt2=&c[0][0];
for(i=0;i<n;i++)
{
for(k=0;k<q;k++)
{
*(pt2+(i*10+k))=0;
for(j=0;j<m;j++)
{
*(pt2+(i*10+k))+=*(pt+(i*10+j))**(pt1+(j*10+k));
}
}
}
for(i=0;i<n;i++)
{
for(j=0;j<q;j++)
{
printf("%d ",c[i][j]);
}
printf("\n");
}
return 0;
}

```

**4. Explain about different storage classes with examples.****[16]****Sol.** Sections 4.12.1, 7.10**5. (a) Discuss about character pointers with examples.****(b) Write a program to print command line arguments on the screen.****[8 + 8]****Sol. (a) Character Pointer:**

Character Pointer is used to access character array.

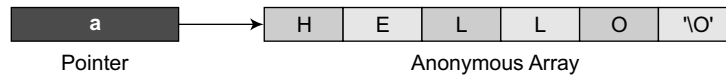
Syntax: char \*a

Example: char \*a="HELLO";

String "Hello" will be stored at any anonymous location in the form of array. We even don't know the location where we have stored string, however, string will have its starting address.



The above syntax will take following form:



We have declared pointer of type character i.e. pointer variable is able to hold the address of character variable. Now base address of anonymous array is stored in character pointer variable. 'a' stores base address of the anonymous array [**Unknown Array**].

Address = [Base Address of Anonymous Array] + [i]

#### Accessing Individual Element:

Consider we have to access `a[3]` then, in short, if 'a' is a pointer, it starts at the location "a", gets the pointer value there, adds 3 to the pointer value, and gets the character pointed to by that value [box].

```

Program
void main()
{
 char *a="hello";
 clrscr();
 printf("%c\n",a[3]);
 printf("%s",a);
 getch();
}
Output
l
hello

```

6. (a) How to declare a structure and initialize values for structure members?

(b) Write a program to insert an element into a linked list at front.

[8 + 8]

Sol. (a) Refer Sections 8.3 and 8.4

(b)

```

node *insert(node *head)
{
 node *find(node *p, int a);
 node *new; /*pointer to new node*/
 node *n1; /*pointer to node preceding key node*/
 int key;
 int x; /*new item (number) to be inserted*/

 printf("Value of new item?");
 scanf("%d", &x);
 printf("Value of key item ? (type -999 if last) ");
 scanf("%d", &key);
 if(head->number == key) /*new node is first*/
 {
 new = (node *)malloc(size of(node));

```



```

 new->number = x;
 new->next = head;
 head = new;
 }
 else /*find key node and insert new node*/
 {
 /*before the key node*/
 n1 = find(head, key); /*find key node*/

 if(n1 == NULL)
 printf("\n key is not found \n");
 else /*insert new node*/
 {
 new = (node *)malloc(sizeof(node));
 new->number = x;
 new->next = n1->next;
 n1->next = new;
 }
 }
 return(head);
}
node *find(node *lists, int key)
{
 if(list->next->number == key) /*key found*/
 return(list);
 else

 if(list->next->next == NULL) /*end*/
 return(NULL);
 else
 find(list->next, key);
}

```

**7. (a) Explain about the input and output operations of a file.**

**(b) Write a program to open a file and to print its contents on screen.**

**[8 + 8]**

**Sol. (a) Refer Section 9.4**

**(b) Program**

```

#include <stdio.h>
#include <stdlib.h>
main()
{
 FILE* fptr;
 char ch;
 fptr = fopen ("Myfile", "r");
 if(fptr == NULL)
 {printf("File could not be opened");
 exit(1);}
 else

```



## SQP16 *Computer Programming*

```
{printf("File is open for reading.\n");
while ((ch = fgetc (fptr)) != EOF) //read character by character
if(ch == 'n')
{ungetc(ch, fptr); //if ch =='n' push it back.
 printf("%c", ch); // display the character
 fclose(fptr); }
 printf("\n");}
}
```

The expected output as is as given below. The function `ungetc()` retrieves the character 'n' while reading the file created in Program.

```
File is open for reading.
n
```



---

# Solved Question Paper

## May 2016 Set 3

---

Subject Code: R13205/R13

I.B.Tech II Semester Regular/Supplementary Examinations May - 2016

COMPUTER PROGRAMMING

(Common to ECE, EEE, EIE, Bio-Tech, E Com E, Agri E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**  
Answering the question in **Part-A** is Compulsory,  
Three Questions should be answered from **Part-B**

### PART-A

1. (a) Differentiate between compiling and linking.
- (b) Write a program to find the  $2^n$  value.
- (c) What is preprocessor?
- (d) How to pass pointer variables to functions?
- (e) What is the importance of typedef? Explain.
- (f) What are different types of files? Explain.

[3 + 4 + 3 + 4 + 4 + 4]

Sol. (a) Refer Section 2.4

```
(b) #include<stdio.h>
#include<math.h>
void main()
{
 int a,b,c,val,result;
 clrscr();
 printf("Enter 2 numbers");
 scanf("%d%d",&a,&b);
 result=pow(a,b);
 printf("%d"result);
 getch();
}
```

(c) **Preprocessor Directives in C**

There are a number of predefined preprocessor directives in C standard library which may be used by a programmer.

#### Applications of Some Preprocessor Directives

# define and #undef

The directive #define is used to create symbolic constants and macros (small function type entities).The directive #define may be used in the following manner:

```
#define Identifier Replacement_text
```



Note that there is no semicolon at the end of line and there is at least one space between define and Identifier, and between Identifier and Replacement\_text. During the preprocessor action whenever the identifier occurs in the program listing it is replaced with replacement\_text. For instance:

```
#define X 7.86
```

In this code, X is defined to be equal to constant value 7.86. Wherever X occurs in the source code, it is replaced by 7.86. The above expression is infact equivalent to the following:

```
Const float X=7.86;
```

Once defined as above ,the value of X cannot be changed in the program unless X is first undefined to shed the previous value and is then redefined to a new value. The code is written as given below:

```
#define X 7.86
#undef X
#define X 9.87
```

- (d) Refer Section 7.10
- (e) Refer Section 8.10
- (f) Refer SQP May 2016, Set 1, Q 1(f)

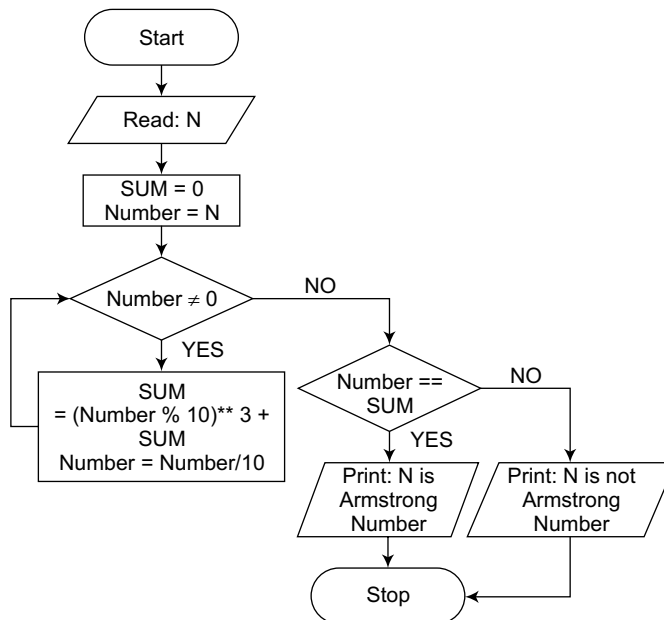
## PART-B

2. (a) Explain about arithmetic and relational operations with examples.  
 (b) Draw the flowchart for finding whether the given no. is Armstrong no. or not.

[8 + 8]

**Sol.** (a) Refer Sections 2.9.1 and 2.9.2

(b)





3. (a) What is matrix? How arrays are used for declaring the matrix?  
 (b) What is multi-dimensional array? How to access multi-dimensional arrays?  
 (c) Write a program to print the following matrix

```
[A B C D
E F G H
I J K L
M N O P]
```

**Sol.** (a) Refer Section 5.5

(b) Refer Section 5.7

```
(c) #include<stdio.h>
#include<conio.h>
void main()
{
 int a[4][4],i,j,n=65;
 clrscr();
 for(i=0;i<4;i++)
 {
 for(j=0;j<4;j++)
 {
 printf("%c",n++);
 }
 printf("\n");
 }
}
```

4. (a) How to declare a function and differentiate calling and called function? Explain with an example program.  
 (b) Write a recursive program for finding the  $n$ th Fibonacci value, using functions. [8 + 8]

**Sol.** (a) Refer Section 4.9

The function which calls another function is known as calling function, while the function that is called is known as called function.

```
(b) #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();
}
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*/
}

```

**5. What is dynamic memory allocation? Explain with example.**

**[16]**

**Sol.** Refer Section 7.16

**6. (a) Explain the differences between structure and union and write a program to find sum of marks in three subjects for a student using structures.**

**(b) Explain in short different bit-wise operators with example.**

**[8 + 8]**

**Sol.** (a) Differences between structures and unions are as follows:

| Structure                                                                                         | Union                                                                                                                                 |
|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| A structure is defined with 'struct' keyword.                                                     | A union is defined with 'union' keyword.                                                                                              |
| The size of a structure object is equal to the sum of the individual sizes of the member objects. | The size of a union object is equal to the size of largest member object.                                                             |
| Structures are not considered as memory efficient in comparison to unions.                        | Unions are considered as memory efficient particularly in situations when the members are not required to be accessed simultaneously. |
| <b>Example:</b>                                                                                   | <b>Example:</b>                                                                                                                       |
| <pre> struct book { char   title[25]; char   author[25]; int    pages; float  price; }; </pre>    | <pre> union result { int    marks; char   grade; float  percent; }; </pre>                                                            |

```

#include<stdio.h>
struct student
{
int s1, s2, s3;
};
int main()
{
struct student st[20];
int n, i, total;
float av;
clrscr();

```



```

printf("\n Enter the number of student:");
scanf("%d", &n);
for(i=1;i<=n;i++)
{
printf("enter the mark of three sub.\n");
total=0;
scanf("%d%d%d",&st[i].s1,&st[i].s2,&st[i].s3);
total=st[i].s1+st[i].s2+st[i].s3;
printf("\n Total marks of student is=%d",total);
}
return 0
}

```

(b) Refer Section 2.9.7

**7. (a) Write short notes in formatted I/O.**

**(b) Write a program to merge two files into one file.**

**[8 + 8]**

**Sol. (a) Refer Sections 2.14.3 and 2.14.5**

**(b)** #include <stdio.h>  
#include <stdlib.h>

```

int main()
{
 FILE *fs1, *fs2, *ft;

 char ch, file1[20], file2[20], file3[20];

 printf("Enter name of first file\n");
 gets(file1);

 printf("Enter name of second file\n");
 gets(file2);

 printf("Enter name of file which will store contents of two files\n");
 gets(file3);

 fs1 = fopen(file1,"r");
 fs2 = fopen(file2,"r");

 if(fs1 == NULL || fs2 == NULL)
 {
 perror("Error ");
 printf("Press any key to exit...\n");
 getch();
 exit(EXIT_FAILURE);
 }

 ft = fopen(file3,"w");

 if(ft == NULL)

```



**SQP22** *Computer Programming*

```
{
 perror("Error ");
 printf("Press any key to exit...\n");
 exit(EXIT_FAILURE);
}

while((ch = fgetc(fs1)) != EOF)
 fputc(ch,ft);

while((ch = fgetc(fs2)) != EOF)
 fputc(ch,ft);

printf("Two files were merged into %s file successfully.\n",file3);

fclose(fs1);
fclose(fs2);
fclose(ft);

return 0;
}
```



---

# Solved Question Paper

## May 2016 Set 4

---

Subject Code: R13205/R13

I.B. Tech II Semester Regular/Supplementary Examinations May - 2016

COMPUTER PROGRAMMING

(Common to ECE, EEE, EIE, Bio-Tech, E Com E, Agri E)

Time: 3 hours

Max. Marks: 70

Question Paper Consists of **Part-A** and **Part-B**  
Answering the question in **Part-A** is Compulsory,  
Three Questions should be answered from **Part-B**

### PART-A

1. (a) Define algorithm and flowchart.
- (b) How to store data in arrays? Give examples.
- (c) What is recursion?
- (d) What are the applications of pointers?
- (e) Define self-referential structures.
- (f) What is file? How to open a file?

[3 + 4 + 3 + 4 + 4 + 4]

Sol. (a) **Features of Algorithm:**

The characteristics of a good algorithm are:

- *Precision*: The steps are precisely stated (defined)
- *Uniqueness*: Results of each step are uniquely defined and only depend on the input and the result of the preceding steps
- *Finiteness*: The algorithm stops after a finite number of instructions are executed.
- *Input*: The algorithm receives input
- *Output*: The algorithm produces output
- *Generality*: The algorithm applies to a set of inputs

**Features of Flowchart:**

- *Flowchart symbols*: A typical flowchart from older basic computer science textbooks may have the following kinds of symbols:
  - *Start and end symbols*: Represented as circles, ovals or rounded rectangles, usually containing the word “Start” or “End.”
  - *Arrows*: Showing “flow of control”. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to. The line for the arrow can be solid or dashed. The meaning of the arrow with dashed line may differ from one flowchart to another.



- *Generic processing steps*: Represented as rectangles. Examples: “Add 1 to X”; “replace identified part”; “save changes” or similar.
  - *Input/Output*: Represented as a parallelogram. Examples: Get X from the user; display X.
  - *Conditional or decision*: Represented as a diamond (rhombus) showing where a decision is necessary, commonly a Yes/No question or True/False test. The conditional symbol is peculiar as it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. The arrows should always be labeled. More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further or replaced with the “pre-defined process” symbol.
  - *Labeled connectors*: Represented by an identifying label inside a circle. Labeled connectors are used in complex or multi-sheet diagrams to substitute for arrows. For each label, the “outflow” connector must always be unique, but there may be any number of “inflow” connectors. In this case, a junction in control flow is implied.
- A flowchart provides an easy way of communication because any other person besides the programmer can understand the way they are represented.
  - A flowchart represents the data flow.
  - A flowchart provides a clear overview of the entire program and problem and solution.
  - A flowchart checks the accuracy in logic flow.
  - A flowchart documents the steps followed in an algorithm.
  - A flowchart provides the facility for coding.
  - A flowchart provides the way of modification of running program.
  - A flowchart shows all major elements and their relationship.
- (b) Refer Sections 5.2 and 5.3
- (c) Refer Section 4.10
- (d) Refer Section 7.1
- (e) Refer Section 8.6.2
- (f) Refer Section 9.2

## PART-B

2. (a) **What is data type? What is the size and range of each data type? Explain with examples.**

(b) **Write an algorithm for finding the biggest of three numbers.**

**[8 + 8]**

**Sol.** (a) Refer Section 1.4

(b) Step 1: Start

Step 2: Declare variables a, b and c.

Step 3: Read variables a, b and c.

Step 4: If a>b

If a>c

Display a is the largest number.

Else

Display c is the largest number.



```

Else
 If b>c
 Display b is the largest number.
Else
 Display c is the greatest number.
Step 5: Stop

```

**3. (a) What is character array? Give examples.**

**(b) Write a program to print Pascal triangle.**

**(c) Write an algorithm for finding whether the given no. is strong no. or not.**

**[3 +8 + 5]**

**Sol. (a)** Refer Sections 6.1 and 6.2

**(b)** #include <stdio.h>

```

long factorial(int);

int main()
{
 int i, n, c;

 printf("Enter the number of rows you wish to see in pascal triangle\n");
 scanf("%d",&n);

 for (i = 0; i < n; i++)
 {
 for (c = 0; c <= (n - i - 2); c++)
 printf(" ");

 for (c = 0 ; c <= i; c++)
 printf("%ld ",factorial(i)/(factorial(c)*factorial(i-c)));

 printf("\n");
 }

 return 0;
}

long factorial(int n)
{
 int c;
 long result = 1;

 for (c = 1; c <= n; c++)
 result = result*c;

 return result;
}

```



(c) Algorithm for Strong Number:

```

Step 1: Start
Step 2: Declare variables num, i, f, r, sum=0, temp.
Step 3: Enter value for num.
Step 4: Set: temp=num
Step 5: while (num>0)
{
 i=1, f=1;
 r=num%10;
 While (i<=r)
 {
 f=f*i;
 i=i+1;
 }
 sum=sum+f;
 num=num/10;
}
Step 6: [checking?]
If (sum==temp) then
 Print: "num" is a strong number.
Else
 Print: "num" is not a strong number.
Step 7: stop

```

**4. (a) Differentiate user defined and predefined function. Explain with one example.**

**(b) How to pass array variables to functions? Explain with examples.**

**[8 + 8]**

**Sol.** (a) Refer Sections 4.1, 4.2 and Page 2.6

(b) Refer Section 4.11

**5. (a) What is pointer? What are the advantages and disadvantages of pointers?**

**(b) What is formal and actual parameter? How to pass variables by their address?**

**[8 + 8]**

**Sol.** (a) Refer Sections 7.1, 7.2

(b) Refer Sections 4.11, 4.12

**6. (a) Differentiate between structures and unions, and write the syntax for nested structures.**

**(b) Write a program to create a linked list.**

**[8 + 8]**

**Sol.** (a) Refer SQP May 2016, Set 3, Q 6 (a)

For syntax, Refer Section 8.5.2

(b) The program in Fig. 1 creates a linear linked list interactively and prints out the list and the total number of items in the list.

The program first allocates a block of memory dynamically for the first node using the statement

```
head = (node *)malloc(sizeof(node));
```

which returns a pointer to a structure of type node that has been type defined earlier. The linked list is then created by the function create. The function requests for the number to be placed in the current node that has been created. If the value assigned to the current node is -999, then null



is assigned to the pointer variable next and the list ends. Otherwise, memory space is allocated to the next node using again the malloc function and the next value is placed into it. Not that the function create calls itself recursively and the process will continue until we enter the number -999.

The items stored in the linked list are printed using the function print, which accept a pointer to the current node as an argument. It is a recursive function and stops when it receives a NULL pointer. Printing algorithm is as follows;

1. Start with the first node.
2. While there are valid nodes left to print
  - (a) print the current item; and
  - (b) advance to next node.

Similarly, the function **count** counts the number of items in the list recursively and return the total number of items to the **main** function. Note that the counting does not include the item -999 (contained in the dummy node).

Program

```
#include <stdio.h>
#include <stdlib.h>
#define NULL 0
struct linked_list
{
 int number;
 struct linked_list *next;
};
typedef struct linked_list node; /*node type defined*/
main()
{
 node *head;
 void create(node *p);
 int count(node *p);
 void print(node *p);
 head = (node *)malloc(sizeof(node));
 create(head);
 printf("\n");
 printf(head);
 printf("\n");
 printf("\nNumber of items = %d \n", count(head));
}
void create(node *list)
{
 printf("Input a number\n");
 printf("(type -999 at end): ");
 scanf("%d", &list->number); /*create current node*/

 if(list->number == -999)
 {
 list->next = NULL;
 }
}
```



```

 else /*create next node*/
 {
 list->next = (node *)malloc(sizeof(node));
 create(list->next); /* Recursion occurs*/
 }
 return;
 }
 void print(node *list)
 {
 if(list->next != NULL)
 {
 printf("%d-->", list->number); /*print current item*/

 if(list->next->next == NULL)
 printf("%d", list->next->number);
 print(list->next); /*move to next item*/
 }
 return;
 }
 int count(node *list)
 {
 if(list->next == NULL)
 return (0);
 else
 return(1+ count(list->next));
 }

```

Output

```

Input a number
(type -999 to end); 60
Input a number
(type -999 to end); 20
Input a number
(type -999 to end); 10
Input a number
(type -999 to end); 40
Input a number
(type -999 to end); 30
Input a number
(type -999 to end); 50
Input a number
(type -999 to end); -999
60 -->20 -->10 -->40 -->30 -->50 --> -999
Number of items = 6

```

**Fig. 1** Creating a linear linked list

7. (a) Explain the system calls fread and fwrite with examples.  
 (b) Write a program to count no. of words and lines in a file.

[8 + 8]



**Sol. (a) Refer Section 9.4**

(b) 

```
#include<stdio.h>
#include<ctype.h>
int main()
{
 FILE *f;
 char ch;
 int line=0, word=0;
 f=fopen("file1.txt","w");
 printf("Enter text press ctrl+z to quit\n");
 do
 {ch=getchar();
 putc(ch,f);
 }
 while(ch!=EOF);
 fclose(f);
 f=fopen("file1.txt","r");while(ch=getc(f))!=EOF
 {
 if(ch=='\n')
 line++;
 if(isspace(ch)||ch=='\t'||ch=='\n')
 word++;
 putchar(ch);
 }
 fclose(f);printf("\n no of line=%d\n", line);
 printf("no of word=%d\n", word);
 return 0;
}
```