PROGRAMMING
FOR
PROBLEM SOLVING

Gujarat Technological University - 2018

About the Author

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

A prolific writer, he has authored a large number of research papers and several books.

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

PROGRAMMING FOR PROBLEM SOLVING

Gujarat Technological University - 2018

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 Naickr Industrial Estate, Alapakkam, Porur, Chennai-600 116

Programming for Problem Solving

Print Edition
ISBN (13 digit): 978-93-5316-278-8
ISBN (10 digit): 93-5316-278-5

E-book Edition
ISBN (13 digit): 978-93-5316-279-5
ISBN (10 digit): 93-5316-279-3

1 2 3 4 5 6 7 8 9 D101417 22 21 20 19 18

Printed and bound in India.
Director—Science & Engineering Portfolio: *Vibha Mahajan*

Senior Portfolio Manager—Science & Engineering: *Hemant K Jha*
Associate Portfolio Manager —Science & Engineering: *Tushar Mishra*

Production Head: *Satinder S Baveja*
Copy Editor: *Taranpreet Kaur*
Assistant Manager—Production: *Suhaib Ali*

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. Apartment, Paschim Vihar, New Delhi 110 063 and printed at

Cover Printer:

Cover Designer: APS Compugraphics

Cover Image Source: Shutterstock

Visit us at: www.mheducation.co.in

Write to us at: info.india@mheducation.com

CIN: U22200TN1970PTC111531

Toll Free Number: 1800 103 5875

Preface

Programming for Problem Solving requires a deep understanding of C. C is a powerful, flexible, portable and elegantly structured programming language. Since C combines the features of high-level language with the elements of the assembler, it is suitable for both systems and applications programming. It is undoubtedly the most widely used general-purpose language today in operating systems, and embedded system development. Its influence is evident in almost all modern programming languages. Since its standardization in 1989, C has undergone a series of changes and improvements in order to enhance the usefulness of the language.

Organization of the Book

Programming for Problem Solving starts with an Introduction to Computer Programming, **Chapter 2** discusses fundamentals of C. Control Structures in C is presented in **Chapter 3**. **Chapter 4** deals with Arrays and Strings. **Chapter 5** discusses Functions. In **Chapter 6** students can study Pointers. While **Chapter 7** details Structure. Dynamic Memory Allocation is discussed in **Chapter 8**. **Chapter 9** details on File Management.

Salient Features of the Book

- Learning Objectives
- Key Concepts
- Content Tagged with LO
- Worked Out Problems
- Tips
- Closing Vignette
- Review Exercises – True False, Fill in the blanks, Questions, Programming Exercises – categorized into LO and Difficulty level (E for Easy, M for Medium and H for High)

Acknowledgements

I owe special thanks to the entire team of McGraw Hill Education India.

A note of acknowledgement is due to the following reviewers for their valuable feedback.

Ms. Kalpana Mudaliar	Gandhinagar Institute of Technology, Gandhinagar, Gujarat
H K Patnaik	*Kalinga Institute of Industrial Technology, KIIT University, Bhubaneswar*
T V Gopal	*College of Engineering, Anna University, Chennai, Tamil Nadu*
Unnati Nitin Chaudhari	*Tolani Maritime Institute, Pune, Maharashtra*
Tanveer Ahmed	*Jamia Millia Islamia, Delhi*
Ravindra Divekar	*KJ Somaiya College of Engineering, Mumbai, Maharashtra*
SC Dutta	*Birsa Institute of Technology, Sindri, Dhanbad, Jharkhand*
D Lakshmi	*Adithya Institute of Technology, Coimbatore, Tamil Nadu*

This book is my sincere attempt to make a footprint on the immensely vast and infinite sands of knowledge. I would request the readers to utilize this book to the maximum extent.

E Balagurusamy

Publisher's Note

McGraw Hill Education (India) invites suggestions and comments from you, 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.

Contents

Roadmap to the Syllabus

Programming For Problem Solving

Code: 3110003

Introduction to computer and programming: Introduction, Basic block diagram and functions of various components of computer, Concepts of Hardware and software, Types of software, Compiler and interpreter, Concepts of Machine level, Assembly level and high level programming, Flowcharts and Algorithms

GO TO **Chapter 1** Introduction to Computer and Programming

Fundamentals of C: Features of C language, structure of C Program, comments, header files, data types, constants and variables, operators, expressions, evaluation of expressions, type conversion, precedence and associativity, I/O functions

GO TO **Chapter 2** Fundamentals of C

Control structure in C: Simple statements, Decision making statements, Looping statements, Nesting of control structures, break and continue, goto statement

GO TO **Chapter 3** Control Structure in C

Array & String: Concepts of array, one and two dimensional arrays, declaration and initialization of arrays, string, string storage, Built-in-string functions

Recursion: Recursion, as a different way of solving problems. Example programs, such as Finding Factorial, Fibonacci series, Ackerman function etc. Quick sort or Merge sort.

GO TO **Chapter 4** Array & String

Functions: Concepts of user defined functions, prototypes, definition of function, parameters, parameter passing, calling a function, recursive function, Macros, Pre-processing

GO TO **Chapter 5** Functions

Pointers: Basics of pointers, pointer to pointer, pointer and array, pointer to array, array to pointer, function returning pointer

GO TO **Chapter 6** Pointers

Structure: Basics of structure, structure members, accessing structure members, nested structures, array of structures, structure and functions, structures and pointers

GO TO **Chapter 7** Structure

Dynamic memory allocation: Introduction to Dynamic memory allocation, malloc, calloc

GO TO **Chapter 8** Dynamic Memory Allocation

File management: Introduction to file management and its functions

GO TO **Chapter 9** File Management

Introduction to Computer and Programming

Chapter
1

LEARNING OBJECTIVES

LO 1.1	Identify the various generations of computers
LO 1.2	Classify computers on the basis of different criteria
LO 1.3	Describe the computer system
LO 1.4	Classify various computer software
LO 1.5	Discuss various operating systems
LO 1.6	Discuss Microsoft software
LO 1.7	Know various networking concepts and protocols
LO 1.8	Identify the various positional number systems
LO 1.9	Carry out number conversions from one number system to another
LO 1.10	Explain how binary arithmetic operations are performed
LO 1.11	Describe primary logic gates
LO 1.12	Discuss various levels of programming languages
LO 1.13	Know various problem solving techniques and computer applications

INTRODUCTION

A computer is an *electronic machine* that takes input from the user, processes the given input and generates output in the form of useful information. A computer accepts input in different forms such as data, programs and user reply. *Data* refer to the raw details that need to be processed to generate some useful *information*. Programs refer to the set of instructions that can be executed by the computer in sequential or non-sequential manner. User reply is the input provided by the user in response to a question asked by the computer.

A computer includes various devices that function as an integrated system to perform several tasks described above (Fig. 1.1). These devices are:

Central Processing Unit (CPU)

It is the processor of the computer that is responsible for controlling and executing instructions in the computer. It is considered as the most significant component of the computer.

Monitor

It is a screen, which displays information in visual form, after receiving the video signals from the computer.

Keyboard and Mouse

These are the devices, which are used by the computer, for receiving input from the user.

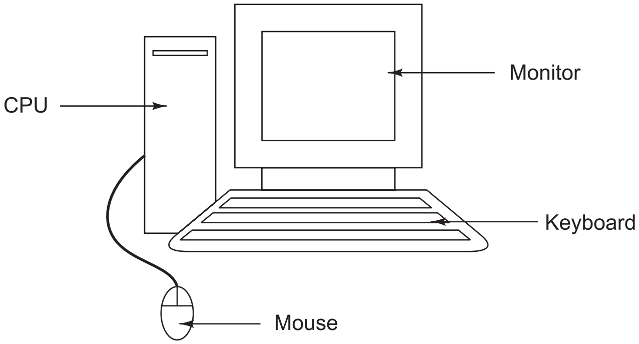


Fig. 1.1 *The components of computer*

Computers store and process numbers, letters and words that are often referred to as *data*.

- How do we communicate data to computers?
- How do the computers store and process data?

Since the computers cannot understand the Arabic numerals or the English alphabets, we should use some 'codes' that can be easily understood by them.

In all modern computers, storage and processing units are made of a set of silicon chips, each containing a large number of transistors. A transistor is a two-state device that can be put 'off' and 'on' by passing an electric current through it. Since the transistors are sensitive to currents and act like switches, we can communicate with the computers using electric signals, which are represented as a series of 'pulse' and 'no-pulse' conditions. For the sake of convenience and ease of use, a pulse is represented by the code '1' and a no-pulse by the code '0'. They are called *bits*, an abbreviation of 'binary digits'. A series of 1s and 0s are used to represent a number or a character and thus they provide a way for humans and computers to communicate with one another. This idea was suggested by John Von Neumann in 1946. The numbers represented by binary digits are known as *binary numbers*. Computers not only store numbers but also perform operations on them in binary form.

In this chapter, we discuss how the numbers are represented using what are known as *binary codes*, how computers perform arithmetic operations using the binary representation, how digital circuits known as *logic gates* are used to manipulate data, how instructions are designed using what are known as *programming languages* and how *algorithms* and *flow charts* might help us in developing programs.

GENERATIONS OF COMPUTERS

LO 1.1
Identify the various generations of computers

The history of computer development is often discussed in terms of different generation of computers, as listed below.

- First generation computers
- Second generation computers
- Third generation computers
- Fourth generation computers
- Fifth generation computers

First Generation Computers

These computers used the *vacuum tubes* technology (Fig. 1.2) for calculation as well as for storage and control purposes. Therefore, these computers were also known as vacuum tubes or thermionic valves based machines. Some examples of first generation computers are ENIAC, EDVAC, EDSAC and UNIVAC.

Advantages

- Fastest computing devices of their time.
- Able to execute complex mathematical problems in an efficient manner.

Disadvantages

- These computers were not very easy to program being machine dependent.
- They were not very flexible in running different types of applications as designed for special purposes.
- The use of vacuum tube technology made these computers very large and bulky and also required to be placed in cool places.
- They could execute only one program at a time and hence, were not very productive.
- They generated huge amount of heat and hence were prone to hardware faults.

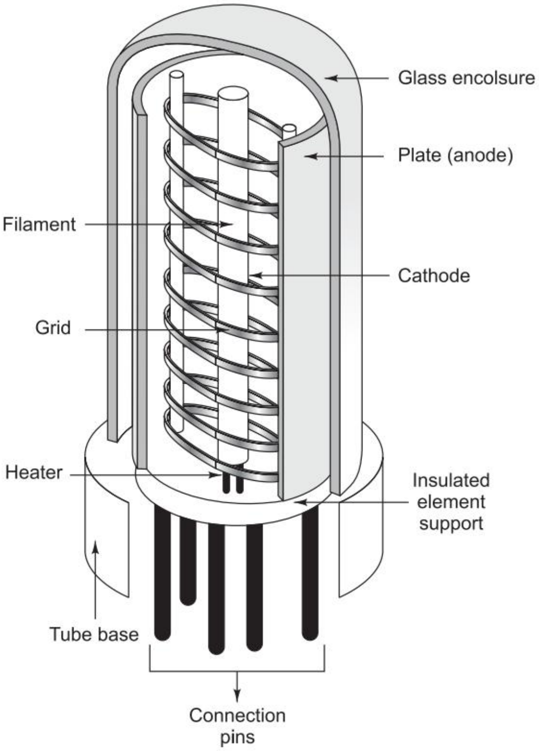


Fig. 1.2 *A vacuum tube*

Second Generation Computers

These computers use *transistors* in place of vacuum tubes in building the basic logic circuits. A transistor is a semiconductor device that is used to increase the power of the incoming signals by preserving the shape of the original signal (Fig. 1.3).

Some examples of second generation computers are PDP-8, IBM 1401 and IBM 7090.

Advantages

- Fastest computing devices of their time.
- Easy to program because of the use of assembly language.
- Small and light weight computing devices.
- Required very less power in carrying out operations.

Disadvantages

- Input and output media for these computers were not improved to a considerable extent.
- Required to be placed in air-conditioned places.

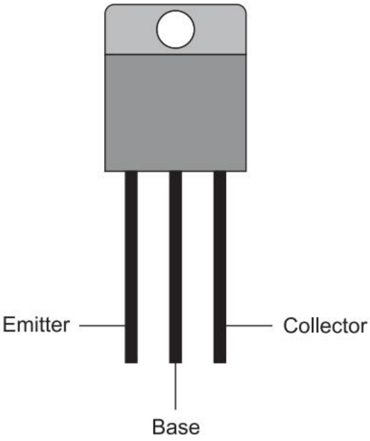


Fig. 1.3 *A transistor*

- Very expensive and beyond the reach of home users.
- Being special-purpose computers they could execute only specific applications.

Third Generation Computers

The major characteristic feature of third generation computer systems was the use of *Integrated Circuits* (ICs). ICs are the circuits that combine various electronic components, such as transistors, resistors, capacitors, etc. onto a single small silicon chip.

Some examples of third generation computers are NCR 395, B6500, IBM 370, PDP 11 and CDC 7600.

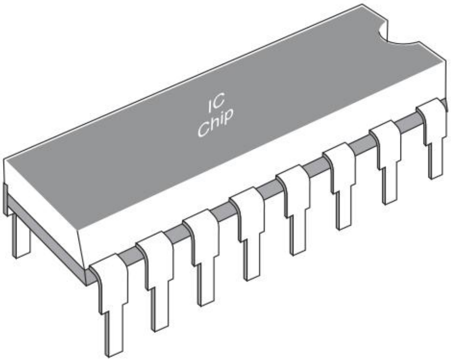


Fig. 1.4 *An integrated circuit*

Advantages

- Computational time for these computers was usually in nanoseconds hence were the fastest computing devices
- Easily transportable because of their small size.
- They used high-level languages which is machine independent hence very easy to use.
- Easily installed and required less space.
- Being able to execute any type of application (business and scientific) these were considered as general-purpose computers.

Disadvantages

- Very less storage capacity.
- Degraded performance while executing complex computations because of the small storage capacity.
- Very expensive.

Fourth Generation Computers

The progress in LSI and VLSI technologies led to the development of *microprocessor*, which became the major characteristic feature of the fourth generation computers. The LSI and VLSI technology allowed thousands of transistors to be fitted onto one small silicon chip.

A microprocessor incorporates various components of a computer—such as CPU, memory and Input/Output (I/O) controls—onto a single chip. Some popular later microprocessors include Intel 386, Intel 486 and Pentium.

Some of the examples of fourth generation computers are IBM PC, IBM PC/AT, Apple and CRAY-1.

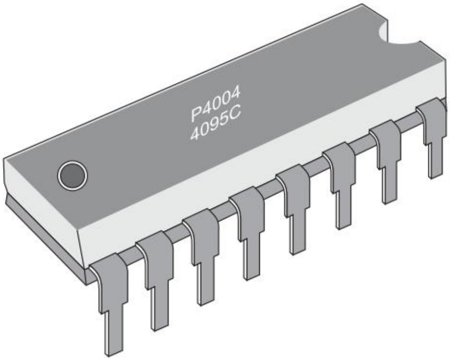


Fig. 1.5 *The Intel P4004 microprocessor chip*

Advantages

- LSI and VLSI technologies made them small, cheap, compact and powerful.
- high storage capacity
- highly reliable and required very less maintenance.

- provided a user-friendly environment with the development of GUIs and interactive I/O devices.
- programs written on these computers were highly portable because of the use of high-level languages.
- very versatile and suitable for every type of applications.
- required very less power to operate.

Disadvantages

- the soldering of LSI and VLSI chips on the wiring board was complicated
- still dependent on the instructions given by the programmer.

Fifth Generation Computers

Fifth generation computers are based on the Ultra Large Scale Integration (ULSI) technology that allows almost ten million electronic components to be fabricated on one small chip.

Advantages

- faster, cheaper and most efficient computers till date.
- They are able to execute a large number of applications at the same time and that too at a very high speed.
- The use of ULSI technology helps in decreasing the size of these computers to a large extent.
- very comfortable to use because of the several additional multimedia features.
- versatile for communications and resource sharing.

Disadvantage

They are not provided with an intelligent program that could guide them in performing different operations.

Figure 1.6 shows a tree of computer family that illustrates the area-wise developments during the last four decades and their contributions to the various generations of computers.

CLASSIFICATION OF COMPUTERS

LO 1.2
Classify computers on the basis of different criteria

Computers can be classified into several categories depending on their computing ability and processing speed. These include

- Microcomputer
- Minicomputer
- Mainframe computers
- Supercomputers

Microcomputers

A microcomputer is defined as a computer that has a microprocessor as its CPU and can perform the following basic operations:

- **Inputting** — entering data and instructions into the microcomputer system.
- **Storing** — saving data and instructions in the memory of the microcomputer system, so that they can be use whenever required.
- **Processing** — performing arithmetic or logical operations on data, where data, such as addition, subtraction, multiplication and division.
- **Outputting** — It provides the results to the user, which could be in the form of visual display and/or printed reports.
- **Controlling** — It helps in directing the sequence and manner in which all the above operations are performed.

Minicomputers

A minicomputer is a medium-sized computer that is more powerful than a microcomputer. It is usually designed to serve multiple users simultaneously, hence called a multiterminal, time-sharing system.

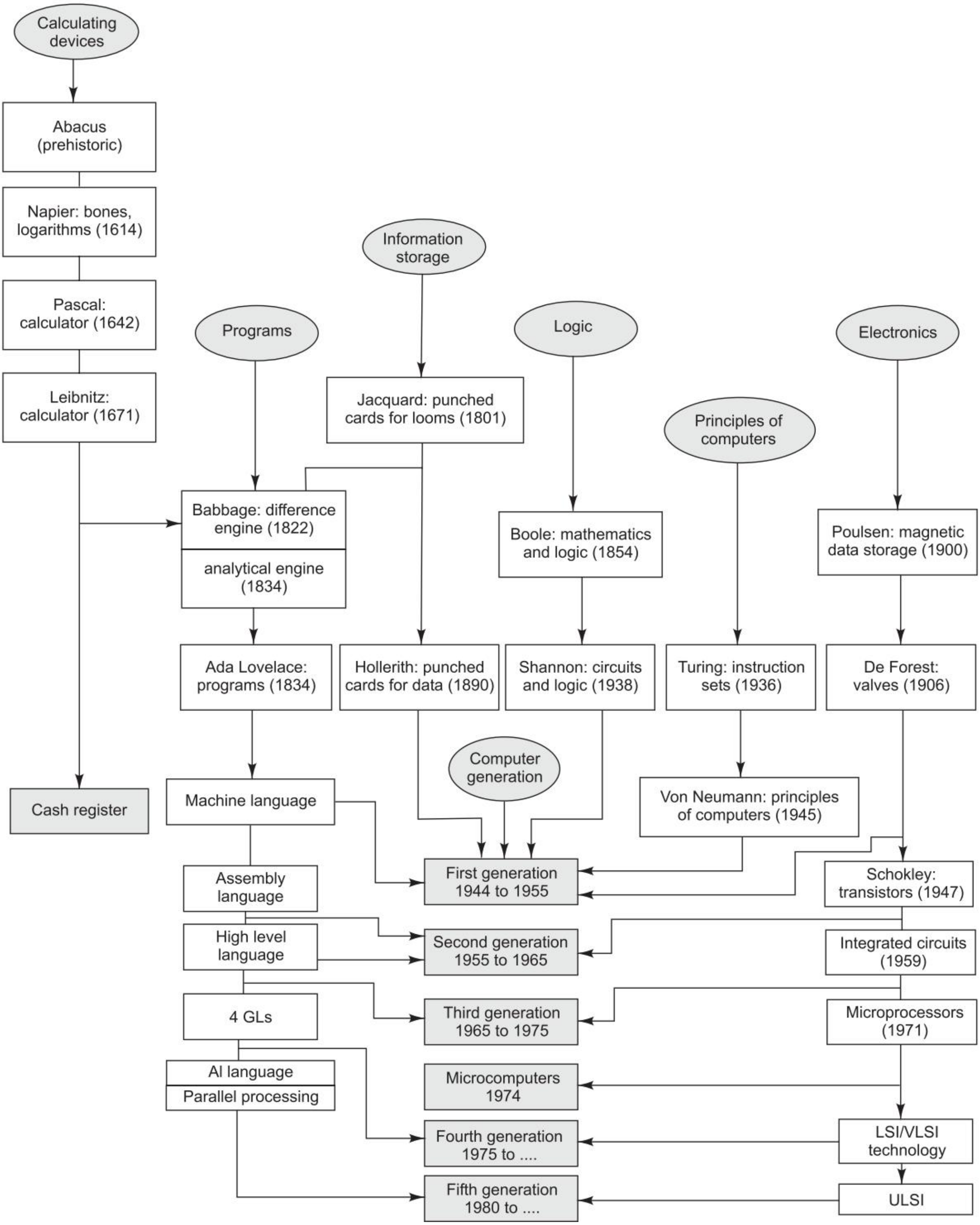

Calculating devices
Abacus (prehistoric)
Napier: bones, logarithms (1614)
Pascal: calculator (1642)
Leibnitz: calculator (1671)
Cash register
Programs
Information storage
Jacquard: punched cards for looms (1801)
Babbage: difference engine (1822)
analytical engine (1834)
Ada Lovelace: programs (1834)
Hollerith: punched cards for data (1890)
Machine language
Assembly language
High level language
4 GLs
AI language
Parallel processing
Logic
Boole: mathematics and logic (1854)
Shannon: circuits and logic (1938)
Computer generation
First generation 1944 to 1955
Second generation 1955 to 1965
Third generation 1965 to 1975
Microcomputers 1974
Fourth generation 1975 to ....
Fifth generation 1980 to ....
Principles of computers
Turing: instruction sets (1936)
Von Neumann: principles of computers (1945)
Electronics
Poulsen: magnetic data storage (1900)
De Forest: valves (1906)
Schokley: transistors (1947)
Integrated circuits (1959)
Microprocessors (1971)
LSI/VLSI technology
ULSI

Fig. 1.6 *Tree of computer family*

Minicomputers are popular among research and business organizations today. They are more expensive than microcomputers.

Mainframe Computers

Mainframe computers help in handling the information processing of various organizations like banks, insurance companies, hospitals and railways. Mainframe computers are placed on a central location and are connected to several user terminals, which can act as access stations and may be located in the same building. Mainframe computers are larger and expensive in comparison to the workstations.

Supercomputers

In supercomputers, multiprocessing and parallel processing technologies are used to promptly solve complex problems. Here, the multiprocessor can enable the user to divide a complex problem into smaller problems. A supercomputer also supports multiprogramming where multiple users can access the computer simultaneously. Presently, some of the popular manufacturers of supercomputers are IBM, Silicon Graphics, Fujitsu, and Intel.

BASIC ANATOMY OF A COMPUTER SYSTEM

A computer system comprises **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.

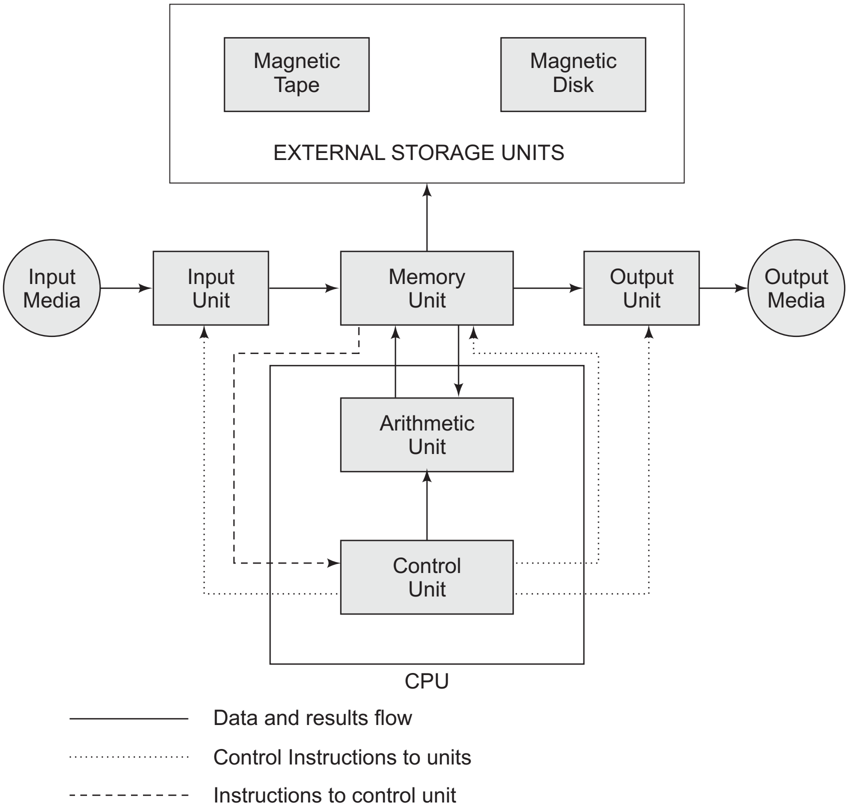


Fig. 1.7 *Interaction among hardware components*

- **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 (operating system and networking system) is mandatory for running application software.

INPUT DEVICES

Input devices are electromechanical devices that are used to provide data to a computer for storing and further processing, if necessary. Depending upon the type or method of input, the input device may belong to one of the following categories:

LO 1.3
Describe the computer system

Keyboard

Keyboard is used to type data and text and execute commands. A standard keyboard, as shown in Fig. 1.8, consists of the following groups of keys:

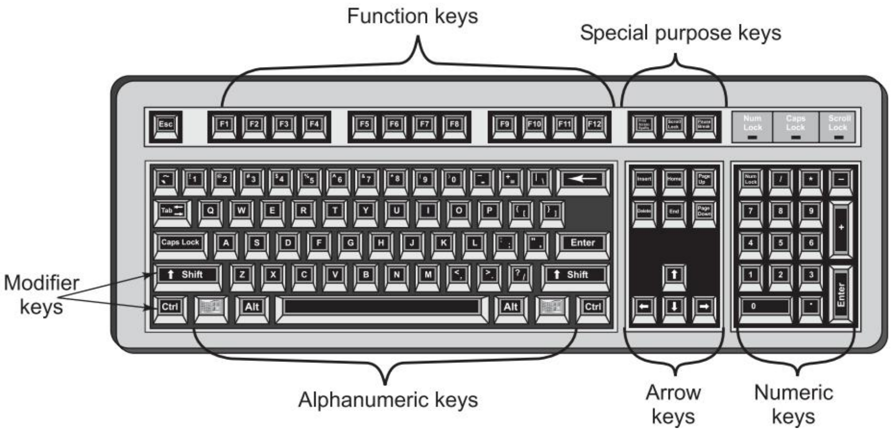


Fig. 1.8 *The presently used keyboard*

Alphanumeric Keys include the number keys and alphabet keys arranged in QWERTY layout.

Function Keys help perform specific tasks, such as searching a file or refreshing a web page.

Central Keys include arrow keys (for moving the cursor) and modifier keys such as SHIFT, ALT and CTRL (for modifying the input).

Numeric Keypad looks like a calculator's keypad with its 10 digits and mathematical operators.

Special Purpose Keys The special purpose keys help perform a certain kind of operation, like exiting a program (Escape) or deleting some characters (Delete) in a document, etc.

Mouse

Mouse is a small hand-held pointing device that basically controls the two-dimensional movement of the cursor on the displayed screen. It is an important part of the Graphical User Interface (GUI) based Operating Systems (OS) as it helps in selecting a portion of the screen and copying and pasting the text.

The mouse, on moving, also moves the pointer appearing on the display device (Fig. 1.9).

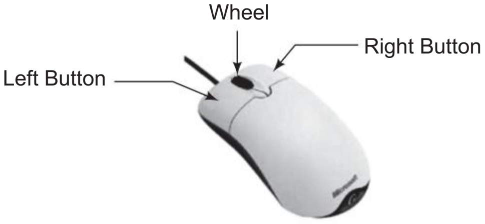


Fig. 1.9 *A mechanical mouse*

Scanning Device

Scanning devices are the input devices that can electronically capture text and images, and convert them into computer readable form (Fig. 1.10).

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

Fig. 1.10 *A Scanner*

- **Flatbed scanner** — It contains a scanner head that moves across a page from top to bottom to read the page and converts the image or text available on the page in digital form. The flatbed scanner is used to scan graphics, oversized documents, and pages from books.
- **Drum scanner** — In this type of scanner, a fixed scanner head is used and the image to be scanned is moved across the head. The drum scanners are used for scanning prepress materials.
- **Slide scanner** — It is a scanner that can scan photographic slides directly to produce files understandable by the computer.
- **Handheld scanner** — It is a scanner that is moved by the end user across the page to be scanned. This type of scanner is inexpensive and small in size.

PROCESSOR

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

Motherboard

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

- **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.
- **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.
- **Microprocessor**: CPU component that performs the processing and controls the activities performed by the different parts of the computer.
- **ROM**: Chip that contains the permanent memory of the computer that stores information, which cannot be modified by the end user.

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

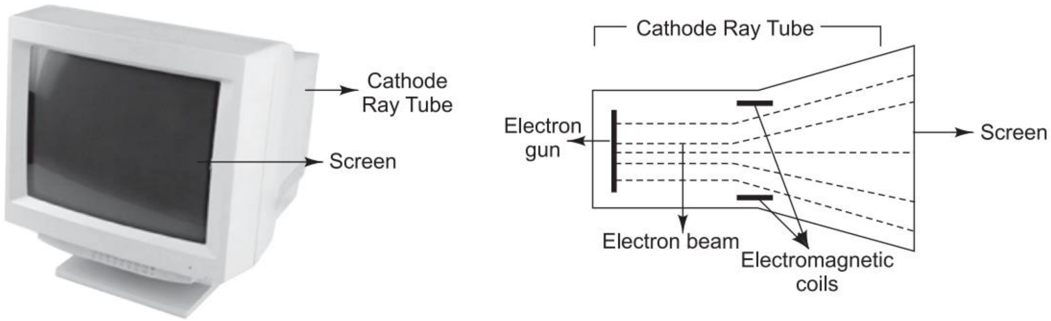
The video card is an interface between the monitor and the CPU. Video cards also include their own RAM and microprocessors that are used for speeding up the processing and display of a graphic. A sound card is a circuit board placed on the motherboard and is used to enhance the sound capabilities of a computer.

OUTPUT DEVICES

The main task of an output device is to convert the machine-readable information into human-readable form which may be in the form of text, graphics, audio or video.

Display Monitors

A monitor produces visual displays generated by the computer. The monitor is connected to the video card placed on the expansion slot of the motherboard.



(a) CRT Monitor

(b) Internal Components of CRT

Fig. 1.11 *A CRT monitor and the internal components of a CRT*

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.

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 transfers the text displayed on the screen, onto paper sheets that can be used by the end user. 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. Dot matrix printers are used to produce multiple copies of a print out.
- **Inkjet printers** — Inkjet printers are slower than dot matrix printers and are used to generate high quality photographic prints.
- **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.

Voice Output Systems

These systems record the simple messages in human speech form and then combine all these simple messages to form a single message. The voice response system is of two types—one uses a reproduction of human voice and other sounds, and the other uses speech synthesis.

The basic application of a voice output system is in Interactive Voice Response (IVR) systems, which are used by the customer care or customer support departments of an organization, such as telecommunication companies, etc.

Projectors

A projector is a device that is connected to a computer or a video device for projecting an image from the computer or video device onto the big white screen. The images projected by a projector are larger in size as compared to the original images. A projector consists of an optic system, a light source and displays, which contain the original images. Projectors were initially used for showing films but now they are used on a large scale for displaying presentations in business organizations and for viewing movies at home.

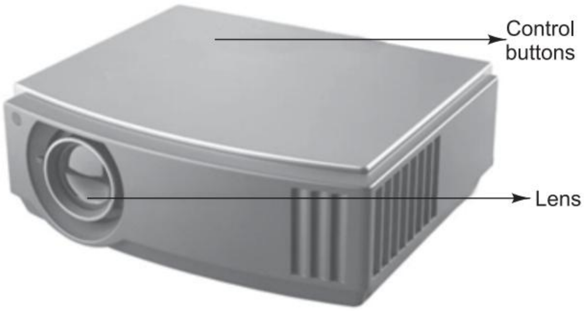


Fig. 1.12 *A portable projector*

MEMORY MANAGEMENT

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 and secondary memory. Computers also use a third type of storage location known as the *internal process memory*. This memory is placed either inside the CPU or near the CPU (connected through special fast bus).

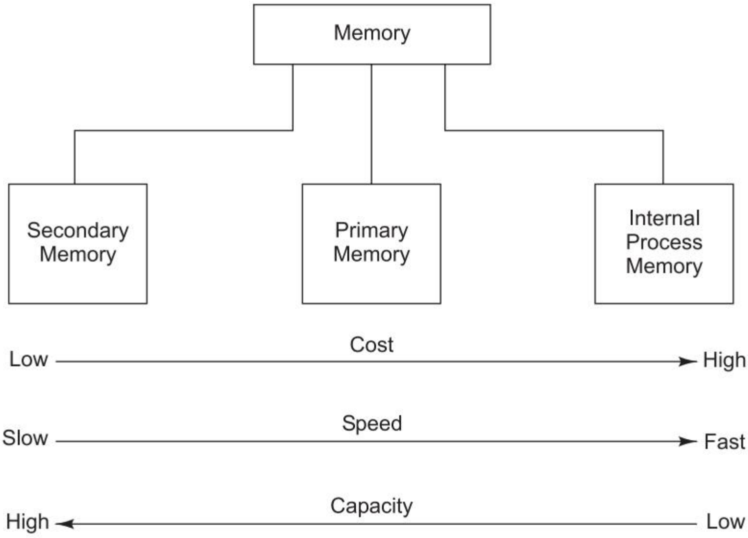


Fig. 1.13 *Memory unit categories of computer*

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 as follows:

- **ROM** — ROM represents Read Only Memory that stores data and instructions, even when the computer is turned off. It is the permanent memory of the computer where the contents cannot be modified by an end user. ROM is a chip that is inserted into the motherboard. It is generally used to store the Basic Input/Output system (BIOS), which performs the Power On Self Test (POST).
- **RAM** — RAM is the read/write memory unit in which the information is retained only as long as there is a regular power supply. When the power supply is interrupted or switched off, the information stored in the RAM is lost. RAM is 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 can be either soldered into the motherboard or is available as a part of RAM.

Secondary Memory

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

- **Magnetic storage device** — The magnetic storage devices store information that can be read, erased and rewritten a number of times. These include floppy disk, hard disk and magnetic tapes.
- **Optical storage device** — The optical storage devices are secondary storage devices that use laser beams to read the stored data. These include CD-ROM, rewritable compact disk (CD-RW), digital video disks with read only memory (DVD-ROM), etc.
- **Magneto-optical storage device** — The magneto-optical devices are generally used to store information, such as large programs, files and back-up data. The end user can modify the information stored in magneto-optical storage devices multiple times. These devices provide higher storage capacity as they use laser beams and magnets for reading and writing data to the device.

TYPES OF COMPUTER SOFTWARE

LO 1.4
Classify various computer software

A computer program is basically a set of logical instructions, written in a computer programming language that tells the computer how to accomplish a task. The software is therefore an essential interface between the hardware and the user (Fig. 1.14).

A computer software performs two distinctive tasks. The first task is to control and coordinate the hardware components and manage their performances and the second one is to enable the users to accomplish their required tasks. The software that is used to achieve the first task is known as the *system software* and the software that is used to achieve the second task is known as the *application software*.

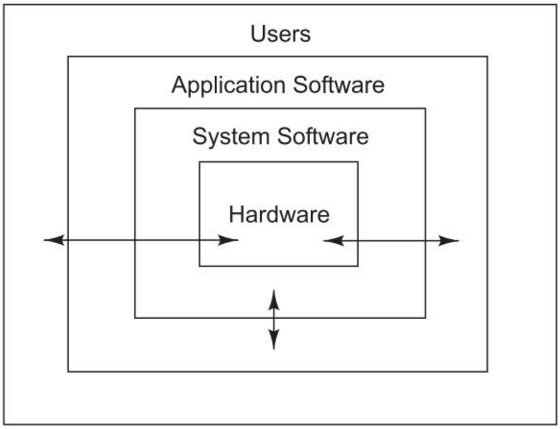


Fig. 1.14 *Layers of software and their interactions*

System Software

System software consists of many different programs that manage and support different tasks. Depending upon the task performed, the system software can be classified into two major groups (Fig. 1.15):

- *System management programs* used for managing both the hardware and software systems. They include:
  - Operating system

  - Utility programs
  - Device drivers
- *System development programs* are used for developing and executing application software. These are:
  - Language translators
  - Linkers
  - Debuggers
  - Editors

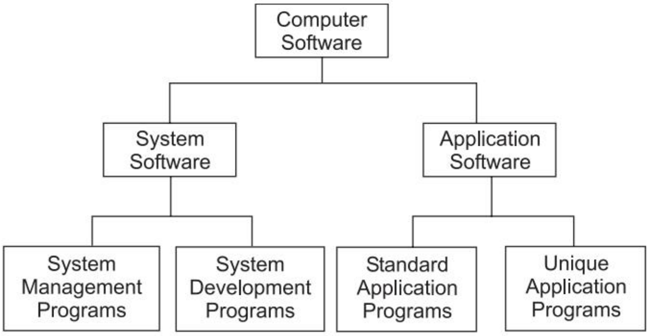

Computer Software
System Software
Application Software
System Management Programs
System Development Programs
Standard Application Programs
Unique Application Programs

Fig. 1.15 *Major categories of computer software*

Application Software

Application software includes a variety of programs that are designed to meet the information processing needs of end users. They can be broadly classified into two groups:

- *Standard application programs* that are designed for performing common application jobs. Examples include:
  - Word processor
  - Spreadsheet
  - Database Manager
  - Desktop Publisher
  - Web Browser
- *Unique application programs* that are developed by the users themselves to support their specific needs. Examples include:
  - Managing the inventory of a store
  - Preparing pay-bills of employees in an organization
  - Reserving seats in trains or airlines

OVERVIEW OF OPERATING SYSTEM

An operating system (OS) is a software that makes the computer hardware to work. While the hardware provides 'raw computer power', the OS is responsible for making the computer power useful for the users. OS is the main component of system software and therefore must be loaded and activated before we can accomplish any other task. The main functions include:

LO 1.5
Discuss various operating systems

- Operates CPU of the computer.
- Controls input/output devices that provide the interface between the user and the computer.

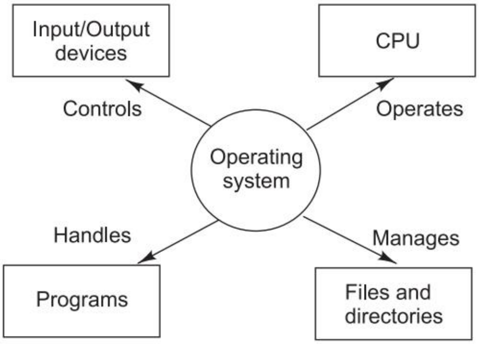


Fig. 1.16 *The roles of an operating system*

- Handles the working of application programs with the hardware and other software systems.
- Manages the storage and retrieval of information using storage devices such as disks.

Based on their capabilities and the types of applications supported, the operating systems can be divided into the following six major categories:

- **Batch operating system** — This is the earliest operating system, where only one program is allowed to run at one time. We cannot modify any data used by the program while it is being run. If an error is encountered, it means starting the program from scratch all over again. A popular batch operating system is MS DOS.
- **Interactive operating system** — This operating system comes after the batch operating system, where also only one program can run at one time. However, here, modification and entry of data are allowed while the program is running. An example of an interactive operating system is Multics (Multiplexed Information and Computing Service).
- **Multiuser operating system** — A multiuser operating system allows more than one user to use a computer system either at the same time or at different times. Examples of multiuser operating systems include Linux and Windows 2000.
- **Multi-tasking operating system** — A multi-tasking operating system allows more than one program to run at the same time. Examples of multi-tasking operating systems include Unix and Windows 2000.
- **Multithreading operating system** — A multithreading operating system allows the running of different parts of a program at the same time. Examples of multithreading operating system include UNIX and Linux.
- **Real-time operating systems** — These operating systems are specially designed and developed for handling real-time applications or embedded applications. Example include MTOS, Lynx, RTX
- **Multiprocessor operating systems** — The multiprocessor operating system allows the use of multiple CPUs in a computer system for executing multiple processes at the same time. Example include Linux, Unix, Windows 7.
- **Embedded operating systems** — The embedded operating system is installed on an embedded computer system, which is primarily used for performing computational tasks in electronic devices. Example include Palm OS, Windows CE

MS DOS Operating System

MS DOS or Microsoft Disk Operating System, which is marketed by Microsoft Corporation and is one of the most commonly used members of the DOS family of operating systems. MS DOS is a command line user interface, which was first introduced in 1981 for IBM computers. Although MS DOS, nowadays, is not used as a stand-alone product, but it comes as an integrated product with the various versions of Windows.

In MS DOS, unlike Graphical User Interface (GUI)-based operating systems, there is a command line interface, which is known as MS DOS prompt. Here, we need to type the various commands to perform the operations in MS DOS operating system. The MS DOS commands can be broadly categorized into the following three classes:

- **Environment command** — These commands usually provide information on or affects operating system environment. Some of these commands are:
  - **CLS**: It allows the user to clear the complete content of the screen leaving only the MS-DOS prompt.
  - **TIME**: It allows the user to view and edit the time of the computer.
  - **DATE**: It allows the user to view the current date as well as change the date to an alternate date.
  - **VER**: It allows us to view the version of the MS-DOS operating system.
- **File manipulation command** — These commands help in manipulating files, such as copying a file or deleting a file. Some of these commands include:
  - **COPY**: It allows the user to copy one or more files from one specified location to an alternate location.
  - **DEL**: It helps in deleting a file from the computer.
  - **TYPE**: It allows the user to view the contents of a file in the command prompt.
  - **DIR**: It allows the user to view the files available in the current and/or parent directories.
- **Utilities** — These are special commands that perform various useful functions, such as formatting a diskette or invoking the text editor in the command prompt. Some of these commands include:
  - **FORMAT**: It allows the user to erase all the content from a computer diskette or a fixed drive.
  - **EDIT**: It allows the user to view a computer file in the command prompt, create and modify the computer files.

MS Windows Operating System

Windows Architecture

The architecture of Windows operating system comprises a modular structure that is compatible with a variety of hardware platforms. Figure 1.17 shows the architecture of Windows 2000; the later releases of Windows operating systems are based on similar architecture.

At a high level, the architecture is divided into three layers, viz.

- **User mode**: Comprises application and I/O specific software components
- **Kernel mode**: Has complete access to system resources and hardware
- **Hardware**: Comprises underlying hardware platform

User Mode

The various subsystems in the user mode are divided into the following two categories:

- **Environment subsystems**: Comprise subsystems that run applications written for other operating systems. These subsystems cannot directly request hardware access; instead such requests are processed by virtual memory manager present in the kernel mode. The three main environment subsystems include Win32, OS/2 and POSIX. Each of these subsystems possess dynamic link libraries for converting user application calls to Windows calls.
- **Integral subsystems**: Takes care of the operating system specific functions on behalf of the environment subsystems. The various integral subsystems include workstation service, server service and security.

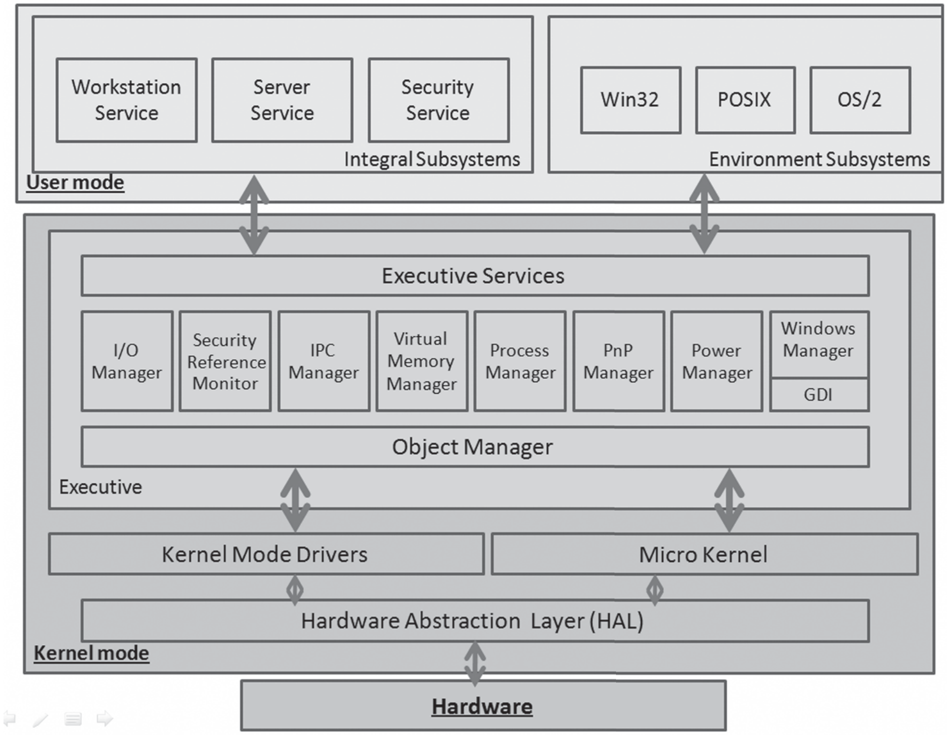


Fig. 1.17 *The architecture of Windows 2000*

Kernel Mode

The kernel mode comprises various components with each component managing specific system function. Each of the components is independent and can be removed, upgraded or replaced without rewriting the entire system. The various kernel-mode components include:

- **Executive**: Comprises the core operating system services including memory management, process management, security, I/O, inter process communication etc.
- **Kernel**: Comprises the core components that help in performing fundamental operating system operations including thread scheduling, exception handling, interrupt handling, multiprocessor synchronization, etc.
- **HAL**: Acts as a bridge between generic hardware communications and those specific to the underlying hardware platform. It helps in presenting a consistent view of system bus, DMA, interrupt controllers, timers, etc. to the kernel.
- **I/O manager**: Handles requests for accessing I/O devices by interacting with the relevant device drivers.
- **Security reference monitor**: Performs access validation and audit checks for Windows objects including files, processes, I/O devices, etc.
- **Virtual Memory Manager**: Performs virtual memory management by mapping virtual addresses to actual physical pages in computer's memory.
- **Process Manager**: Creates and deletes objects and threads throughout the life cycle of a process.
- **PnP manager**: Supports plug-and-play devices by determining the correct driver for a device and further loading the driver.

- **Power manager**: Performs power management for the various devices. It also optimizes power utilization by putting the devices to sleep that are not in use.
- **GDI**: Stands for Graphics Device Interface and is responsible for representing graphical objects in Windows environment. It also transfers the graphical objects to the output devices such as printer and monitor.
- **Object manager**: Manages Windows Executive objects and abstract data types that represent the various resources such as processes, threads, etc.

Unix Operating System

UNIX operating system was developed by a group of AT&T employees at Bell Labs in the year 1969. UNIX is primarily designed to allow multiple users access the computer at the same time and share resources. The UNIX operating system is written in C language. The significant properties of UNIX include:

- Multi-user capability
- Multi-tasking capability
- Portability
- Flexibility
- Security

Architecture of UNIX

UNIX has a hierarchical architecture consisting of several layers, where each layer provides a unique function as well as maintains interaction with its lower layers. The layers of the UNIX operating system are:

- Kernel
- Service
- Shell
- User applications

Figure 1.18 shows the various layers of the UNIX operating system.

- **Kernel** Kernel is the core of the UNIX operating system and it gets loaded into memory whenever we switch on the computer. Three components of kernel are:
  - **Scheduler** — It allows scheduling the processing of various jobs.
  - **Device driver** — It helps in controlling the Input/Output devices attached to the computer.
  - **I/O buffer** — It controls the I/O operations in the computer.

  Various functions performed by the kernel are:
  - Initiating and executing different programs at the same time
  - Allocating memory to various user and system processes
  - Monitoring the files that reside on the disk
  - Sending and receiving information to and from the network
- **Service** In the service layer, requests are received from the shell and they are then transformed into commands to the kernel. The service layer, which is also known as the *resident module layer*, is indistinguishable from the kernel and consists of a collection of programs providing various services, which include:
  - Providing access to various I/O devices, such as keyboard and monitor
  - Providing access to storage devices, such as disk drives
  - Controlling different file manipulation activities, such as reading from a file and writing to a file

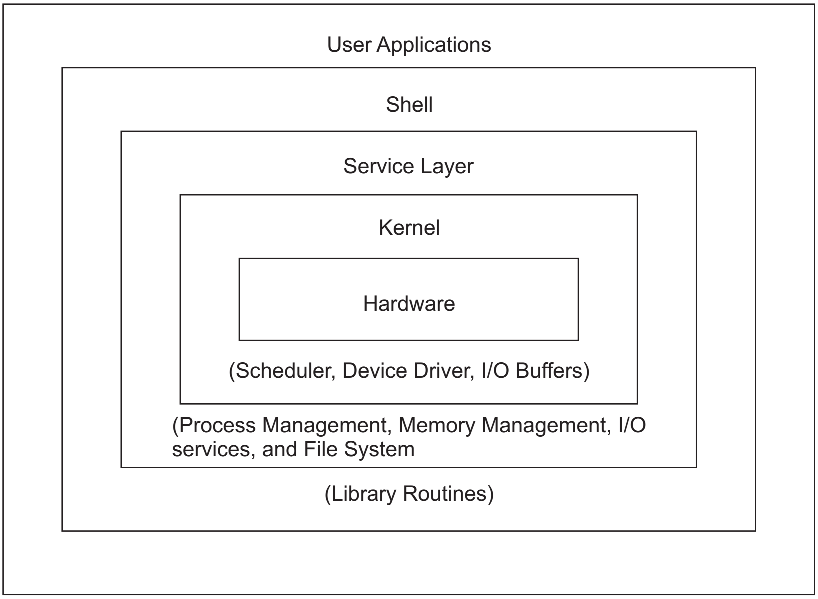

User Applications
Shell
Service Layer
Kernel
Hardware
(Scheduler, Device Driver, I/O Buffers)
(Process Management, Memory Management, I/O services, and File System
(Library Routines)

Fig. 1.18 *The layers of UNIX operating system*

- **Shell** The third layer in the UNIX architecture is the shell, which acts as an interface between a user and the computer for accepting the requests and executing programs. The shell is also known as the command interpreter that helps in controlling the interaction with the UNIX operating system. The primary function of the shell is to read the data and instructions from the terminal, and then execute commands and finally display the output on the monitor. The shell is also termed as the utility layer as it contains various library routines for executing routine tasks. The various shells that are found in the UNIX operating system are:
    - Bourne shell
    - C shell
    - Korn shell
    - Restricted shell
- **User applications** The last layer in the UNIX architecture is the user applications, which are used to perform several tasks and communicating with other users of UNIX. Some of the important examples of user applications include text processing, software development, database management and electronic communication.

MS WORD

LO 1.6
Discuss Microsoft software

MS Word is application software that can be used to create, edit, save and print personal as well as professional documents in a very simple and efficient manner. MS Word is an important tool of the MS office suite that is mainly designed for word processing. Other word processing applications available are, Open Office Writer and Google Docs.

Accessing MS Word

For working in MS Word, we need to install MS Office in a computer system. After installing MS Office, we can start MS Word by using any of the following two ways:

- Start menu
- Run command

We can start MS Word by performing the following steps using the Start menu:

1. Select Start → All Programs → Microsoft Office,
2. Select the Microsoft Office Word 2007 option to display the Graphical User Interface (GUI) of MS Word, as shown in Fig. 1.19.

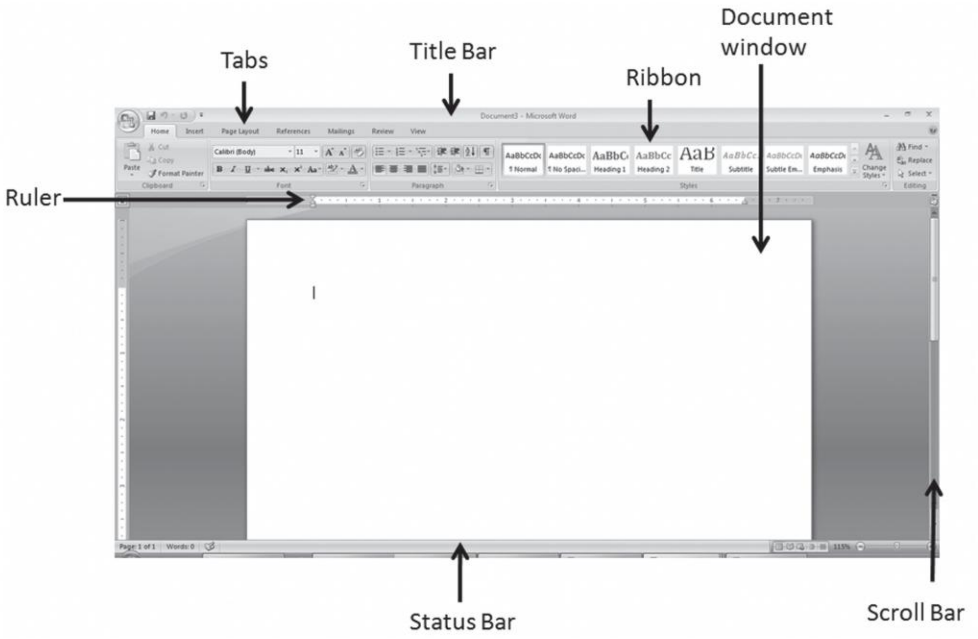


Fig. 1.19 *The Document1 – Microsoft Word window*

Using Run command We can also start MS Word by performing the following steps using the Run command:

1. Select Start → All Programs → Accessories → Run to display the Run dialog box.
2. Type winword in the Open text box and click OK to display the Document1 – Microsoft Word window.

Basic Operations Performed in MS Word

The following are the key operations that we can perform in MS Word:

- Creating a document
- Saving a document
- Editing a document
- Formatting a document
- Printing a document

MS EXCEL SYSTEM

MS Excel is an application program that allows us to create spreadsheets, which are represented in the form of a table containing rows and columns. The horizontal sequence in which the data is stored is referred to as a row. The vertical sequence in which the data is stored is referred to as a column. In a spreadsheet, a row is identified by a row header and a column is identified by a column header. Each value in a spreadsheet is stored in a cell, which is the intersection of rows and columns. A cell can contain either numeric value or a character string. We can also specify the contents of a cell using formulas. In a spreadsheet, we can perform various mathematical operations using formulas, such as addition, subtraction, multiplication, division, average, percentage, etc.

MS Excel also allows us to represent the complex data pictorially in the form of graphs. These are generally used to represent the information with the help of images, colours, etc., so that their presentation is simple and more meaningful. Some of the graphs available in spreadsheet are bar graphs, line graphs, 3-D graphs, area graphs, etc.

Accessing MS Excel

For working with MS Excel, we first need to install MS Office in our computer system. After installing MS Office, we can start MS Excel using any of the following two ways:

- Start menu
- Run command

Using Start menu We can start MS Excel by performing the following steps using the Start menu:

1. Select Start → All Programs → Microsoft Office, as shown in Fig. 1.20.
2. Select the Microsoft Office Excel 2007 option to display the GUI of MS Excel,

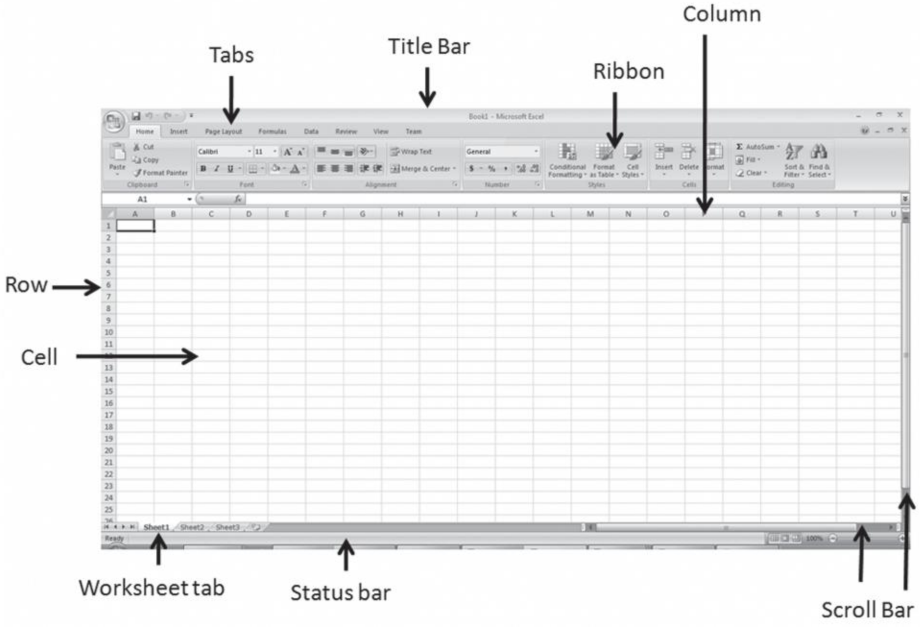


Fig. 1.20 *The Microsoft Excel—Book1 window*

Figure 1.20 shows the initial workbook of MS Excel, which in turn contains worksheets. Each worksheet contains rows and columns where we can enter data.

Using Run command We can also start MS Excel by performing the following steps using the Run command:

1. Select Start → All Programs → Accessories → Run to display the Run dialog box.
2. Type excel in the Open text box and click OK to display the Microsoft Excel – Book1 window.

Basic Operations Performed in MS Excel

Worksheet is the actual working area consisting of rows and columns. The worksheets are also known as the spreadsheets. A workbook in MS Excel is a combination of several worksheets. Each workbook of MS Excel contains three worksheets by default. The key operations that are performed in MS Excel include:

- Creating a worksheet
- Saving a worksheet
- Modifying a worksheet
- Renaming a worksheet
- Deleting a worksheet
- Moving a worksheet
- Editing a worksheet

MS POWERPOINT SYSTEM

MS PowerPoint is a software application included in the MS Office package that allows us to create presentations. PowerPoint provides a GUI with the help of which we can create attractive presentations quickly and easily. The presentation may include slides, handouts, notes, outlines, graphics and animations. A slide in PowerPoint is a combination of images, text, graphics, charts, etc., that is used to convey some meaning information. The presentations in MS PowerPoint are usually saved with the extension .ppt. The interface of MS PowerPoint is similar to the other interfaces of MS Office applications. PowerPoint presentations are commonly used in business, schools, colleges, training programmes, etc.

Accessing MS PowerPoint

For working in MS PowerPoint, we need to first install the MS Office package in our computer system. After installing MS Office, we can start MS PowerPoint using any of the following two ways:

- Start menu
- Run command

Using Start menu We can start MS PowerPoint by performing the following steps using the Start menu:

1. Select Start → All Programs → Microsoft Office,
2. Select the Microsoft Office PowerPoint 2007 option to display the GUI of MS PowerPoint, as shown in Fig. 1.21.

Using Run command We can also start MS PowerPoint by performing the following steps using the Run command:

1. Select Start → All Programs → Accessories → Run to display the Run dialog box.
2. Type powerpnt in the Open text box and click OK to display the Microsoft PowerPoint – [Presentation1] window.

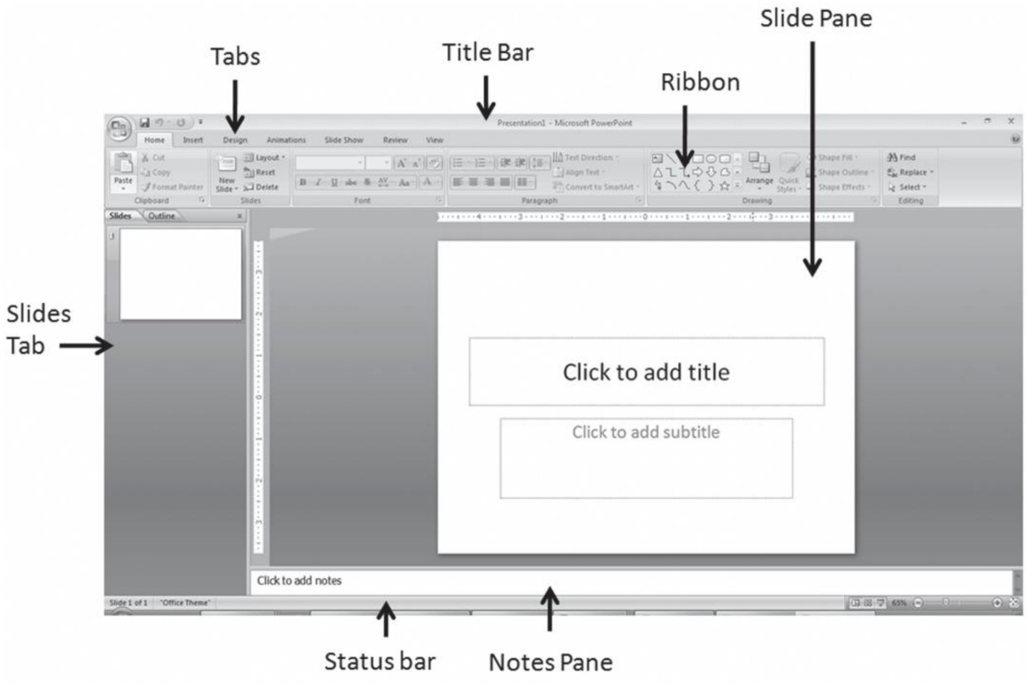


Fig. 1.21 *The Microsoft PowerPoint—[Presentation1] Window*

Basic Operations Performed on a Presentation

The following are the key operations that can be performed in MS PowerPoint:

- Creating a new presentation
- Designing the presentation
- Saving a new presentation
- Adding slides to the presentation
- Printing the presentation

NETWORKING CONCEPTS

LO 1.7
Know various networking concepts and protocols

Computer network is a system of interconnected computers that enable the computers to communicate with each other and share their resources, data and applications. The physical location of each computer is tailored to personal and organisational needs. A network may include only personal computers or a mix of PCs, minis and mainframes spanning a particular geographical area. Computer networks that are commonly used today may be classified as follows:

- Based on geographical area:
  - Local Area Networks (LANs)
  - Wide Area Networks (WANs)
  - Metropolitan Area Networks (MANs)
  - International Network (Internet)
  - Intranet

- Based on how computer nodes are used:
  - Client Server Networks (CSNs)
  - Peer-to-peer Networks (PPNs)
  - Value-added Networks (VANs)

Local Area Network (LAN)

LAN is a group of computers, as shown in Fig. 1.22, that are connected in a small area such as building, home, etc. Through this type of network, users can easily communicate with each other by sending and receiving messages. LAN is generally used for connecting two or more personal computers through some medium such as twisted pair, coaxial cable, etc. Though the number of computers connected in a LAN is limited, the data is transferred at an extremely faster rate.

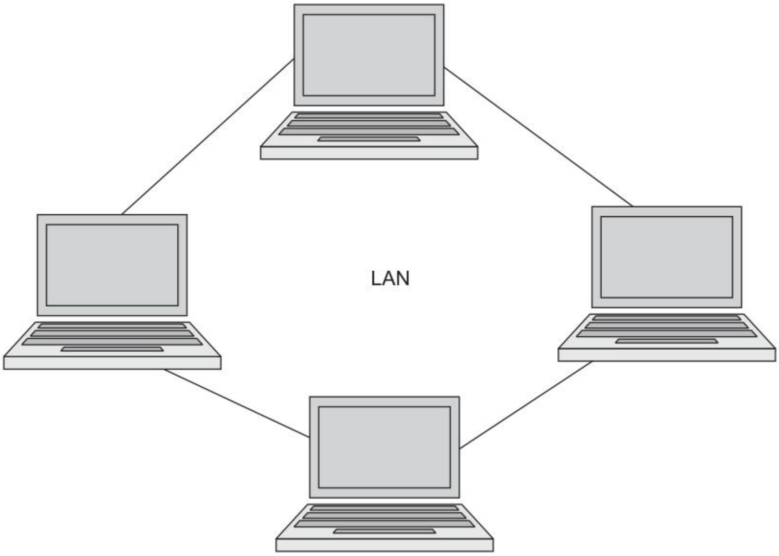


Fig. 1.22 *A LAN*

Wide Area Network (WAN)

WAN is a group of computers that are connected in a large area such as continent, country, etc. WAN is generally used for connecting two or more LANs through some medium such as leased telephone lines, microwaves, etc. In WAN, data is transferred at slow rate. A typical WAN network is shown in Fig. 1.23.

Metropolitan Area Network (MAN)

MAN is a network of computers that covers a large area like a city. The size of the MAN generally lies between that of LAN and WAN, typically covering a distance of 5 km to 50 km. The geographical area covered by MAN is comparatively larger than LAN but smaller than WAN. MAN is generally owned by private organisations. MAN is generally connected with the help of optical fibres, copper wires etc. One of the most common example of MAN is cable television network within a city as shown in Fig. 1.24. A network device known as *router* is used to connect the LANs together. The router directs the information packets to the desired destination.

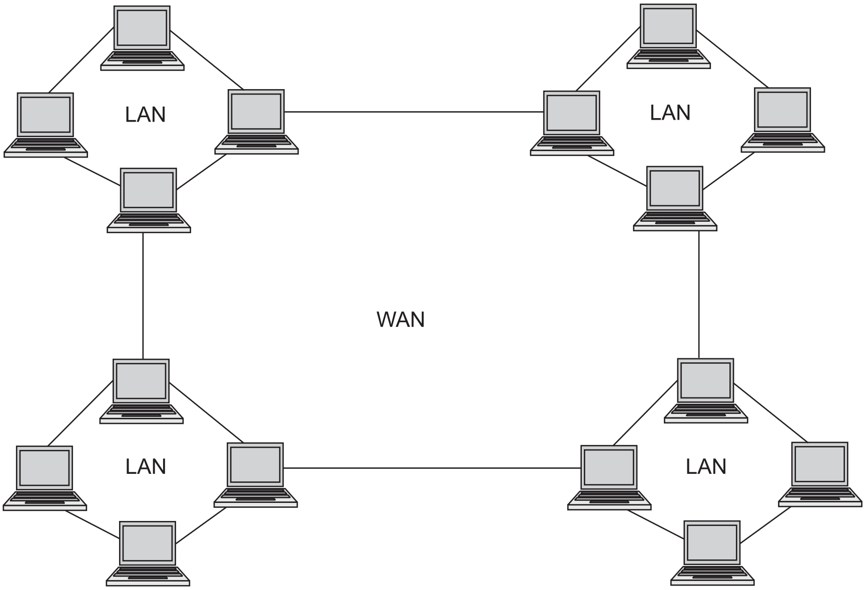


Fig. 1.23 *A WAN system*

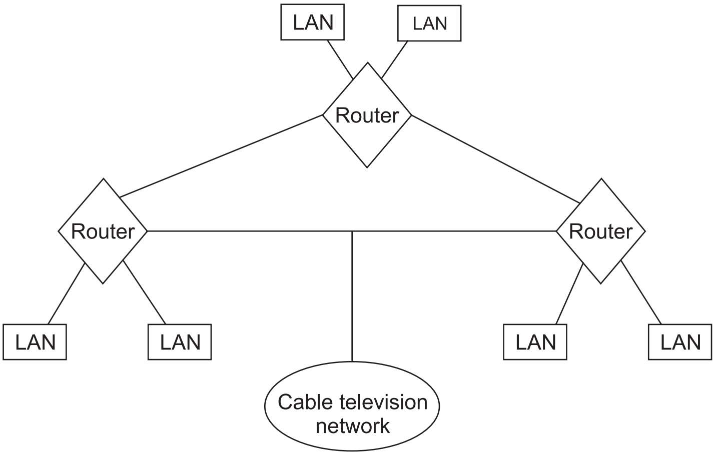


Fig. 1.24 *A typical MAN system*

NETWORK TOPOLOGIES

Network topology refers to the arrangement of computers connected in a network through some physical medium such as cable, optical fibre etc. Topology generally determines the shape of the network and the communication path between the various computers (nodes) of the network. The various types of network topologies are as follows:

- Hierarchical topology
- Bus topology

- Star topology
- Ring topology
- Mesh topology
- Hybrid topology

Hierarchical Topology

The hierarchical topology is also known as tree topology, which is divided into different levels connected with the help of twisted pair, coaxial cable or fibre optics. Figure 1.25 shows the arrangement of computers in hierarchical topology.

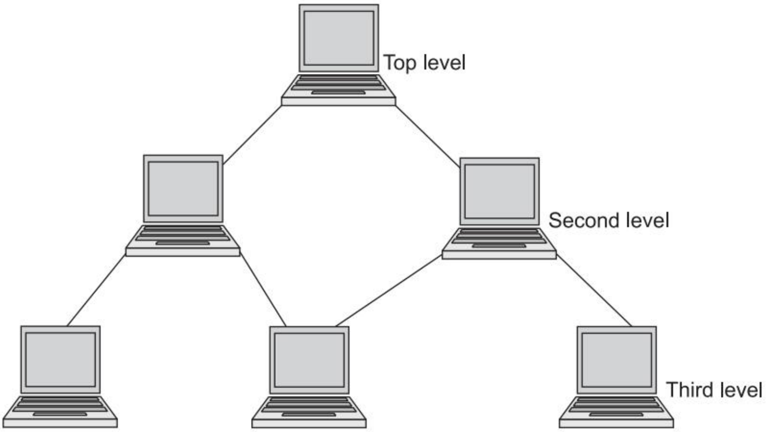


Fig. 1.25 *The hierarchical topology*

Advantages of hierarchical topology are:

- The hierarchical topology is generally supported by most hardware and software.
- In the hierarchical topology, data is received by all the nodes efficiently because of point-to-point link.

The following are the disadvantages of hierarchical topology:

- In the hierarchical topology, when the root node fails, the whole network crashes.
- The hierarchical topology is difficult to configure.

Linear Bus Topology

In the linear bus topology, all the nodes are connected to the single backbone or bus with some medium such as twisted pair, coaxial cable, etc. Figure 1.26 shows the arrangement of computers in the linear bus topology.

Advantages of linear bus topology are:

- The linear bus topology usually requires less cabling.
- The linear bus topology is relatively simple to configure and install.
- In the linear bus topology, the failure of one computer does not affect the other computers in the network.

The following are the disadvantages of linear bus topology:

- In the linear bus topology, the failure of the backbone cable results in the breakdown of entire network.
- Addition of computers in the linear bus topology results in the performance degradation of the network.
- The bus topology is difficult to reconstruct in case of faults.

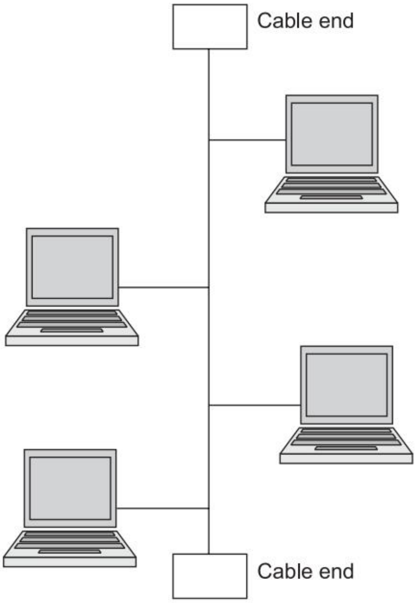


Fig. 1.26 *A linear bus topology*

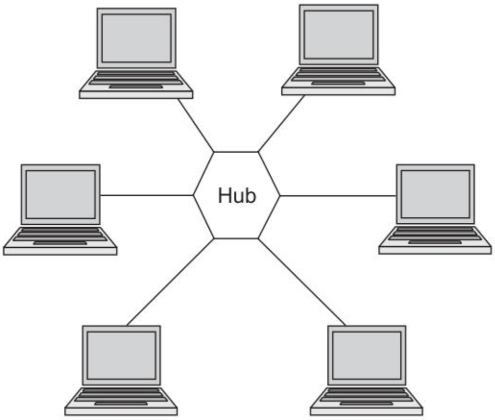


Fig. 1.27 *A star topology*

Star Topology

In the star topology, all the nodes are connected to a common device known as hub. Nodes are connected with the help of twisted pair, coaxial cable or optical fibre. Figure 1.27 shows the arrangement of computers in star topology.

Advantages of star topology are:

- This topology allows easy error detection and correction.
- In the star topology, the failure of one computer does not affect the other computers in the network.
- Star topology is easy to install.

The following are the disadvantages of star topology:

- In the star topology, the hub failure leads to the overall network crash.
- The star topology requires more amount of cable for connecting the nodes.
- It is expensive due to the cast of hub.

Ring Topology

In the ring topology, the nodes are connected in the form of a ring with the help of twisted pair. Each node is connected directly to the other two nodes in the network. Figure 1.28 shows the arrangement of computers in the ring topology.

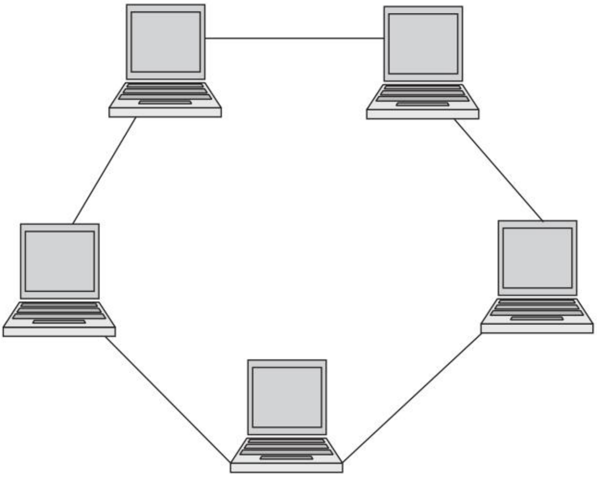


Fig. 1.28 *A ring topology*

Advantages of ring topology are:

- Each node has an equal access to other nodes in the network.
- Addition of new nodes does not degrade the performance of the network.
- Ring topology is easy to configure and install.

The following are the disadvantages of ring topology:

- It is relatively expensive to construct the ring topology.
- The failure of one node in the ring topology affects the other nodes in the ring.

Mesh Topology

In mesh topology, each computer is connected to every other computer in point-to-point mode as shown in Fig. 1.29. If we have n computers, we must have n(n – 1)/2 links.

Advantages of mesh topology are:

- Message delivery is more reliable.
- Network congestion is minimum due to large number of links.

The following are the disadvantages:

- It is very expensive to implement.
- It is very difficult to configure and install.

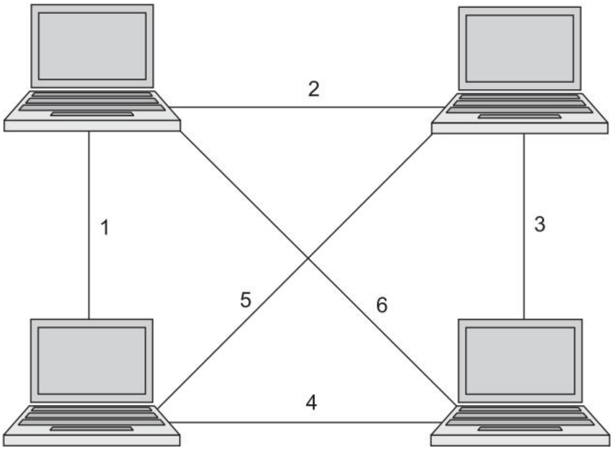


Fig. 1.29 *Mesh topology*

Hybrid Topology

The hybrid topology is the combination of multiple topologies, used for constructing a single large topology. Figure 1.30 shows a typical arrangement of computers in hybrid topology.

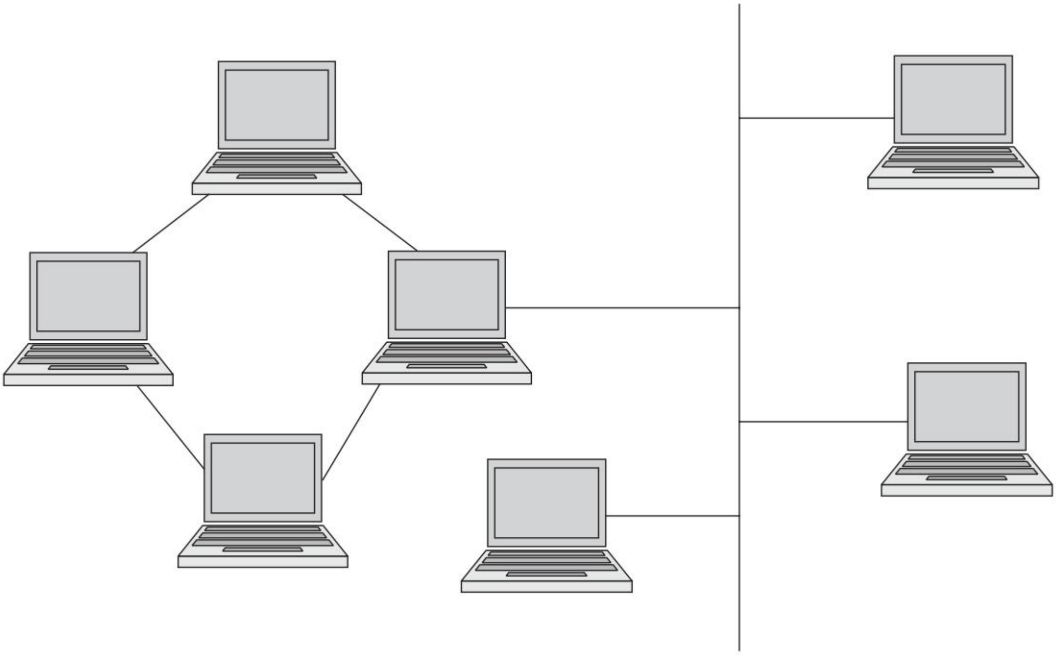


Fig. 1.30 *A hybrid topology*

Advantages of hybrid topology are:

- The hybrid topology is more effective as it uses multiple topologies.
- The hybrid topology contains the best and efficient features of the combined topologies from which it is constructed.

The following are the disadvantages of hybrid topology:

- The hybrid topology is relatively more complex than the other topologies.
- The hybrid topology is difficult to install and configure.

NETWORK PROTOCOLS AND SOFTWARE

In order to share data between computers, it is essential to have appropriate network protocols and software. With the help of network protocol, computers can easily communicate with each other and can share data, resources, etc.

Network Protocol

Network protocols are the set of rules and regulations that are generally used for communication between two networks. Using network protocol, the following tasks can be performed:

- Identification of the type of the physical connection used
- Error detection and correction of the improper message
- Initiation and termination of the communication session
- Message formatting

Some of the commonly used network protocols are Hyper Text Transfer protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), Transmission Control Protocol/Internet Protocol (TCP/IP), Telecommunications Network (Telnet), Domain Name System (DNS) etc.

HTTP

Hyper Text Transfer Protocol (HTTP) is the communication protocol used by the World Wide Web. It acts as a request-response protocol where the client browser and the Web server interact with each other through HTTP protocol rules. These rules define how messages are formatted and transmitted and what actions should the browser and Web server take in response to these messages. For example, when we type a URL in the address bar of a browser, then an HTTP request is sent to the Web server to fetch the requested Web page. The Web page details are transmitted to the client browser and rendered on the browser window through HTML.

In a typical situation, the client browser submits an HTTP request to the server and the server processes the request and returns an HTTP response to the client. The response contains status information pertaining to the request as well as the requested content (Figs 1.31–1.32).

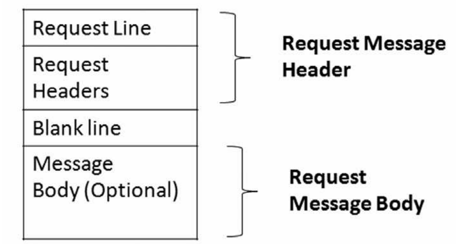


Fig. 1.31 *HTTP Request Message format*

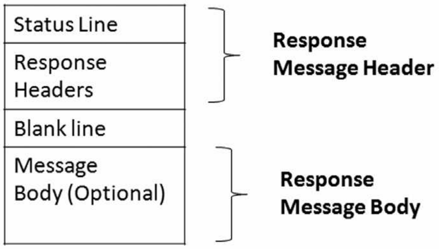


Fig. 1.32 *HTTP Response Message format*

HTTP protocol supports various methods that are used by the client browsers to send request messages to the server. Some of the common HTTP methods are:

- **GET**: Gets information from the specified resource
- **HEAD**: Gets only the HTTP headers
- **POST**: Posts information to the specified resource
- **DELETE**: Deletes the specified resource
- **OPTIONS**: Returns the list of HTTP methods that are supported by the Web server
- **TRACE**: Returns a diagnostic trace of the actions taken at the server end

The first line in an HTTP response object comprises a status line, which carries the response status code indicating the outcome of the HTTP request processed by the server. The status code is a 3-digit number and carries specific meaning, as described below:

- **1xx**: Comprises information status messages indicating that the server is still processing the request
- **2xx**: Comprises success status messages indicating that the request was received, accepted and processed by the server
- **3xx**: Comprises redirection status messages indicating that further action needs to be taken in order to process the request
- **4xx**: Comprises error status messages indicating error at client side, for example incorrect request syntax
- **5xx**: Comprises error status messages indicating error at server side, for example inability of the server to process the request

SMTP

Simple Mail Transfer Protocol (SMTP) is an e-mail protocol that is widely used for sending e-mail messages between mail servers. While SMTP supports capabilities for both sending and receiving e-mail messages, e-mail systems primarily used SMTP protocol for sending e-mail messages. For receiving, they use other protocols such as POP3 of IMAP. In Unix-based systems, sendmail is the most widely used SMTP server for e-mail. In Windows-based systems, Microsoft Exchange comes with an SMTP server and can be configured to include POP3 support.

FTP

File Transfer Protocol (FTP) is a standard protocol used for sharing files over the Internet. FTP is based on the client-server architecture and uses Internet's TCP/IP protocol for file transfer. The users need to authenticate themselves by specifying user name/password in order to establish a connection with the FTP server. However, some FTP sites also support anonymous login where users are not required to enter their credentials. To facilitate secure transfer of user's credentials and file contents over the Internet, FTP encrypts the content using cryptographic protocols such as TLS/SSL.

The following steps illustrate how file transfer happens through FTP:

1. The client machine uses Internet to connect to the FTP server's IP address.
2. User authentication happens by entering relevant user name and password.
3. Once the connection is established, the client machine sends FTP commands to access and transfer files. Now-a-days, various GUI-based FTP software are available that enable transfer of files through simple operations, such as drag and drop.

Telnet

Telnet is a protocol that allows users to connect to remote computers over a TCP/IP network, such as intranet or internet. While HTTP and FTP protocols are used for transferring Web pages and files over the Internet, the Telnet protocol is used for logging onto a remote computer and performing operations just as a normal user. The users need to enter their credentials before logging on the remote host machine.

Command-line based telnet access is available in major operating systems such as Windows, Mac OS, Unix and Linux. Generic format of the telnet command is given below:

```
Telnet host port
```

Here,

- **telnet**: Is the command that establishes telnet connection
- **host**: Is the address of the host machine
- **port**: Is the port number on which telnet services are available on the host machine

DECIMAL SYSTEM

The *decimal system* is the most common number system used by human beings. It is a positional number system that uses 10 as a base to represent different values. Therefore, this number system is also known as *base10 number system*. In this system, 10 symbols are available for representing the values. These symbols include the digits from 0 to 9. The common operations performed in the decimal system are addition (+), subtraction (–), multiplication (×) and division (/).

The decimal system can be used to represent both the integer as well as floating point values. The floating point values are generally represented in this system by using a period called decimal point. The decimal point is used to separate the integer part and the fraction part of the given floating point number. However, there is no need to use a decimal point for representing integer values. The value of any number represented in the decimal system can be determined by first multiplying the weight associated with each digit in the given number with the digit itself and then adding all these values produced as a result of multiplication operation. The weight associated with any digit depends upon the position of the digit itself in the given number. The most common method to determine the weight of any digit in any number system is to raise the base of the number system to a power that initially starts with a 0 and then increases by 1 as we move from right to left in the given number. To understand this concept, let us consider the following floating point number represented in the decimal system:

Decimal point
↓
6 5 4 3 . 1 2 4

In the above example, the value 6543, which comes before the decimal point, is called *integer value* and the value 124, which comes after the decimal point, is called *fraction value*. Table 1.1 lists the weights associated with each digit in the given decimal number.

Table 1.1 *Place Values in Decimal System*

Digit	6	5	4	3	.	1	2	4
Weight	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}

The above table shows that the powers to the base increases by 1 towards the left for the integer part and decreases by 1 towards the right for the fraction part. Using the place values, the floating point number 6543.124 in decimal system can be computed as:

$$
\begin{aligned}
& 6 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 1 \times 10^{-1} + 2 \times 10^{-2} + 4 \times 10^{-3} \\
&= 6000 + 500 + 40 + 3 + 0.1 + 0.02 + 0.004 \\
&= 6543.124
\end{aligned}
$$

BINARY SYSTEM

Among all the positional number systems, the binary system is the most dominant number system that is employed by almost all the modern digital computer systems. The binary system uses base 2 to represent different values. Therefore, the binary system is also known as *base-2 system*. As this system uses base 2, only two symbols are available for representing the different values in this system. These symbols are 0 and 1, which are also known as *bits* in computer terminology. Using binary system, the computer systems can store and process each type of data in terms of 0s and 1s only.

The following are some of the technical terms used in binary system:

- **Bit.** It is the smallest unit of information used in a computer system. It can either have the value 0 or 1. Derived from the words **B**inary dig**it**.
- **Nibble.** It is a combination of 4 bits.
- **Byte.** It is a combination of 8 bits. Derived from words ‘by eight’.
- **Word.** It is a combination of 16 bits.
- **Double word.** It is a combination of 32 bits.
- **Kilobyte (KB).** It is used to represent the 1024 bytes of information.
- **Megabyte (MB).** It is used to represent the 1024 KBs of information.
- **Gigabyte (GB).** It is used to represent the 1024 MBs of information.

We can determine the weight associated with each bit in the given binary number in the similar manner as we did in the decimal system. In the binary system, the weight of any bit can be determined by raising 2 to a power equivalent to the position of bit in the number. To understand this concept, let us consider the following binary number:

Binary point

$$\overset{\downarrow}{1\ 0\ 1\ 00\ 1\ .\ 0\ 1\ 0\ 1}$$

In binary system, the point used to separate the integer and the fraction part of a number is known as *binary point*. Table 1.2 lists the weights associated with each bit in the given binary number.

Table 1.2 *Place Values in Binary System*

Digit	1	0	1	0	0	1	.	0	1	0	1
Weight	2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}

Like the decimal system, the powers to the base increases by 1 towards the left for the integer part and decreases by 1 towards the right for the fraction part. The value of the given binary number can be determined as the sum of the products of the bits multiplied by the weight of the bit itself. Therefore, the value of the binary number 101001.0101 can be obtained as:

$$\begin{aligned} & 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} \\ &= 32 + 8 + 1 + 0.25 + 0.0625 \\ &= 41.3125 \end{aligned}$$

The binary number 101001.0101 represents the decimal value 41.3125.

Table 1.3 lists the 4-bit binary representation of decimal numbers 0 through 15.

Table 1.3 *Binary Representation of First 16 Numbers*

Decimal Number	*4-bit Binary*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

HEXADECIMAL SYSTEM

The hexadecimal system is a positional number system that uses base 16 to represent different values. Therefore, this number system is known as *base-16 system*. As this system uses base 16, 16 symbols are available for representing the values in this system. These symbols are the digits 0–9 and the letters A, B, C, D, E and F. The digits 0–9 are used to represent the decimal values 0 through 9 and the letters A, B, C, D, E and F are used to represent the decimal values 10 through 15.

LO 1.8
Identify the various positional number systems

The weight associated with each symbol in the given hexadecimal number can be determined by raising 16 to a power equivalent to the position of the digit in the number. To understand this concept, let us consider the following hexadecimal number:

Hexadecimal point

4 A 9 . 2 B

In hexadecimal system, the point used to separate the integer and the fraction part of a number is known as *hexadecimal point*. Table 1.4 lists the weights associated with each digit in the given hexadecimal number.

Table 1.4 *Place Values in Hexadecimal System*

Digit	4	A	9	.	2	B
Weight	16^2	16^1	16^0		16^{-1}	16^{-2}

The value of the hexadecimal number can be computed as the sum of the products of the symbol multiplied by the weight of the symbol itself. Therefore, the value of the given hexadecimal number is:

$$\begin{aligned} & 4 \times 16^2 + 10 \times 16^1 + 9 \times 16^0 + 2 \times 16^{-1} + 11 \times 16^{-2} \\ &= 1024 + 160 + 9 + 0.125 + 0.0429 \\ &= 1193 + 0.1679 \\ &= 1193.1679 \end{aligned}$$

The hexadecimal number 4A9. 2B represents the decimal value 1193.1679.

OCTAL SYSTEM

The octal system is the positional number system that uses base 8 to represent different values. Therefore, this number system is also known as *base-8 system*. As this system uses base 8, eight symbols are available for representing the values in this system. These symbols are the digits 0 to 7.

The weight associated with each digit in the given octal number can be determined by raising 8 to a power equivalent to the position of digit in the number. To understand this concept, let us consider the following octal number:

Octal point
↓
215.43

In octal system, the point used to separate the integer and the fraction part of a number is known as *octal point*. Table 1.5 lists the weights associated with each digit in the given octal number.

Table 1.5 *Place Values in Octal System*

Digit	2	1	5	.	4	3
Weight	8^2	8^1	8^0		8^{-1}	8^{-2}

Using these place values, we can now determine the value of the given octal number as:

$$\begin{aligned} & 2 \times 8^2 + 1 \times 8^1 + 5 \times 8^0 + 4 \times 8^{-1} + 3 \times 8^{-2} \\ &= 128 + 8 + 5 + 0.5 + 0.0469 \\ &= 141 + 0.5469 \\ &= 141.5469 \end{aligned}$$

The octal number 215.43 represents the decimal value 141.5469.

Table 1.6 lists the octal representation of decimal numbers 0 through 15.

Table 1.6 *Octal Representation of First 16 Numbers*

Decimal Number	*Octal Representation*
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17

CONVERSION OF NUMBERS

The computer systems accept the data in decimal form, whereas they store and process the data in binary form. Therefore, it becomes necessary to convert the numbers represented in one system into the numbers represented in another system. The different types of number system conversions can be divided into the following major categories:

LO 1.9
Carry out number conversions from one number system to another

- Non-decimal to decimal
- Decimal to non-decimal
- Octal to hexadecimal

Non-Decimal to Decimal

The non-decimal to decimal conversions can be implemented by taking the concept of place values into consideration. The non-decimal to decimal conversion includes the following number system conversions:

- Binary to decimal conversion
- Hexadecimal to decimal conversion
- Octal to decimal conversion

Binary to decimal conversion A binary number can be converted to equivalent decimal number by calculating the sum of the products of each bit multiplied by its corresponding place value.

Example 1.1 *Convert the binary number 10101101 into its corresponding decimal number.*

Solution

The given binary number is 10101101.

Now, calculate the sum of the products of each bit multiplied by its place value as:

$(1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

$= 128 + 0 + 32 + 0 + 8 + 4 + 0 + 1$

$= 173$

Therefore, the binary number 10101101 is equivalent to 173 in the decimal system.

Example 1.2 *Convert the binary number 1101 into its equivalent in decimal system.*

Solution

The given binary number is 1101.

Now, calculate the sum of the products of each bit multiplied by its place value as:

$(1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

$= 8 + 4 + 1$

$= 13$

Therefore, the binary number 1101 is equivalent to 13 in the decimal system.

Example 1.3 *Convert the binary number 10110001 into its equivalent in decimal system.*

Solution

The given binary number is 10110001.

Now, calculate the sum of the products of each bit multiplied by its place value as:

$(1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$

$= 128 + 0 + 32 + 16 + 0 + 0 + 0 + 1$

$= 177$

Therefore, the binary number 10110001 is equivalent to 177 in the decimal system.

Example 1.4 *Convert the binary number 1011.010 into its equivalent in decimal system.*

Solution

The given binary number is 1011.010.

Now, calculate the sum of the products of each bit multiplied by its place value as:

$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3})$

$= 8 + 2 + 1 + \frac{1}{4}$

$= 11 + 0.25$

$= 11.25$

Therefore, the binary number 1011.010 is equivalent to 11.25 in the decimal system.

Example 1.5 *Convert the binary number 11011.0110 to its equivalent in decimal system.*

Solution

The given binary number is 11011.0110.

Now, calculate the sum of the products of each bit multiplied by its place value as:

$(1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) + (0 \times 2^{-4})$

$= 16 + 8 + 1 + \frac{1}{4} + \frac{1}{8}$

= 27 + 0.25 + 0.125

= 27.375

Therefore, the binary number 11011.0110 is equivalent to 27.375 in the decimal system.

Hexadecimal to decimal conversion A hexadecimal number can be converted into its equivalent number in decimal system by calculating the sum of the products of each symbol multiplied by its corresponding place value.

Example 1.6 *Convert the hexadecimal number A53 into its equivalent in decimal system.*

Solution

The given hexadecimal number is A53.

Now, calculate the sum of the products of each symbol multiplied by its place value as:

$(10 \times 16^2) + (5 \times 16^1) + (3 \times 16^0)$

$= 2560 + 80 + 3$

$= 2643$

Therefore, the hexadecimal number A53 is equivalent to 2643 in the decimal system.

Example 1.7 *Convert the hexadecimal number 6B39 into its equivalent in the decimal system.*

Solution

The given hexadecimal number is 6B39.

Now, calculate the sum of the products of each symbol multiplied by its place value as:

$(6 \times 16^3) + (11 \times 16^2) + (3 \times 16^1) + (9 \times 16^0)$

$= 24576 + 2816 + 48 + 9$

$= 27449$

Therefore, the hexadecimal number 6B39 is equivalent to 27449 in the decimal system.

Example 1.8 *Convert the hexadecimal number 5A6D into its equivalent in the decimal system.*

Solution

The given hexadecimal number is 5A6D.

Now, calculate the sum of the products of each symbol multiplied by its place value as:

$(5 \times 16^3) + (10 \times 16^2) + (6 \times 16^1) + (13 \times 16^0)$

$= 20480 + 2560 + 96 + 13$

$= 23149.$

Therefore, the hexadecimal number 5A6D is equivalent to 23149 in the decimal system.

Example 1.9 *Convert the hexadecimal number AB21.34 into its equivalent in the decimal system.*

Solution

The given hexadecimal number is AB21.34.

Now, calculate the sum of the products of each symbol multiplied by its place value as:

$(10 \times 16^3) + (11 \times 16^2) + (2 \times 16^1) + (1 \times 16^0) + (3 \times 16^{-1}) + (4 \times 16^{-2})$

$$= 40960 + 2816 + 32 + 1 + \frac{3}{16} + \frac{4}{256}$$

$= 43809 + 0.1875 + 0.015625$

$= 43809.203$

Therefore, the hexadecimal number AB21.34 is equivalent to 43809.203 in the decimal system.

Example 1.10 *Convert the hexadecimal number 6A11.3B into its equivalent in the decimal system.*

Solution

The given hexadecimal number is 6A11.3B.

Now, calculate the sum of the products of each symbol multiplied by its place value as:

$(6 \times 16^3) + (10 \times 16^2) + (1 \times 16^{-1}) + (1 \times 16^0) + (3 \times 16^{-1}) + (11 \times 16^{-2})$

$$= 24576 + 2560 + 16 + 1 + \frac{3}{16} + \frac{11}{256}$$

$= 27153 + 0.1875 + 0.043$

$= 27153.2305$

Therefore, the hexadecimal number 6A11.3B is equivalent to 27153.2305 in the decimal system.

Octal to decimal conversion An octal number can be converted into its equivalent number in decimal system by calculating the sum of the products of each digit multiplied by its corresponding place value.

Example 1.11 *Convert the octal number 5324 into its equivalent in decimal system.*

Solution

The given octal number is 5324.

Now, calculate the sum of the products of each digit multiplied by its place value as:

$(5 \times 8^3) + (3 \times 8^2) + (2 \times 8^1) + (4 \times 8^0)$

$= 2560 + 192 + 16 + 4$

$= 2772$

Therefore, the octal number 5324 is equivalent to 2772 in the decimal system.

Example 1.12 *Convert the octal number 13256 into its equivalent in decimal system.*

Solution

The given octal number is 13256.

Now, calculate the sum of the products of each digit multiplied by its place value as:

$(1 \times 8^4) + (3 \times 8^3) + (2 \times 8^2) + (5 \times 8^1) + (6 \times 8^0)$

$= 4096 + 1536 + 128 + 40 + 6$

$= 5806$

Therefore, the octal number 13256 is equivalent to 5806 in the decimal system.

Example 1.13 *Convert the octal number 4567 into its equivalent in decimal system.*

Solution

The given octal number is 4567.

Now, calculate the sum of the products of each digit multiplied by its place value as:

$(4 \times 8^3) + (5 \times 8^2) + (6 \times 8^1) + (7 \times 8^0)$

$= 2048 + 320 + 48 + 7$

$= 2423$

Therefore, the octal number 4567 is equivalent to 2423 in the decimal system.

Example 1.14 *Convert the octal number 325.12 into its equivalent in decimal system.*

Solution

The given octal number is 325.12.

Now, calculate the sum of the products of each digit multiplied by its place value as:

$(3 \times 8^2) + (2 \times 8^1) + (5 \times 8^0) + (1 \times 8^{-1}) + (2 \times 8^{-2})$

$$= 192 + 16 + 5 + \frac{1}{8} + \frac{2}{64}$$

= 213 + 0.125 + 0.03125

= 213.15625

Therefore, the octal number 325.12 is equivalent to 213.15625 in the decimal system.

> ***Example 1.15*** *Convert the octal number 7652.01 into its equivalent in decimal system.*

Solution

The given octal number is 7652.01.

Now, calculate the sum of the products of each digit multiplied by its place value as:

$$(7 \times 8^3) + (6 \times 8^2) + (5 \times 8^1) + (2 \times 8^0) + (0 \times 8^{-1}) + (1 \times 8^{-2})$$

$$= 3584 + 384 + 40 + 2 + \frac{1}{64}$$

= 4010 + 0.015625

= 4010.0156

Therefore, the octal number 7652.01 is equivalent to 4010.0156 in the decimal system.

Decimal to Non-Decimal

The decimal to non-decimal conversions are carried out by continually dividing the decimal number by the base of the desired number system till the decimal number becomes zero. After the decimal number becomes zero, we may note down the remainders calculated at each successive division from last to first to obtain the decimal number into the desired system. The decimal to non-decimal conversion includes the following number system conversions:

- Decimal to binary conversion
- Decimal to hexadecimal conversion
- Decimal to octal conversion

Decimal to binary conversion The decimal to binary conversion is performed by repeatedly dividing the decimal number by 2 till the decimal number becomes zero and then reading the remainders from last to first to obtain the binary equivalent of the given decimal number. The following examples illustrate the method of converting decimal number to its binary equivalent:

> ***Example 1.16*** *Convert the decimal number 30 into its equivalent binary number.*

Solution

The given decimal number is 30.

The following table lists the steps showing the conversion of the given decimal number to its binary equivalent:

Decimal Number	*Divisor*	*Quotient*	*Remainder*
30	2	15	0
15	2	7	1
7	2	3	1
3	2	1	1
1	2	0	1

Now, read the remainders calculated in the above table in upward direction to obtain the binary equivalent, which is 11110.

Therefore, the binary equivalent of the decimal number 30 is 11110.

Example 1.17 *Convert the decimal number 111 into its equivalent binary number.*

Solution

The given decimal number is 111.

The following table lists the steps showing the conversion of the given decimal number to its binary equivalent:

Decimal Number	Divisor	Quotient	Remainder
111	2	55	1
55	2	27	1
27	2	13	1
13	2	6	1
6	2	3	0
3	2	1	1
1	2	0	1

Now, read the remainders calculated in the above table in upward direction to obtain the binary equivalent, which is 1101111.

Therefore, the binary equivalent of the decimal number 111 is 1101111.

Example 1.18 *Convert the decimal number 215 into its equivalent binary number.*

Solution

The given decimal number is 215.

The following table lists the steps showing the conversion of the given decimal number to its binary equivalent:

Decimal Number	Divisor	Quotient	Remainder
215	2	107	1
107	2	53	1
53	2	26	1
26	2	13	0
13	2	6	1
6	2	3	0
3	2	1	1
1	2	0	1

Now, read the remainders calculated in the above table in upward direction to obtain the binary equivalent, which is 11010111.

Therefore, the binary equivalent of the decimal number 215 is 11010111.

The procedure of converting the fractional part of the given decimal number to its binary equivalent is different. In this procedure, we need to continually multiply the fractional part by 2 and then note down the whole number part of the result. The multiplication process will terminate when the fractional part becomes zero or when we have achieved the desired number of bits.

Example 1.19 *Convert the decimal number 45796 to its equivalent octal number.*

Solution

The given decimal number is 45796.

The following table lists the steps showing the conversion of the given decimal number to its octal equivalent:

Decimal Number	Divisor	Quotient	Remainder
45796	8	5724	4
5724	8	715	4
715	8	89	3
89	8	11	1
11	8	1	3
1	8	0	1

Now, read the remainders calculated in the above table in upward direction to obtain the octal equivalent, which is 131344.

Therefore, the corresponding octal equivalent of 45796 is 131344.

Example 1.20 *Convert the decimal number 9547 into its equivalent octal number.*

Solution

The given decimal number is 9547.

The following table lists the steps showing the conversion of the given decimal number to its octal equivalent:

Decimal Number	Divisor	Quotient	Remainder
9547	8	1193	3
1193	8	149	1
149	8	18	5
18	8	2	2
2	8	0	2

Now, read the remainders calculated in the above table in upward direction to obtain the octal equivalent, which is 22513.

Therefore, the corresponding octal equivalent of 9547 is 22513.

Example 1.21 *Convert the decimal number 1567 into its equivalent hexadecimal number.*

Solution

The given decimal number is 1567.

The following table lists the steps showing the conversion of the given decimal number to its hexadecimal equivalent:

Decimal Number	*Divisor*	*Quotient*	*Remainder*
1567	16	97	15
97	16	6	1
6	16	0	6

Now, read the remainders calculated in the above table in upward direction to obtain the hexadecimal equivalent, which is 61F.

Therefore, the hexadecimal equivalent of the given decimal number is 61F.

Example 1.22 *Convert the decimal number 9463 into its equivalent hexadecimal number.*

Solution

The given decimal number is 9463.

The following table lists the steps showing the conversion of the given decimal number to its hexadecimal equivalent:

Decimal Number	*Divisor*	*Quotient*	*Remainder*
9463	16	591	7
591	16	36	15
36	16	2	4
2	16	0	2

Now, read the remainders calculated in the above table in upward direction to obtain the hexadecimal equivalent, which is 24F7.

Therefore, the hexadecimal equivalent of the given decimal number is 24F7.

Decimal to octal conversion The decimal to octal conversion is performed by repeatedly dividing the decimal number by 8 till the decimal number becomes zero and reading the remainders from last to first to obtain the octal equivalent of the given decimal number. The following examples illustrate the method of converting decimal number to its octal equivalent:

Example 1.23 *Convert the decimal number 45796 to its equivalent octal number.*

Solution

The given decimal number is 45796.

The following table lists the steps showing the conversion of the given decimal number to its octal equivalent:

Decimal Number	*Divisor*	*Quotient*	*Remainder*
45796	8	5724	4
5724	8	715	4
715	8	89	3
89	8	11	1
11	8	1	3
1	8	0	1

Now, read the remainders calculated in the above table in upward direction to obtain the octal equivalent, which is 131344.

Therefore, the corresponding octal equivalent of 45796 is 131344.

Example 1.24 *Convert the decimal number 9547 into its equivalent octal number.*

Solution

The given decimal number is 9547.

The following table lists the steps showing the conversion of the given decimal number to its octal equivalent:

Decimal number	*Divisor*	*Quotient*	*Remainder*
9547	8	1193	3
1193	8	149	1
149	8	18	5
18	8	2	2
2	8	0	2

Now, read the remainders calculated in the above table in upward direction to obtain the octal equivalent, which is 22513.

Therefore, the corresponding octal equivalent of 9547 is 22513.

Octal to Hexadecimal

The given octal number can be converted into its equivalent hexadecimal number in two different steps. Firstly, we need to convert the given octal number into its binary equivalent. After obtaining the binary equivalent, we need to divide the binary number into 4-bit sections starting from the LSB.

The octal to binary conversion is a simple process. In this type of conversion, we need to represent each digit in the octal number to its equivalent 3-bit binary number. Table 1.7 lists the binary representation of all the digits used in an octal system.

Table 1.7 *Binary Representation of Octal Symbols*

Octal	*Binary Representation*
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Example 1.25 *Convert the octal number 365 into its equivalent hexadecimal number.*

Solution

Octal number:	3	6	5	
↓	↓	↓	↓	
Binary equivalent:	011	110	101	*Step 1*
↓				
Regrouping into 4-bit sections:	0000	1111	0101	*Step 2*
↓	↓	↓	↓	
Hexadecimal equivalent:	0	F	5	*Step 3*

Hexadecimal number is F5

Example 1.26 *Convert the octal number 6251 into its equivalent hexadecimal number.*

Solution

Octal number:	6	2	5	1	
↓	↓	↓	↓	↓	
Binary equivalent:	110	010	101	001	*Step 1*
↓					
4-bits grouping:	1100	1010	1001		*Step 2*
↓	↓	↓	↓		
Hexadecimal equivalent:	C	A	9		*Step 3*

Hexadecimal number is CA9

BINARY ARITHMETIC OPERATIONS

The computer arithmetic is also referred as *binary arithmetic* because the computer system stores and processes the data in the binary form only. Various binary arithmetic operations can be performed in the same way as the decimal arithmetic operations, but by following a predefined set of rules. Each binary arithmetic operation has an associated set of rules that should be adhered to while carrying out that operation. The binary arithmetic operations are usually simpler to carry out as compared to the decimal operations because one needs to deal with only two digits, 0 and 1, in the binary operations. The different binary arithmetic operations performed in a computer system are:

LO 1.10
Explain how binary arithmetic operations are performed

- Binary addition
- Binary multiplication
- Binary subtraction
- Binary division

Binary Addition

Binary addition is the simplest arithmetic operation performed in the computer system. Like decimal system, we can start the addition of two binary numbers column-wise from the right-most bit and move towards the left-most bit of the given numbers. However, we need to follow certain rules while carrying out the binary addition of the given numbers. Table 1.8 lists the rules for binary addition.

Table 1.8 *Binary Addition Rules*

A	*B*	*A + B*	*Carry*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

In the above table, the first three entries do not generate any carry. However, a carry would be generated when both A and B contain the value, 1. The carry, if it is generated, while performing the binary addition in a column would be forwarded to the next most significant column.

Example 1.27 *Perform the binary addition operation on the following binary numbers:*

0 0 1 0
0 1 1 1

Solution

The given binary numbers are 0010 and 0111.
Now, perform the binary addition of the given numbers as:

Binary number	**Decimal value**
0 0 1 0	2
0 1 1 1	7
1 0 0 1	9

Therefore, the result of the binary addition performed on 0010 and 0111 is 1001.

Note *In the above example, a carry is generated in the 2nd and the 3rd column only.*

Example 1.28 *Perform the binary addition of the following binary numbers:*

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

Solution

The given binary numbers are 101010 and 010011.
Now, perform the binary addition of the given numbers as:

Binary number	**Decimal value**
1 0 1 0 1 0	42
0 1 0 0 1 1	19
1 1 1 1 0 1	61

Therefore, the result of the binary addition performed on 101010 and 010011 is 111101.

Note *In the above example, a carry is generated in the 2nd column only.*

Example 1.29 *Evaluate the binary sum of the following numbers:*

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

Solution

The given binary numbers are 00011010 and 10001100.

Now, perform the binary addition of the given numbers as:

Binary number	Decimal value
0 0 0 1 1 0 1 0	26
1 0 0 0 1 1 0 0	140
1 0 1 0 0 1 1 0	166

Therefore, the result of the binary addition performed on 00011010 and 10001100 is 10100110.

Note *In the above example, a carry is generated in the 4th and the 5th column only.*

We can also perform the binary addition on more than two binary numbers. Table 1.9 lists the rules for adding three binary numbers.

Table 1.9 *Rules for Adding Three Binary Numbers*

A	*B*	*C*	*A + B + C*	*Carry*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

To understand the concept of triple binary addition, let us consider the following examples:

Example 1.30 *Perform the binary addition operation on the following three numbers:*

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

Solution

The given binary numbers are 0010, 0001 and 0111.
Now, perform the binary addition of the given numbers as:

Binary number	Decimal value
0 0 1 0	2
0 0 0 1	1
0 1 1 1	7
1 0 1 0	10

Therefore, the result of the binary addition performed on 0010, 0001 and 0111 is 1010.

Note *In the above example, a carry is generated in the 1st and the 2nd column only.*

Example 1.31 *Evaluate the binary sum of the following numbers:*

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

Solution

The given binary numbers are 01010, 00110 and 01111.

Now, perform the binary addition of the given numbers as:

```
Binary number          Decimal value
 0  1  0  1  0              10
 0  0  1  1  0               6
 0  1  1  1  1              15
---------------        ------------
 1  1  1  1  1              31
---------------        ------------
```

Therefore, the result of the binary addition performed on 01010, 00110 and 01111 is 11111.

Note *In the above example, a carry is generated in the 2nd, 3rd and 4th column only.*

Binary Multiplication

The multiplication of two binary numbers can be carried out in the same manner as the decimal multiplication. However, unlike decimal multiplication, only two values are generated as the outcome of multiplying the multiplicand bit by 0 or 1 in the binary multiplication. These values are either 0 or 1. The binary multiplication can also be considered as repeated binary addition. For instance, when we are multiplying 7 with 3, it simply means that we are adding 7 to itself 3 times. Therefore, the binary multiplication is performed in conjunction with the binary addition operation. Table 1.10 lists the rules for binary multiplication.

Table 1.10 *Binary Multiplication Rules*

A	*B*	*A* × *B*
0	0	0
0	1	0
1	0	0
1	1	1

The above table clearly shows that binary multiplication does not involve the concept of carry. To understand the concept of binary multiplication, let us consider the following examples:

Example 1.32 *Perform the binary multiplication of the decimal numbers 12 and 10.*

Solution

The equivalent binary representation of the decimal number 12 is 1100.

The equivalent binary representation of the decimal number 10 is 1010.

Now, perform the binary multiplication of the given numbers as:

```
          1  1  0  0        Multiplicand
          1  0  1  0        Multiplier
        ------------
          0  0  0  0        First partial product
       1  1  0  0
    0  0  0  0
 1  1  0  0
--------------------
 1  1  1  1  0  0  0        Final product
--------------------
```

Therefore, the result of the binary multiplication performed on the decimal numbers 12 and 10 is 1111000.

Example 1.33 *Evaluate the binary product of the decimal numbers 15 and 14.*

Solution

The equivalent binary representation of the decimal number 15 is 1111.

The equivalent binary representation of the decimal number 14 is 1110.

Now, perform the binary multiplication of the given numbers as:

```
        1 1 1 1     Multiplicand
        1 1 1 0     Multiplier
      ---------
        0 0 0 0     First partial product
      1 1 1 1
    1 1 1 1
  1 1 1 1
---------------
1 1 0 1 0 0 1 0     Final product
---------------
```

Therefore, the result of the binary multiplication performed on the decimal numbers 15 and 14 is 11010010.

Example 1.34 *Perform the binary multiplication of the following numbers:*

1101

111

Solution

The given binary numbers are 1101 and 111.

Now, perform the binary multiplication of the given numbers as:

```
      1 1 0 1     Multiplicand
        1 1 1     Multiplier
    ---------
      1 1 0 1     First partial product
    1 1 0 1
  1 1 0 1
-------------
1 0 1 1 0 1 1     Final product
-------------
```

Therefore, the result of the binary multiplication performed on the numbers 1101 and 111 is 1011011.

Example 1.35 *Evaluate the binary product of the following numbers:*

100010

10010

Solution

The given binary numbers are 100010 and 10010.

Now, perform the binary multiplication of the given numbers as:

```
        1 0 0 0 1 0     Multiplicand
          1 0 0 1 0     Multiplier
        -------------
        0 0 0 0 0 0     First partial product
      1 0 0 0 1 0
    0 0 0 0 0 0
  0 0 0 0 0 0
1 0 0 0 1 0
-------------------
1 0 0 1 1 0 0 1 0 0     Final product
-------------------
```

Therefore, the result of the binary multiplication performed on the numbers 100010 and 10010 is 1001100100.

Binary Subtraction

The binary subtraction is performed in the same way as the decimal subtraction. Like binary addition and binary multiplication, binary subtraction is also associated with a set of rules that need to be followed while carrying out the operation. Table 1.11 lists the rules for binary subtraction.

Table 1.11 *Binary Subtraction Rules*

A	*B*	*A – B*	*Borrow*
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

The above table shows that the binary subtraction like the decimal subtraction uses the borrow method to subtract one number from another. To understand the concept of binary subtraction, let us consider the following examples:

Example 1.36 *Subtract the following binary numbers:*
0 1 0 1
0 0 1 0

Solution

The given binary numbers are 0101 and 0010.

Now, perform the binary subtraction of the given numbers as:

```
        1        Borrow
0   1   0   1    Minuend
0   0   1   0    Subtrahend
-------------
0   0   1   1    Difference
-------------
```

Therefore, the result of the binary subtraction performed on the numbers 0101 and 0010 is 0011.

Example 1.37 *Perform the binary subtraction of the following numbers:*
1 0 1 0 1
0 1 1 1 0

Solution

The given binary numbers are 10101 and 01110.

Now, perform the binary subtraction of the given numbers as:

```
    1   1   1        Borrow
1   0   1   0   1    Minuend
0   1   1   1   0    Subtrahend
-----------------
0   0   1   1   1    Difference
-----------------
```

Therefore, the result of the binary subtraction performed on the numbers 10101 and 01110 is 00111

Example 1.38 *Perform the binary subtraction of the following numbers:*
10111011
01001001

Solution

The given binary numbers are 10111011 and 01001001.

Now, perform the binary subtraction of the given numbers as:

	1							Borrow
1	0	1	1	1	0	1	1	Minuend
0	1	0	0	1	0	0	1	Subtrahend
0	1	1	1	0	0	1	0	Difference

Therefore, the result of the binary subtraction performed on the numbers 10111011 and 01001001 is 1110010.

Example 1.39 *Perform the binary subtraction of the following numbers:*
101110101010
001111011100

Solution

The given binary numbers are 101110101010 and 001111011100.

Now, perform the binary subtraction of the given numbers as:

	1	1	1	1	1		1	1	1			Borrow
1	0	1	1	1	0	1	0	1	0	1	0	Minuend
0	0	1	1	1	1	0	1	1	1	0	0	Subtrahend
0	1	1	1	1	1	0	0	1	1	1	0	Difference

Therefore, the result of the binary subtraction performed on the numbers 101110101010 and 001111011100 is 11111001110.

Binary Division

Binary division is also performed in the same way as we perform decimal division. Like decimal division, we also need to follow the binary subtraction rules while performing the binary division. The dividend involved in binary division should be greater than the divisor. The following are the two important points, which need to be remembered while performing the binary division:

- If the remainder obtained by the division process is greater than or equal to the divisor, put 1 in the quotient and perform the binary subtraction.
- If the remainder obtained by the division process is less than the divisor, put 0 in the quotient and append the next most significant digit from the dividend to the remainder.

Example 1.40 *Divide 14 by 7 in binary form.*

Solution

The equivalent binary representation of the decimal number 14 is 1110.

The binary representation of 7 is 111.

Now, perform the binary division of the given numbers as:

```
1 1 1 ) 1 1 1 0 (10   (Quotient)
        1 1 1
        -------
        0 0 0 0
        -------
```

Therefore, the result of the binary division performed on the decimal numbers 14 and 7 is 10.

Example 1.41 *Perform the binary division of the decimal numbers 18 and 8.*

Solution

The equivalent binary representation of the decimal number 18 is 10010.

The equivalent binary representation of the decimal number 8 is 1000.

Now, perform the binary division of the given numbers as:

```
1 0 0 0 ) 1 0 0 1 0 ( 1 0      (Quotient)
          1 0 0 0
          ---------
          0 0 0 1 0
          0 0 0 0 0
          ---------
          0 0 0 1 0            (Remainder)
          ---------
```

Therefore, the result of the binary division performed on the decimal numbers 18 and 8 is 10 with a remainder of 10.

Example 1.42 *Perform the binary division of the decimal numbers 11011 and 1001.*

Solution

The given binary numbers are 11011 and 1001.

Now, perform the binary division of the given binary numbers as:

```
1 0 0 1 ) 1 1 0 1 1 ( 1 1      (Quotient)
          1 0 0 1
          -------
            1 0 0 1
            1 0 0 1
            -------
            0 0 0 0
            -------
```

Therefore, the result of the binary division performed on the numbers 11011 and 1001 is 11.

Example 1.43 *Perform the binary division of 217 and 12.*

Solution

The equivalent binary representation of the decimal number 217 is 11011001.

The equivalent binary representation of the decimal number 12 is 1100.

Now, perform the binary division of the given numbers as:

```
1 1 0 0 ) 1 1 0 1 1 0 0 1 ( 1 0 0 1 0      (Quotient)
          1 1 0 0
          -------
          0 0 0 1 1
          0 0 0 0 0
         -----------
              0 1 1 0
              0 0 0 0
              -------
              1 1 0 0
              1 1 0 0
              -------
              0 0 0 1                      (Remainder)
```

Therefore, the result of the binary division performed on the decimal number 217 and 12 is 10010 with a remainder of 1.

LOGIC GATES

LO 1.11
Describe primary logic gates

Logic gates are the basic building blocks of a digital computer. In general, all the logic gates have two input signals and one output signal. These two input signals are nothing but two binary values, 0 or 1 that helps represent different voltage levels. In all logic gates, the binary value 0 represents the low state of voltage that is approximately 0 volt and the binary value 1 represents the high state of voltage that is approximately +5 volts. The three basic logic gates are:

- AND
- OR
- NOT

All logic gates have a logical expression, symbol, and truth table. The logical expression helps find the output of the logic gate on the basis of its inputs. A symbol is the pictorial presentation of a logic gate that can have one or more than one input and one output. The truth table helps find the final logical state, such as true/false or 1/0 of the logic gate in the form of its output.

AND Gate

The AND gate is one of the basic logic gates that give an output signal of value 1 only when all its input signals are of value 1. In other words, the AND gate gives an output signal of value 0 whenever its one input signal is of value 0.

Logical Expression

The logical expression for the AND function is:

$$F = A.B$$

where, F is the output that depends on inputs, A and B.

Symbol

The symbol of the AND gate is shown in Fig. 1.33.

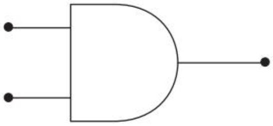


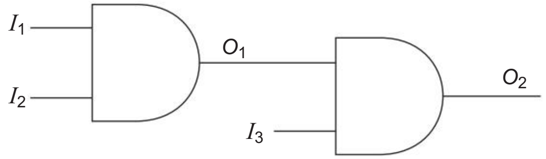
Fig. 1.33 *AND gate*

Truth Table

Table 1.12 *Truth Table for AND Gate*

Input A	*Input B*	*Output F*
0	0	0
0	1	0
1	0	0
1	1	1

Example 1.44 *Consider the following system that has two AND gates:*



Solution

Assuming

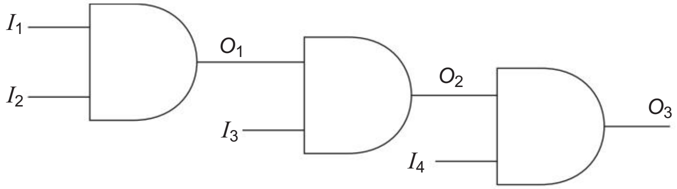
$$I_1 = 1, \quad I_2 = 0 \quad \text{and} \quad I_3 = 0$$

Outputs would be

$$O_1 = I_1{\cdot}I_2 = 1{\cdot}0 = 0$$

$$O_2 = I_3{\cdot}O_1 = 0{\cdot}0 = 0$$

Example 1.45 *Consider the following system with three AND gates:*



Solution

Assuming

$$I_1 = 1, I_2 = 1, I_3 = 1 \quad \text{and} \quad I_4 = 1$$

Outputs would be:

$$O_1 = I_1{\cdot}I_2 = 1{\cdot}1 = 1$$

$$O_2 = I_3{\cdot}O_1 = 1{\cdot}1 = 1$$

$$O_3 = I_4{\cdot}O_2 = 1{\cdot}1 = 1$$

OR Gate

The OR gate is another basic logic gate that gives an output signal of value 1 whenever its one input signal is of value 1. In other words, the OR gate gives an output signal of value 0 when all its input signals are of value 0.

Logical Expression

The logical expression for the OR function is:

$$F = A + B$$

where, *F* is the output that depends on inputs *A* and *B*.

Symbol

The symbol of the OR gate is shown in Fig. 1.34.

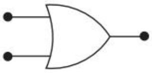


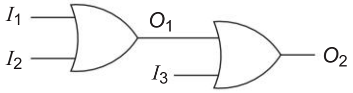
Fig. 1.34 *OR Gate*

`Truth Table`

Table 1.13 *Truth Table for OR Gate*

Input A	*Input B*	*Output F*
0	0	0
0	1	1
1	0	!
1	1	1

Example 1.46 *Consider the following configuration of OR gates:*



Solution

When

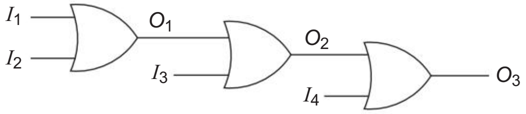
$$I_1 = 1, I_2 = 0 \quad \text{and} \quad I_3 = 1$$

Outputs

$$O_1 = I_1 \cdot I_2 = 1 \cdot 0 = 1$$

$$O_2 = I_3 \cdot O_1 = 1 \cdot 1 = 1$$

Example 1.47 *Consider the following system three OR gates,*



Solution

Assuming

$$I_1 = 0, I_2 = 0, I_3 = 1 \quad \text{and} \quad I_4 = 1$$

Outputs O_1, O_2 and O_3 would be

$$O_1 = I_1 \cdot I_2 = 0 \cdot 0 = 0$$

$$O_2 = I_3 \cdot O_1 = 1 \cdot 0 = 1$$

$$O_3 = I_4 \cdot O_2 = 1 \cdot 1 = 1$$

NOT Gate

The third basic logic gate is NOT gate which produces an output of the opposite state to its input. This logic gate always has only one input signal and one output signal.

Logical Expression

The logical expression for the NOT function is:

$$F = \overline{A}$$

where, F is the output that depends on input, A.

Symbol

The symbol of the NOT gate is shown in Fig. 1.35.

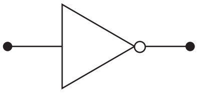


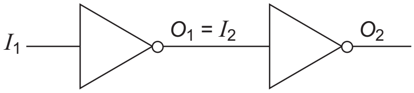
Fig. 1.35 *NOT gate*

Truth Table

Table 1.14 *Truth Table for NOT Gate*

Input A	*Input F*
0	1
1	0

Example 1.48 *Consider two NOT gates configured is shown below:*



Solution

If $I_1 = 1$, then $O_1 = \overline{I_1} = \overline{1} = 0$
and therefore

$$I_2 = O_1 = 0$$

$$O_2 = \overline{I_1} = \overline{0} = 1$$

PROGRAMMING LANGUAGES

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

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

LO 1.12
Discuss various levels of programming languages

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

Three levels of programming languages are available. They are:

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

Machine Language

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

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

First, it is difficult to understand and remember the various combinations of 1s and 0s 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.

Assembly Language

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

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

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

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

```
10110000 01100001
```

Its equivalent assembly language representation is:

```
mov al, 061h
```

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

```
ORG 500          /Origin of program is location 500
LDA SUB          /Load subtrahend to AC
CMA              /Complement AC
INC              /Increment AC
ADD MIN          /Add minuend to AC
```

```
STA DIF         /Store difference
HLT             /Halt computer
MIN, DEC 56     /Minuend
SUB,   DEC -2   /subtrahend
DIF,   HEX 0 /Difference stored here
END   /End of symbolic program
```

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

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

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 the 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 put 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);
}
```

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.

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.

TRANSLATOR PROGRAMS

Assembler

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

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

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

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

Compiler

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

In 1952, Grace Hopper wrote the first compiler for the A-0 programming language. In 1957, John Backus at IBM introduced the first complete compiler. With the increasing complexity of computer architectures and expanding functionality supported by newer programming languages, compilers have become more and more complex. Though early compilers were written in assembly languages, nowadays it has become common practice to implement a compiler in the language it compiles. Compilers are also classified as *single-pass compilers* and *multi-pass compilers*. Though single-pass compilers are generally faster than multi-pass compilers, for sophisticated optimization, multi-pass assemblers are required to generate high-quality code.

Interpreter

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

PROBLEM-SOLVING TECHNIQUES

LO 1.13
Know various problem solving techniques and computer applications

In today's world, a computer is used to solve various types of problems because it takes very less time as compared to a human being. The following steps are performed while solving a problem:

1. Analyse the given problem.
2. Divide the process used to solve the problem in a series of elementary tasks.
3. Formulate the algorithm to solve the problem.
4. Express the algorithm as a precise notation, which is known as a computer program.
5. Feed the computer program in the computer. CPU interprets the given program, processes the data accordingly, and generates the result.
6. Send the generated result to the output unit, which displays it.

Algorithms and flow charts are two important techniques that help users in solving problems or accomplishing tasks using a computer.

Algorithms

An algorithm is a complete, detailed, and precise step-by-step method for solving a problem independently of the software or hardware of the computer. Algorithms are very essential, as they instruct the computer what specific steps it needs to perform to carry out a particular task or to solve a problem. To understand how an algorithm works, let us consider the following example:

Let us assume that XYZ company gives each of its salespersons ₹5000 at the starting of the month for covering various expenses, such as food, lodge, and travel. At the end of the month, the salesperson must submit the receipts of his/her total expenditures to the company. If the amount is less than ₹5000, then the remaining amount must be returned to the company. Now, a simple algorithm can be developed to find out how much money, if any, should be returned to the company.

1. Calculate the total expense receipts of the month.
2. Subtract this amount from ₹5000.
3. If the remainder is greater than 0, return the amount to the company.

Top-down Approach of Algorithms

The top-down approach of an algorithm to solve a given problem is also known as divide and conquer. In this approach, the given problem is divided into two or more sub problems, each of which resembles the original problem. The solution of each sub problem is taken out independently. Finally, the solution of all sub problems is combined to obtain the solution of the main problem. One of the most common examples of the implementation of top-down approach is binary search.

Binary search is a method, which helps search the required data from a given list of data. This method involves comparing the data to be searched and the data present at the middle position of the list. If the data available at the middle position of the list is similar to the data to be searched, the search is considered successful. Otherwise, the list is divided into two parts, left half and right half. The data to be searched is compared with the data present at the mid position. If it is lesser than the data available at the mid position, the left half of the list is searched and if it is greater than the data at the mid position, the right half of the list is searched. This process is repeated until the data to be searched is found or the whole list has been searched. If the data to be searched is found then the search is successful, otherwise the search becomes unsuccessful.

Program Verification

Computer programs are regarded as formal mathematical objects and the properties of these computer programs are subjected to mathematical proofs. Program verification refers to the use of formal, mathematical techniques to debug a program and its specifications. For example, suppose we have coded a program for implementing binary search. Now, we want to verify whether the coded program is correct or not. This can be verified by implementing the program on a given list of data.

Consider an array of 11 elements X[11] = {8,18,26,40,47,69,84,115,126,136,177}. Use the binary search technique to find whether the element '26' is present in this array or not. Now, perform the steps of binary search method to search the required elements. Here, 'Low' represents the location of the first element in the list, 'High' represents the location of the last element in the list, and 'Mid' represents the location of the element available at the middle position in the list. First, search the element '26' in the given array. During the first iteration, the values of Low, High, and Mid are as follows:

- Low = 1
- High = 11
- Mid = 6

The element at the 6th position is '69', which is not the required element. Since, the value of the element at the 6th position is greater than '26', the algorithm searches the left half of the array. During the second iteration, the values of Low, High, and Mid are as follows:

- Low = 1
- High = 5
- Mid = 3

The element at the 3rd position is '26', which is the required element. Thus, the search is successful as the element '26' is present in the array.

Implement the same program twice or thrice on the given list for different elements. If the program gives the correct result, then it is verified that the program is correct.

Efficiency of an Algorithm

Efficiency of an algorithm means how fast it can produce the correct result for the given problem. The efficiency of an algorithm depends upon its time complexity and space complexity. The complexity of an algorithm is a function that provides the running time and space for data, depending on the size provided by us. The two important factors for judging the complexity of an algorithm are as follows:

- Space complexity
- Time complexity

Space complexity of an algorithm refers to the amount of memory required by the algorithm for its execution and generation of the final output.

Time complexity of an algorithm refers to the amount of computer time required by an algorithm for its execution. This time includes both compile time and run time. The compile time of an algorithm does not depend on the instance characteristics of the algorithm. The run time of an algorithm is estimated by determining the number of various operations, such as addition, subtraction, multiplication, division, load, and store, executed by it.

Analysis of Algorithm The analysis of an algorithm determines the amount of resources, such as time and space required by it for its execution. Generally, the algorithms are formulated to work with the inputs of arbitrary length. Algorithm analysis provides theoretical estimates required by an algorithm to solve a problem.

In theoretical notation, the complexity of an algorithm is estimated in asymptotic notations. Asymptotic notations are used to represent the asymptotic run time of an algorithm. These notations are represented in terms of function T(n), where n is the set of natural numbers, 1, 2, 3, 4,…, n. The basic notations used to represent the complexity of an algorithm are:

- **Θ-notation** — It is used to represent the worst case running time of an algorithm.
- **O-notation** — It is used to provide upper boundary constraints over a given function.
- **Ω-notation** — It is used to provide an asymptotic lower bound on the given function.
- **o-notation** — It is used to denote asymptotic loose upper bound.
- **ω-notation** — It is used to denote asymptotic loose lower bound.

Flow Charts

Now to visualize the working of an algorithm, one needs to take the help of a flow chart, which is the pictorial representation of the algorithm depicting the flow of the various steps. If we consider the above example of the expenses of the salesperson, then the flow chart of the algorithm can be represented, as shown in Fig. 1.36.

Example 1.49 *Write an algorithm for finding greatest among three numbers.*

Let x, y and z be the numbers. Now, we can follow the algorithm below to determine the greatest number among the three:

```
1. Read the three numbers.
2. If $x > y$
   a. If $x > z$, then $x$ is the greatest number.
   b. Else, $z$ is the greatest number
3. Else,
   a. If $y > z$, then $y$ is the greatest number.
   b. Else, $z$ is the greatest number.
```

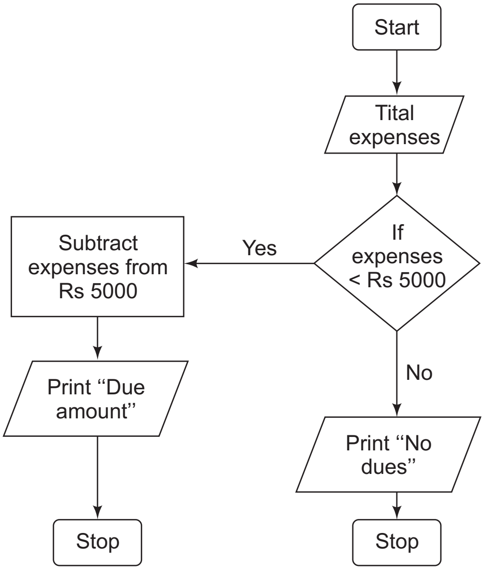


Fig. 1.36 *Flow chart representation of an algorithm*

Example 1.50 *Write the algorithm for converting the degree in Celsius from Fahrenheit*

Let us consider *x* to be the temperature given in Celsius. Now, we need to follow the algorithm below to determine the temperature in Fahrenheit:

1. Read *x*
2. Multiply *x* with 9/5.
3. Add 32 to the multiplied result.
4. Print the final value which is the temperature in Fahrenheit.

Example 1.51 *Write the algorithm for calculating the average of n integers.*

The algorithm for calculating the average of *n* integers is as follows:

1. Read *n* integers.
2. Calculate the sum of the integers.
3. Divide the sum by the total number of integers, that is, *n*.
4. Print the final value which is the average of *n* integers.

Example 1.52 *Write the algorithm for checking whether a number is odd or even.*

The following is the algorithm to determine whether a number is odd or even:

1. Read the given number, say *x*.
2. Divide *x* by 2.
3. If the remainder is 1, then print *x* is odd.
4. Else, print *x* is even.

Example 1.53 *Write the algorithm to determine whether a number is positive, negative or zero.*

1. Read the given number, say *x*.

```
2. If $x \neq 0$,
   a. If $x > 0$, the value of $x$ is positive.
   b. Else, the value of $x$ is negative.
3. Else, the value of $x$ is zero.
```

Example 1.54 *Write an algorithm to find the factorial of a given number.*

The factorial of a non-negative integer *n*, which is denoted by *n*! is the product of all positive integers less than or equal to 1. The algorithm for determining the factorial of a given number is:

```
1. Read the given number, say $x$.
2. Multiply the number $x$ with $x$-1, and store the resultant, say $m$.
3. Repeat the step 2, until the value of $x$ becomes 1.
4. Print the final value, which gives the factorial of the given number.
```

Example 1.55 *Write an algorithm to generate the Fibonacci series.*

The Fibonacci series is defined by the following expression:

$$F(n) = \left\{ \begin{array}{ll} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n\text{-}1) + F(n\text{-}2) & \text{if } n > 1; \end{array} \right\}$$

The above expression states that after two starting values, each number is the sum of two preceding numbers. The algorithm for generating the Fibonacci series is:

```
1. Read the number of terms in the series, say $n$.
2. Set $a$ = 0 and $b$ = 1.
3. Print the value of $a$ and $b$.
4. Set count = 2.
5. While count $\leq n$, $c = a + b$.
6. Print the value of $c$.
7. Set $a = b$ and $b = c$.
8. Increase the value of count by 1.
9. Repeat steps 5 to 8, until count becomes equal to $n$.
```

Example 1.56 *Write an algorithm to find the factors of a given number.*

```
1. Read a number, say num.
2. If num<=0, then go to step 11.
3. Set i=1.
4. Repeat step 5 to 10.
5. If i> num, then go to 10.
6. Else
7. Divide num by i.
8. If the remainder of the division is 0, print i.
9. Increment i by 1 and go to step 5.
10. Endif.
11. Exit.
```

A program to implement this algorithm using *C* language is given in Fig. 1.37

Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int num,i,j;
   clrscr();
   printf("Enter a number to find its factors: ");
   scanf("%d",&num);
   printf("\nFactors of the number %d are: ",num);
   for(i=1;i<=num;i++)
   {
      if(num%i==0)
      printf("%d\t",i);
   }
   getch();
}
```

Output

```
Enter a number to find its factor:12
Factors of the number 12 are:1  2  3  4  6  12
```

Fig. 1.37 *Program to find factors of a given number*

Example 1.57 *Write an algorithm to find the prime factor of a number.*

```
1. Read a number, say n.
2. If n<=1, then go to step 12.
3. Set x=2.
4. Repeat step 5 to 11.
5. If n<=x num, then go to 12
6. Else
7. Divide n by x.
8. If the remainder of the division is 0, print x.
9. Set n=n/x.
10. Increment x by 1 and go to step 5.
11. Endif
12. Exit.
```

A program to implement this algorithm using *C* language is given in Fig. 1.38

Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int n,x;
   clrscr();
```

```
        printf("Enter a number to find its prime factors:");
        scanf("%d",&n);
        if(n<=1)
        {
            printf("Enter a value greater than 1.");
            getch();
            exit(0);
        }
        x=2;
        do
        {
            if(n%x==0)
            {
                printf("%d\t",x);
                n/=x;
            }
            else
                x++;
        }
        while (x<=n);
        getch();
    }
Output
    Enter a number to find its prime factors:
    72

    The prime factors of 72 are:
    2 2 2 3 3
    Enter a number to find its prime factors:
    1
    Enter a value greater than 1.
```

Fig. 1.38 *Program to find prime factors of a given number*

Example 1.58 *Write an algorithm to find the square root of a number.*

```
1. Read a number, say s.
2. If s<0, then go to step 16.
3. Else if s=0
4. Print the value of sq as 0.
5. Else
6. Set n=1.
7. While (!(s>=n*n && s<(n+1)*(n+1))
8. Do increment n by 1
9. End while
```

```
10. d=s-(n*n)
11. P=(double)d/(2*n).
12. a=(double)n+p
13. root=(double)a-((p*p)/(2*a));
14. Print the value of root.
15. Endif
16. Exit.
```

The program in Fig. 1.39 implements above algorithm in C language.

Program

```
#include <stdio.h>
int main()
{
   int s,n;
   double d,p,a,root;
   clrscr();
   printf("Enter a number:");
   scanf("%d",&s);
   if(s<0)
      printf("Enter a positive integer value.");
   else if(s==0)
      printf("Square root of 0 is 0");
   else
   {
      n=1;
      while(!(s>=n*n && s<(n+1)*(n+1)))
      {
         n++;
      }
      d=s-(n*n);
      p=(double)d/(2*n);
      a=(double)n+p;
      root=(double)a-((p*p)/(2*a));
      printf("\nSquare root of %d is %.3f",s,root);
   }
   getch();
}
```

Output

```
Enter a number:16
Square root of the 16 is 4.000.
```

Fig. 1.39 *Program to find square root of a given number*

Example 1.59 *Write an algorithm to find whether the given number is prime or not.*

1. Read a number, say *n* up to which you want to print the prime numbers.
2. Since 1 and 2 are prime numbers, so print them.
3. Check each number up to *n* whether it is prime number or not.
4. Print all the prime numbers up to *n*.

The program in Fig. 1.40 illustrates the implementation of this algorithm.

Program

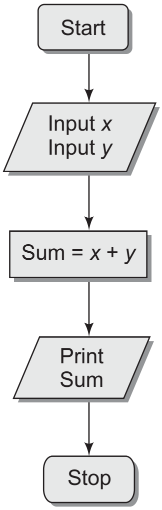
```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
   int n,i,j;
   clrscr();
   printf("Enter a number up to which you want prime numbers:");
   scanf("%d",&n);
   if(n<=1)
   {
      printf("Enter a number greater than 1.");
      getch();
      exit(0);
   }
   printf("Prime numbers between 1 and %d are:",n);
   printf("\n2");

   for(i=3;i<=n;i++)
   {
      for(j=2;j<=sqrt(i);j++)
      {
         if(i%j==0)
            break;
      }
      if(j>sqrt(i))
         printf("\n%d",i);
   }
   getch();
}
```

Output

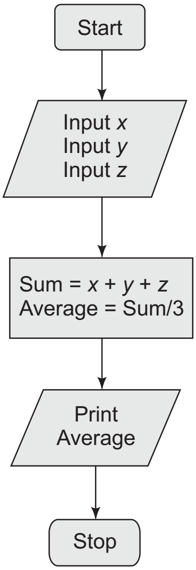
```
Enter a number up to which you want prime numbers:
5
Prime numbers between 1 and 5 are:
2
3
5
```

Fig. 1.40 *Program to find prime numbers up to a given number*

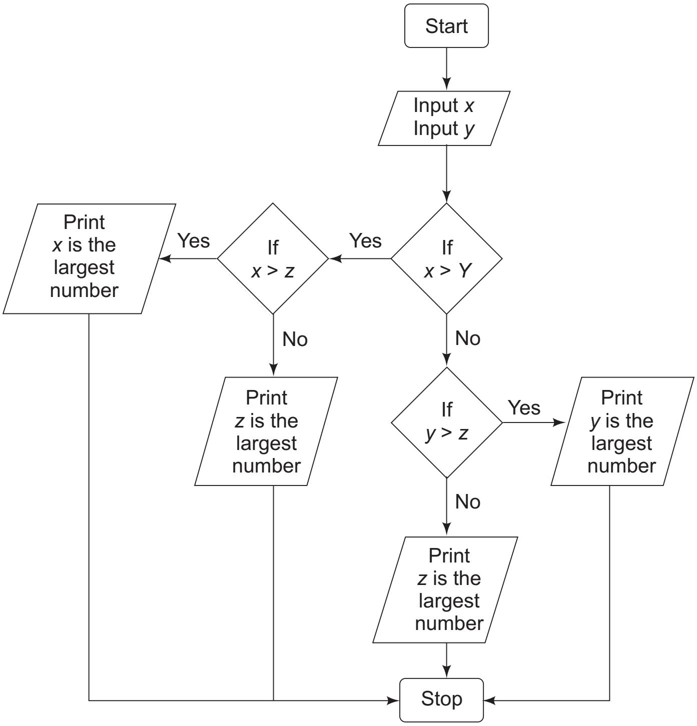
Example 1.60 *Give a flow chart for addition of two numbers.*



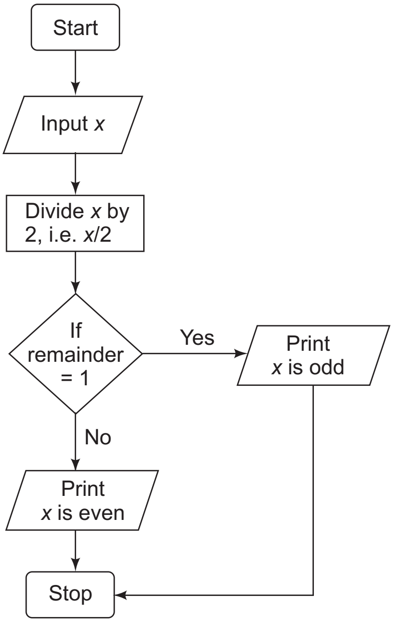
Example 1.61 *Give a flow chart to print he average of three numbers.*



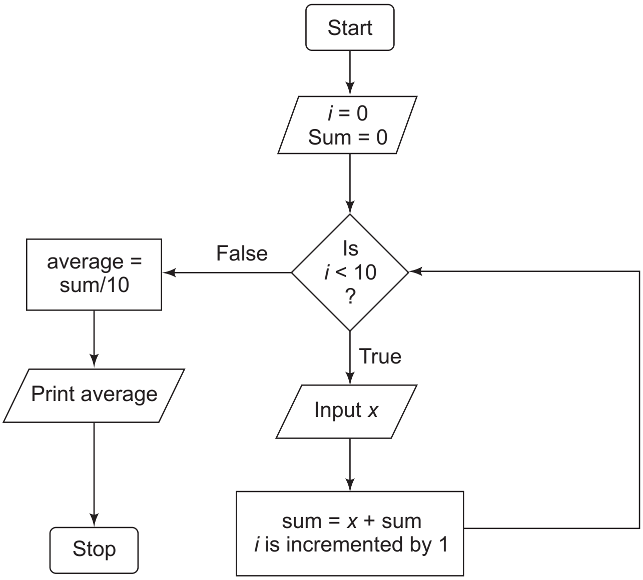
Example 1.62 *Give a flow chart for Example 1.49*


Start
Input x
Input y
If x > Y
Yes
If x > z
Yes
Print x is the largest number
No
Print z is the largest number
No
If y > z
Yes
Print y is the largest number
No
Print z is the largest number
Stop

Example 1.63 *Give a flow chart for Example 1.52*



Example 1.64 *Give a flow chart to determine the average of 10 numbers.*



USING THE COMPUTER

Computers can be used to solve specific problems that may be scientific or commercial in nature. In either case, there are some basic steps involved in using the computers. These are as follows:

- **Problem analysis** Identify the known and unknown parameters and state the constraints under which the problem is to be solved. Select a method of solution.
- **Collecting information** Collect data, information and the documents necessary for solving the problem and also plan the layout of output results.
- **Preparing the computer logic** Identify the sequence of operations to be performed in the process of solving the problem and plan the program logic, preferably using a program flow chart.
- **Writing the computer program** Write the program of instructions for the computer in a suitable language.
- **Testing the program** There are usually errors(bugs) in it. Remove all these errors which may be either in using the language or in the logic.
- **Preparing the data** Prepare input data in the required form.
- **Running the program** This may be done either in batch mode or interactive mode. The computations are performed by the computer and the results are given out.

The use of a particular input/output device depends upon the nature of the problem, type of input data in the form of output required.

LEARNING OUTCOMES

- There are five generations of computer development which have seen tremendous shift in technology, size, and speed. **[LO 1.1]**
- On the basis of the size and capability, computers are categorized into microcomputers, mini computers, super computers and mainframe computers. **[LO 1.2]**

- Input devices help in inputting the data from any outside source into the computer system and output devices are used to pass on the processed data to the end users. **[LO 1.3]**
- Computer systems use two types of memory, namely primary memory and secondary memory. **[LO 1.3]**
- System software is responsible for managing and controlling the hardware resources of a computer system. Application software is specially designed to cater the information processing needs of end users. **[LO 1.4]**
- Operating system is system software installed on a computer system that performs several key tasks, such as process management, memory management, device management, file management, etc. **[LO 1.5]**
- MS Word is used for creating professional as well as personal documents, MS Excel is a spreadsheet application program and MS PowerPoint is application software for creating presentations. **[LO 1.6]**
- A cluster of computers connected together in order to share resources is termed as a computer network. The computers connected in a network generally communicate with the help of network protocols. **[LO 1.7]**
- Computer codes help the computer system to convert the data received in a different number system to the data in the binary form so that it can be stored and processed in an efficient manner. **[LO 1.8]**
- In computer terminology, the number system used to represent data is generally known as positional number system, because the value of the number represented in this system depends upon the position of the digits in the given number. **[LO 1.8]**
- The positional number system can be of four different types, namely, decimal system, binary system, hexadecimal system and octal system. **[LO 1.8]**
- We can easily convert the number represented in one system to its equivalent in another system. The major number system conversions are non-decimal to decimal, decimal to non-decimal and octal to hexadecimal. **[LO 1.9]**
- The basic arithmetic operations performed by the computer system are binary addition, binary multiplication, binary subtraction and binary division. **[LO 1.10]**
- The basic unit of the hardware components of a computer system is the logic gate. **[LO 1.11]**
- There are three Basic Logic gates – AND gate, OR gate and NOT gate. **[LO 1.11]**
- Three levels of programming languages are available – machine languages, assembly languages and procedure-oriented languages. **[LO 1.12]**
- Algorithms and flow charts are two important techniques that help in solving problem using a computer. **[LO 1.13]**

KEY CONCEPTS

- **Transistor**: A semiconductor device that is used to increase the power of the incoming signals by preserving the shape of the original signal. **[LO 1.1]**
- **Microprocessor:** An integrated circuit that contains the entire central processing unit of a computer on a single chip. **[LO 1.1]**
- **Vacuum Tube:** An electron tube from which all or most of the gas has been removed, permitting electrons to move with low interaction with any remaining gas molecules. **[LO 1.1]**
- **LSI:** Large Scale Integration. **[LO 1.1]**
- **VLSI:** Very large-scale integration (VLSI) refers to an IC or technology with many devices on one chip. **[LO 1.1]**
- **ICs**: The circuits that combine various electronic components, such as transistors, resistors, capacitors, etc. onto a single small silicon chip. **[LO 1.1]**
- **Microcomputer**: A small digital computer that is designed to be used by individuals. **[LO 1.2]**

- **Super computer**: The fastest type of computer that can perform complex operations at a very high speed. **[LO 1.2]**
- **Mainframe computer**: A very large computer that is employed by large business organisations for handling major applications, such as financial transaction processing applications and ERP. **[LO 1.2]**
- **Input device**: It is an electromechanical device that is generally used for entering information into a computer system. **[LO 1.3]**
- **Keyboard**: It is a computer input device consisting of keys or buttons arranged in the similar fashion as they are arranged in a typewriter. **[LO 1.3]**
- **Mouse**: It is a pointing device that basically controls the two-dimensional movement of the cursor on the displayed screen. **[LO 1.3]**
- **Scanning devices**: These are the input devices that electronically capture text and images and convert them into computer readable form. **[LO 1.3]**
- **Monitor**: Monitor is the most commonly used output device, which displays the soft copy output of text and graphics to the users. **[LO 1.3]**
- **Printers**: Printers are the output devices that are used to produce a hard copy output of the text or the documents stored in a computer. **[LO 1.3]**
- **Speakers**: Speakers are the output devices used to generate output in an audio format from the computer. **[LO 1.3]**
- **Projectors**: Projectors are the output devices that are used to project big picture of the data stored on some storage device such as CD and DVD on a white screen. **[LO 1.3]**
- **Primary memory:** It refers to the storage locations that are used to hold the programs and data temporarily in a computer system. The primary memory is usually known as memory. **[LO 1.3]**
- **Secondary memory:** It refers to the storage locations that are used to hold the data and programs permanently. The secondary memory of a computer system is popularly known as storage. **[LO 1.3]**
- **Application software**: The programs, which are designed to perform a specific task for the user. **[LO 1.4]**
- **System software:** The programs, which are designed to control the different operations of the computer system. **[LO 1.4]**
- **Operating system:** Operating system is a set of various small system software, which control the execution of various sub processes in a computer system. **[LO 1.5]**
- **MS-DOS**: It is an operating system that makes use of Command Line Interface (CLI) for interacting with the users. **[LO 1.5]**
- **Command**: It can be defined as an instruction provided by a user in order to perform some specific task on the computer system. **[LO 1.5]**
- **MS Word**: It is an application software bundled in MS Office package that allows us to create edit, save and print personal as well as professional documents in a very simple and efficient manner. **[LO 1.6]**
- **MS Excel**: MS Excel is a spreadsheet application program that enables the users to create the spreadsheets. **[LO 1.6]**
- **MS PowerPoint**: MS PowerPoint is an application software included in the MS Office package that allows us to create presentations. **[LO 1.6]**
- **Data communication**: It is the process of transmission of data from the source computer to the destination computer. **[LO 1.7]**
- **Network topology**: The network topology is the physical arrangement of the computers connected with each other in a network such as ring, star, bus, hierarchical and hybrid. **[LO 1.7]**
- **Network protocol**: The network protocol is the standard according to which different computers over the network communicate with each other. **[LO 1.7]**
- **Computer codes**: The computer codes are the codes that help in converting the data entered by the users into the binary form. **[LO 1.8]**

- **Positional number system**: The positional number system is a system in which numbers are represented using certain symbols called digits and the values of these numbers is determined by taking the position of digits into consideration. **[LO 1.8]**
- **Decimal system**: The decimal system is a positional number system that uses base 10 to represent different values. **[LO 1.8]**
- **Binary system**: The binary system is a positional number system that uses base 2 to represent different values. **[LO 1.8]**
- **Hexadecimal system**: The hexadecimal system is a positional number system that uses base 16 to represent different values. **[LO 1.8]**
- **Octal system**: The octal system is a positional number system that uses base 8 to represent different values. **[LO 1.8]**
- **Number system conversions:** The different type of number system conversions can be divided into three major categories: non-decimal to decimal, decimal to non-decimal and octal to hexadecimal. **[LO 1.9]**
- **ALU:** ALU is an important component of CPU that is used to perform various arithmetic and logical operations in the computer system. **[LO 1.10]**
- **Integer arithmetic:** Integer arithmetic refers to various arithmetic operations involving integer operands only. **[LO 1.10]**
- **Floating-point arithmetic:** Floating-point arithmetic refers to various arithmetic operations involving floating-point operands only. **[LO 1.10]**
- **Unsigned binary number:** Unsigned binary number is the number with a magnitude of either zero or greater than zero. **[LO 1.10]**
- **Basic logic gates:** Basic logic gates are the building blocks of digital circuits that perform logical operations such as AND, OR and NOT, on the binary inputs. **[LO 1.11]**
- **Machine Language**: The computer instructions written using binary codes 1 and 0 are machine code or machine language. **[LO 1.12]**
- **Assembly Language**: In an assembly language, the 0s and 1s of machine language are replaced with abbreviations or mnemonic code. **[LO 1.12]**
- **High Level Language**: High level language code is executed by translating it into corresponding machine language code with the help of a compiler or interpreter. **[LO 1.12]**
- **Algorithms**: An algorithm is a complete, detailed and precise step-by-step method for solving a problem independently of the software or hardware of the computer. **[LO 1.13]**
- **Flow charts**: A flow chart is the pictorial representation of the algorithm depicting the flow of the various steps. **[LO 1.13]**

REVIEW QUESTIONS

1. Fill in the blanks in the following statements.

   1.1 A ___________ is an electronic machine that takes input from the user and stores and processes the given input to generate the output in the form of useful information to the user. **[LO 1.1 E]**

   1.2 The raw details that need to be processed to generate some useful information is known as ___________. **[LO 1.1 M]**

   1.3 The set of instructions that can be executed by the computer is known as ___________. **[LO 1.1 M]**

E for Easy, **M** for Medium and **H** for High

1.4 ___________ is the processor of the computer that is responsible for controlling and executing the various instructions. **[LO 1.1 M]**

1.5 ___________ is a screen, which displays the information in visual form, after receiving the video signals from the computer. **[LO 1.1 E]**

1.6 ___________ computers were also known as vacuum tubes or thermionic valves based machines. **[LO 1.1 M]**

1.7 A ___________ is a semiconductor device that is used to increase the power of the incoming signals by preserving the shape of the original signal. **[LO 1.1 H]**

1.8 ___________ is a low-level language that allows the programmer to use simple English words, called mnemonics, to represent different instructions in a program. **[LO 1.1 M]**

1.9 The main characteristic feature of third generation computers was the use of ___________. **[LO 1.1 E]**

1.10 The invention of ___________ and ___________ technology led to the development of the fourth generation computers. **[LO 1.1 M]**

1.11 The fifth generation computers are based on the ___________ technology that allows almost ten million electronic components to be fabricated on one small chip. **[LO 1.1 M]**

1.12 ___________, also known as digital information processing system, is a type of computer that stores and processes data in digital form. **[LO 1.2 M]**

1.13 A ___________ is the fastest type of computer that can perform complex operations at a very high speed. **[LO 1.2 M]**

1.14 The term ___________ refers to the programs and instructions that help the computer in carrying out their processing. **[LO 1.2 E]**

1.15 The programs, which are designed to perform a specific task for the user, are known as ___________. **[LO 1.4 M]**

1.16 The programs, which are designed to control the different operations of the computer, are known as ___________. **[LO 1.4 M]**

1.17 An input device generally acts as an interface between ___________ and __________. **[LO 1.3 E]**

1.18 The arrow keys used for controlling the movement of ___________ are known as ___________ keys. **[LO 1.3 M]**

1.19 Keyboards are also classified as ___________ and ___________ keyboards, based on additional keys present on them. **[LO 1.3 E]**

1.20 ____________ devices are used for changing the position of the cursor on the screen. **[LO 1.3 E]**

1.21 A mechanical mouse basically consists of _____________, ___________ and __________ buttons. **[LO 1.3 M]**

1.22 An optical mouse consists of ________________, ___________________ and ___________ for moving the position of the pointer on the screen. **[LO 1.3 H]**

1.23 Hand-held scanners are also called ______________. **[LO 1.3 M]**

1.24 The methods used for recognising the voice of the users are ___________ and _____________. **[LO 1.3 M]**

1.25 Computer software is classified into two categories, namely, ______________ and ______________. **[LO 1.4 E]**

1.26 System software consists of two groups of programs: ______________ and ______________. **[LO 1.4 M]**

1.27 ______________ is responsible for managing the allocation of devices and resources to the various processes. **[LO 1.4 M]**

1.28 Application software includes two types programs: ______________ and ______________. **[LO 1.4 M]**

1.29 ____________ is a system software that allows the users to interact with the hardware and other resources of a computer system. **[LO 1.5 E]**

1.30 In ___________ operating system, jobs are grouped into groups called batches and assigned to the computer system with the help of a card reader. **[LO 1.5 M]**

1.31 In ____________ operating system, multiple users can make use of computer system's resources simultaneously. **[LO 1.5 M]**

1.32 UI facilitates communication between a _________ and its ______ by acting as an intermediary between them. **[LO 1.5 H]**

1.33 ___________ is the central part of the UNIX operating system that manages and controls the communication between the various hardware and software components. **[LO 1.5 H]**

1.34 MS-DOS is an operating system that makes use of ___________ interface. **[LO 1.5 H]**

1.35 _________ commands are stored in the command interpreter of MS-DOS. **[LO 1.5 H]**

1.36 RD, TYPE and DEL are __________ commands. **[LO 1.5 H]**

1.37 _________ and _________ are external commands. **[LO 1.5 H]**

1.38 _____________ is an application software included in MS Office for working with documents. **[LO 1.6 M]**

1.39 MS Word can be accessed either using ___________ or _____________. **[LO 1.6 M]**

1.40 MS Word uses a ___________ interface to interact with the users. **[LO 1.6 M]**

1.41 The horizontal bar at the top of the MS Word window is called ___________. **[LO 1.6 E]**

1.42 The blinking bar in MS Word that indicates the position of the next key stroke or the character to be inserted is called ___________. **[LO 1.6 H]**

1.43 ________________ is a spreadsheet application program that is widely used in business applications. **[LO 1.6 E]**

1.44 The horizontal sequence of data stored in a spreadsheet is known as ______________. **[LO 1.6 E]**

1.45 The vertical sequence of data stored in a spreadsheet is known as _______________. **[LO 1.6 E]**

1.46 _____________ is an application software included in MS Office package for creating presentations. **[LO 1.6 M]**

1.47 The presentations in the MS PowerPoint are usually saved with the ___________ extension. **[LO 1.6 H]**

1.48 When computers are connected together in order to share resources, they are said to be in a __________. **[LO 1.7 M]**

1.49 ____________ is used for connecting the computers within a few kilometres of area. **[LO 1.7 M]**

1.50 ____________ is used for connecting the computers in a large geographical area. **[LO 1.7 M]**

1.51 The size of the MAN generally lies between that of LAN and WAN, typically covering a distance of ____________ to _______________. **[LO 1.7 M]**

1.52 Hierarchical topology is also known as _______________. **[LO 1.7 M]**

1.53 ____________ is the common point where all the nodes of the network are connected in the bus topology. **[LO 1.7 M]**

1.54 ____________ is used for connecting the nodes in the star topology. **[LO 1.7 H]**

1.55 The combination of multiple topologies connected in a network is known as ______________. **[LO 1.7 M]**

1.56 ______________ is the set of rules and regulations based on which computers in a network communicate. **[LO 1.7 M]**

1.57 _____________ is one of the tasks that can be performed using network protocol. **[LO 1.7 M]**

1.58 _____________ is used for transferring files from one computer to another over the network. **[LO 1.7 M]**

1.59 The most common system used by computer systems is _________. **[LO 1.8 E]**

1.60 The weight of any digit in the number system generally depends upon its _________ in the given number. **[LO 1.8 M]**

1.61 The binary system represents each type of data in the form of _________ and _________. **[LO 1.8 E]**

1.62 The digits in binary system are referred as _________. **[LO 1.8 E]**

1.63 The base of any number system depends upon the number of _________ in the system. **[LO 1.8 M]**

1.64 Computer designers and professionals generally deal with _________ number system. **[LO 1.8 M]**

1.65 The octal system is also known as _________ system. **[LO 1.8 M]**

1.66 The octal number 5624 is equivalent to _________ in decimal system. **[LO 1.9 H]**

1.67 The binary number 1001010 represents a decimal value of _________. **[LO 1.9 H]**

1.68 The hexadecimal system consists of _________ symbols. **[LO 1.9 E]**

1.69 Human beings usually supply data to the computer system in the _________ form. **[LO 1.9 M]**

1.70 Computer codes help computer systems convert the decimal data into _________ data. **[LO 1.9 M]**

1.71 The hexadecimal number B45A is equivalent to _________ in decimal system. **[LO 1.9 M]**

1.72 The hexadecimal representation of the octal number 2564 is _________. **[LO 1.9 H]**

1.73 The arithmetic operations are usually performed in the computer system by _________ and _________ unit of the CPU. **[LO 1.10 E]**

1.74 The computer arithmetic is also referred to as the _________ arithmetic. **[LO 1.10 M]**

1.75 The binary multiplication can be considered as the _________ process of binary _________. **[LO 1.10 M]**

1.76 Unsigned binary number is a number with a magnitude of either _________ or _________. **[LO 1.10 M]**

1.77 The different arithmetic laws hold true for _________ as well _________ operations. **[LO 1.10 M]**

1.78 Logic gates are the building blocks of digital circuits that perform various ______________ on the binary input. **[LO 1.11 E]**

1.79 The values of the input and the corresponding output of the logic gates can be represented using a table called ______________ . **[LO 1.11 E]**

1.80 The output of the ______________ gate is true if any one of the inputs is true. **[LO 1.11 M]**

1.81 The ______________ inverts the value of the input for producing the output. **[LO 1.11 M]**

1.82 The output of ______________ gate is true if both the inputs are same. **[LO 1.11 M]**

1.83 The ___________ is a translation program that converts each high-level program statement into the corresponding machine code. **[LO 1.12 M]**

1.84 An __________is a complete, detailed and precise step-by-step method for solving a problem independently of the software or hardware of the computer. **[LO 1.13 E]**

1.85 A flow chart is the __________ of the algorithm depicting the flow of the various steps. **[LO 1.13 M]**

2. Multiple Choice Questions

1.1 Which component of the computer is known as the brain of computer? **[LO 1.1 M]**

A. Monitor B. CPU
C. Memory D. None of the above

1.2 Which of the following is an input device? **[LO 1.1 M]**
A. Printer B. Monitor
C. Mouse D. None of the above

1.3 Which of the following is a characteristic of the modern digital computer? **[LO 1.1 M]**
A. High speed B. Large storage capacity
C. Greater accuracy D. All of the above

1.4 Who is known as the father of modern digital computers? **[LO 1.1 E]**
A. Gottfried Wilhem Von Leibriz B. Charles Babbage
C. Alan Mathison D. John Mauchly

1.5 What are the different number of computer generations? **[LO 1.1 E]**
A. Four B. Five
C. Six D. Seven

1.6 Which technology was used in the first generation computers? **[LO 1.1 M]**
A. Transistors B. Vacuum tubes
C. ICs D. None of the above

1.7 Which technology was used in the second generation computers? **[LO 1.1 M]**
A. Transistors B. Vacuum tubes
C. Microprocessors D. ICs

1.8 Which technology was used in the third generation computers? **[LO 1.1 M]**
A. Transistors B. Vacuum tubes
C. ICs D. All of the above

1.9 Which technology was used in the fourth generation computers? **[LO 1.1 M]**
A. Microprocessors B. Vacuum tubes
C. ICs D. Transistors

1.10 Which semiconductor device is used to increase the power of the incoming signals by preserving the shape of the original signal? **[LO 1.1 H]**
A. Sand table B. Transistor
C. Vacuum tubes D. None of the above

1.11 In which generation of computers, assembly language was introduced? **[LO 1.1 H]**
A. First B. Second
C. Third D. Fourth

1.12 Which generation uses the ULSI technology? **[LO 1.1 E]**
A. Second B. Third
C. Fourth D. Fifth

1.13 On what basis computers can be classified? **[LO 1.1 E]**
A. Operating principles B. Applications
C. Size and capability D. All of the above

1.14 What is the main function of an input device in a computer? **[LO 1.3 M]**
A. Receiving data from a computer B. Providing data to a computer
C. Storing data for processing D. Processing the data

1.15 Which of the following devices is not an input device? **[LO 1.3 E]**
A. Scanner B. Keyboard
C. Disk D. Joystick

1.16 Which one of the following is a modifier key? **[LO 1.3 H]**
A. Tab B. ALT
C. Insert D. Pause

1.17 Which of the following belongs to the category of special purpose keys? **[LO 1.3 M]**
A. Tab
B. SHIFT
C. ALT
D. CTRL

1.18 Which of the following statements is not true for a mouse? **[LO 1.3 E]**
A. It controls the two-dimensional movement of the cursor on the displayed screen.
B. It is usually of two different types: mechanical mouse and optical mouse.
C. It can be used as an alternate to keyboard for all purposes.
D. It is an input device.

1.19 What is the other name of a hand-held scanner? **[LO 1.3 M]**
A. Drum scanner
B. Slide scanner
C. Half page scanner
D. Full page scanner

1.20 Which of the following devices is not an optical recognition device? **[LO 1.3 E]**
A. MICR
B. OMR
C. OCR
D. Microphone

1.21 What does MICR stand for? **[LO 1.3 H]**
A. Magnetic Ink Character Recognition
B. Magnetic Input Column Reader
C. Magnetic Ink Column Recognition
D. Magnetic Ink Character Reader

1.22 Which of the following devices are used for recognising the characters in the supermarkets? **[LO 1.3 M]**
A. OCR device
B. OMR device
C. MICR device
D. Bar code reader

1.23 Which of the following is not an output device? **[LO 1.3 E]**
A. Scanner
B. Plotter
C. Printer
D. Speaker

1.24 Which of the following monitors are commonly used with desktop computers? **[LO 1.3 E]**
A. CBT monitors
B. CRT monitors
C. CPT monitors
D. None of the above

1.25 Which of the following are the properties of a printer? **[LO 1.3 M]**
A. Resolution
B. Speed
C. Pages per minute
D. All of the above

1.26 Which of the following is a hard copy output device? **[LO 1.3 E]**
A. Printer
B. Speaker
C. Display monitor
D. Projector

1.27 Which of the following is an impact printer? **[LO 1.3 M]**
A. Dot matrix printer
B. Ink-jet printer
C. Laser printer
D. All of the above

1.28 Which of the following is a non-impact printer? **[LO 1.3 M]**
A. Daisy wheel printer
B. Dot matrix printer
C. Laser printer
D. All of the above

1.29 Which of the following is one of the components of a CRT? **[LO 1.3 M]**
A. Toner
B. Liquid crystals
C. Electromagnetic coils
D. None of the above

1.30 Which of the following are the components of a projector? **[LO 1.3 H]**
A. Optic system
B. Displays
C. Electron beam
D. Both A and B

1.31 Which of the following are portable projectors? **[LO 1.3 M]**
A. Conference room projectors
B. Fixed installation projectors
C. Ultralight projectors
D. All of the above

1.32 Which of the following devices are included in a terminal? **[LO 1.3 M]**

A. Monitor and printer B. Printer and keyboard

C. Keyboard and monitor D. All of the above

1.33 Which of the following is a type of terminal? **[LO 1.3 E]**

A. Intelligent terminal B. Dumb terminal

C. Both A and B D. All of the above

1.34 Which of the following can be considered as both an input and an output device? **[LO 1.3 E]**

A. Printer B. Projector

C. Terminal D. Plotter

1.35 Which of the following display device uses an electron given as one of the components for generating the output? **[LO 1.3 M]**

A. CRT monitor B. TFT monitor

C. LCD monitor D. None of the above

1.36 Which of the following is not a system software? **[LO 1.4 M]**

A. Linkers B. Device drivers

C. Operating system D. Word processor

1.37 Which of the following software helps the users to detect the errors while executing a program? **[LO 1.4 H]**

A. Language Translator B. Debugger

C. Loader D. Linker

1.38 A software, which links different elements of an object code with the library files, is known as: **[LO 1.4 H]**

A. Editor B. Linker

C. Loader D. Debugger

1.39 Which of the following options is not a utility system? **[LO 1.4 H]**

A. Virus scanner B. System profiler

C. Disk defragmenter D. Debugger

1.40 Which of the following is a system tool provided by Windows operating system for making necessary changes in the registry? **[LO 1.4 E]**

A. System profiler B. Disk Defragmenter

C. Registry Editor D. Registry Manager

1.41 Which of the following is not an example of unique application program? **[LO 1.4 M]**

A. Inventory Management System B. Pay-roll system

C. Income tax calculator D. Database Management System

1.42 Which of the following activities are performed by a user while solving a problem using a computer? **[LO 1.4 E]**

A. Identifying parameters and constraints B. Identifying logical structure

C. Debugging the program D. All of the above

1.43 Which of the following program is essential for the functioning of a computer system? **[LO 1.5 M]**

A. MS Word B. Operating system

C. MS Excel D. System software

1.44 Which of the following operating systems makes use of CLI? **[LO 1.5 M]**

A. MS-DOS B. Windows 2000

C. Windows Server 2003 D. None of the above

1.45 Which of the following operating systems makes use of GUI? **[LO 1.5 M]**

A. Windows 2000 B. Windows Server 2003

C. Windows Vista D. All of the above

1.46 Which of the following operating systems makes use of both command line interface and GUI? **[LO 1.5 M]**

A. Windows 2000 B. Linux
C. Windows Vista D. None of the above

1.47 Which one of the following types of the operating systems allows multiple users to work simultaneously? **[LO 1.5 H]**

A. Multi-tasking operating system B. Multi-user operating system
C. Multiprocessor operating system D. None of the above

1.48 Which of the following type of UI allows a user to enter commands at command line? **[LO 1.5 H]**

A. GUI B. CLI
C. Both GUI and CLI D. Neither GUI nor CLI

1.49 Which of the following is a part of MS-DOS? **[LO 1.5 H]**

A. DOS.SYS B. CONFIGURATION.SYS
C. EXEC.BAT D. COMMAND.COM

1.50 Which of the following is the core component of UNIX? **[LO 1.5 M]**

A. Command shell B. Kernel
C. Directories and programs D. None of the above

1.51 Which of the following is a feature of MS-DOS operating system? **[LO 1.6 M]**

A. 16-bit B. Single-user
C. Single tasking D. All of the above

1.52 Which of the following commands are used in MS-DOS operating system? **[LO 1.6 M]**

A. Internal commands B. External commands
C. Batch commands D. All of the above

1.53 Which of the following commands is used for viewing the contents of a file in MS-DOS operating system? **[LO 1.6 H]**

A. DIR B. TYPE
C. MD D. CD

1.54 Which of the following commands is used to print a message on the command prompt? **[LO 1.6 H]**

A. %DIGIT B. %VARIABLE%
C. ECHO D. REM

1.55 Which of the following makes use of CLI? **[LO 1.6 M]**

A. MS Excel B. MS PowerPoint
C. MS-DOS D. MS Access

1.56 Which one of the following is typed in the Run dialog box to access MS Word? **[LO 1.6 E]**

A. winword B. word
C. msword D. wordprogram

1.57 Which of the following is a word processing program? **[LO 1.6 E]**

A. MS Excel B. MS-DOS
C. MS Word D. MS PowerPoint

1.58 Which of the following is a spreadsheet application program? **[LO 1.6 E]**

A. MS Access B. MS Word
C. MS Excel D. MS-DOS

1.59 MS Word is basically used for _______. **[LO 1.6 E]**

A. Analysing the data B. Preparing the various documents
C. Preparing the slides D. None of the above

1.60 What text should be typed in the Run dialog box for accessing MS Excel? **[LO 1.6 M]**

A. msexcel B. excel
C. xcel D. msspreadsheet

1.61 What text should be typed in the Run dialog box for accessing MS PowerPoint? **[LO 1.6 M]**

A. powerpoint B. powerpnt
C. mspowerpnt D. ppt

1.62 What is the name of the task pane used for designing slides in MS PowerPoint? **[LO 1.6 M]**

A. Slide Design B. Slide Layout
C. Design Slide D. None of the above

1.63 What is the intersection of row and column called in MS Excel? **[LO 1.6 E]**

A. Cell B. Worksheet
C. Workbook D. None of the above

1.64 What is correct expansion of MS DOS? **[LO 1.6 H]**

A. Microsoft Data Operating system B. Microsoft Disk Operating system
C. Microsoft Digital Operating system D. None of the above

1.65 What is the combination of worksheets in MS Excel called? **[LO 1.6 M]**

A. Workbook B. Spread sheet
C. Excel sheet D. None of the above

1.66 Which one of the following uses light pulses for carrying information? **[LO 1.7 M]**

A. Satellite B. Microwave
C. Optical fibre D. Coaxial cable

1.67 Which of the following network is used for connecting the computers in a small geographical area? **[LO 1.7 M]**

A. MAN B. WAN
C. LAN D. Internet

1.68 What is the full form of TCP? **[LO 1.7 M]**

A. Transfer Control Protocol B. Transmission Control Protocol
C. Transmit Control Protocol D. Transfer Communication Protocol

1.69 Which one of the following Internet services provides one to one communication? **[LO 1.7 M]**

A. Online chat B. Online messaging
C. E-mail D. Usenet

1.70 A network that is restricted to use by a single organisation is referred to as: **[LO 1.7 M]**

A. LAN B. WAN
C. Internet D. Intranet

1.71 Which type network cannot work under heavy load? **[LO 1.7 M]**

A. MAN B. LAN
C. PPN D. VAN

1.72 Which topology is arranged in the form of a tree structure? **[LO 1.7 E]**

A. Hybrid topology B. Bus topology
C. Star topology D. Hierarchical topology

1.73 Which one of the following topologies is not easy to reconstruct when a fault occurs? **[LO 1.7 H]**

A. Star topology B. Bus topology
C. Ring topology D. Hybrid topology

1.74 Which one of the following topologies allow easy error detection and correction? **[LO 1.7 H]**

A. Linear bus topology B. Hybrid topology
C. Ring topology D. Star topology

1.75 Which device is used for connecting the computers in a star topology? **[LO 1.7 H]**

A. Router B. Bridge
C. Hub D. Repeater

1.76 Which topology is the combination of multiple topologies? **[LO 1.7 M]**

A. Star topology B. Bus topology
C. Hybrid topology D. Mesh topology

1.77 In which topology data is transferred in a circular pattern? **[LO 1.7 E]**

A. Star topology B. Ring topology
C. Bus topology D. Hybrid topology

1.78 Which of the following topologies is the most complex but efficient? **[LO 1.7 M]**

A. Star topology B. Bus topology
C. Ring topology D. Hybrid topology

1.79 What is the technique used for routing the packets to the destination according to their addresses? **[LO 1.7 H]**

A. Circuit switching B. Packet switching
C. Routing D. None of the above

1.80 Which one of the following is not a network protocol? **[LO 1.7 H]**

A. FTP B. HTTP
C. SMTP D. NMP

1.81 A set of rules that are used for communication between two networks is referred to as: **[LO 1.7 E]**

A. Network software B. Network media
C. Network protocol D. Network operating system

1.82 Which of the following is not a positional number system? **[LO 1.8 E]**

A. Octal system B. Decimal system
C. Binary system D. Roman number system

1.83 Human beings usually employ the following number system for their routine computations: **[LO 1.8 E]**

A. Decimal system B. Octal system
C. Binary system D. Hexadecimal system

1.84 The number system with base 2 is known as: **[LO 1.8 E]**

A. Decimal system B. Binary system
C. Octal system D. Hexadecimal system

1.85 The 4-bit binary equivalent of the decimal number 6 is: **[LO 1.9 M]**

A. 0111 B. 1000
C. 0010 D. 0110

1.86 The octal representation of 15 is: **[LO 1.9 M]**

A. 17 B. 16
C. 15 D. 14

1.87 Which of the following form of data is processed more efficiently by the computer system? **[LO 1.9 M]**

A. Binary data B. Octal data
C. Hexadecimal data D. Decimal data
E. Hexadecimal point F. None of the above

1.88 The system implemented by the computer systems to convert the decimal numbers into equivalent binary numbers is known as: **[LO 1.9 M]**

A. BCD system B. Octal system
C. Weighted system D. Gray code system

1.89 Which of the following codes is a type of digital code? **[LO 1.9 M]**

A. ASCII code B. Packed code
C. 8421 code D. None of the above

1.90 Which of the following is not a valid computer number system conversion? **[LO 1.9 E]**

A. Non-decimal to decimal B. Decimal to non-decimal
C. Octal to hexadecimal D. Roman to decimal

1.91 The hexadecimal equivalent of the octal number 4263 is: **[LO 1.9 H]**
A. 8B3 B. A42
C. 923 D. BA31

1.92 Which of the following is not an appropriate operand for arithmetic operations? **[LO 1.10 H]**
A. Integers B. Strings
C. Real D. None of the above

1.93 Which of the following is not a valid binary addition rule? **[LO 1.10 H]**
A. 0 + 0 = 0 B. 1 + 0 = 1
C. 1 + 1 = 0 with a carry 1 D. 1 + 1 = 0 with no carry

1.94 What is the result of the binary addition performed on the numbers 1001 and 0101? **[LO 1.10 M]**
A. 0010 B. 1110
C. 1010 D. 1111

1.95 The binary multiplication can be considered as the repetitive process of: **[LO 1.10 E]**
A. Binary addition B. Binary subtraction
C. Binary division D. Binary multiplication

1.96 Which of the following is not a valid binary multiplication rule? **[LO 1.10 H]**
A. 0 × 0 = 1 B. 0 × 1 = 0
C. 1 × 1 = 1 D. 1 × 0 = 0

1.97 What is the result of the binary multiplication performed on the numbers 12 and 10? **[LO 1.10 M]**
A. 101011 B. 0111101
C. 1111000 D. 1010000

1.98 Which of the following is not a valid binary subtraction rule? **[LO 1.10 H]**
A. 0 – 0 = 0 B. 1 – 0 = 1 with no borrow
C. 1 – 1 = 0 D. 0 – 1 = 1 with no borrow

1.99 What is the result of binary subtraction performed on the numbers 1001 and 0101? **[LO 1.10 M]**
A. 0001 B. 0101
C. 1000 D. 0011

1.100 Binary division is closely related with the arithmetic operation: **[LO 1.10 E]**
A. Binary addition B. Binary subtraction
C. Binary multiplication D. Binary division
E. Whether the number is zero F. None of the above

1.101 Which of the following is not an arithmetic law? **[LO 1.10 H]**
A. Identity law B. Distributive law
C. Commutative law D. Law of negation

1.102 Which of the following components is actually responsible for executing an instruction? **[LO 1.11 E]**
A. Software B. Hardware
C. Flip-flops D. Counter

1.103 Which of the following are the building blocks of digital circuit? **[LO 1.11 E]**
A. Flip-flops B. Logic gates
C. Register D. None of the above

1.104 Which of the following types of operations can be performed by logic gates? **[LO 1.11 E]**
A. Assignment operation B. Arithmetical operation
C. Logical operation D. Shift operation

1.105 Which of the following digital circuits is used to add binary numbers? **[LO 1.11 M]**
A. Register B. Logic gates
C. Adder D. All of the above

1.106 Which of the following logic gates is also known as inverter? **[LO 1.11 M]**
A. AND B. OR
C. NAND D. NOT

DISCUSSION QUESTIONS

1.1 What are the different components of a computer? Explain, each of them. **[LO 1.1 M]**

1.2 Discuss briefly the various generations of a computer. **[LO 1.1 M]**

1.3 Describe the various types of computers on the basis of size and capability. **[LO 1.2 M]**

1.4 Draw the block diagram of a microcomputer. **[LO 1.2 H]**

1.5 What is meant by an input device? What is the importance of an input device in a computer system? **[LO 1.3 M]**

1.6 List different categories of input devices. **[LO 1.3 E]**

1.7 Explain all the categories of keys found on a typical keyboard with the help of a diagram. **[LO 1.3 H]**

1.8 Explain the basic functioning of mechanical and optical mouses with the help of sketches. **[LO 1.3 M]**

1.9 What are scanning devices? Explain the basic characteristics of these devices. **[LO 1.3 M]**

1.10 What does voice recognition system mean? **[LO 1.3 H]**

1.11 Explain the different methods used for identifying the voice of the user in the voice recognition system. **[LO 1.3 H]**

1.12 What is an output device? Why is it a vital part of computer hardware? **[LO 1.3 M]**

1.13 Name some of the output devices, which are commonly used with the computer system. **[LO 1.3 E]**

1.14 Define a display monitor. **[LO 1.3 M]**

1.15 Name the different types of monitors available in the market. **[LO 1.3 M]**

1.16 Explain the use of a printer in a computer system. **[LO 1.3 E]**

1.17 What are the advantages and disadvantages of a CRT monitor? **[LO 1.3 M]**

1.18 Which is a better monitor—a CRT or a TFT? State the reasons as well. **[LO 1.3 H]**

1.19 What is a voice response system? List the different types of voice response systems that are used today. **[LO 1.3 H]**

1.20 What is a projector? Why is it needed? **[LO 1.3 E]**

1.21 Explain the different types of computer software. **[LO 1.4 M]**

1.22 What do you understand by the term system software? **[LO 1.4 M]**

1.23 Explain the major functions of an operating system. **[LO 1.4 H]**

1.24 Explain the application of system development programs. **[LO 1.4 M]**

1.25 What does utility program mean? **[LO 1.4 H]**

1.26 What is an operating system? Explain briefly with the help of examples. **[LO 1.5 M]**

1.27 Briefly explain the various functions of an operating system. **[LO 1.5 M]**

1.28 Explain the core components of UNIX operating system. **[LO 1.5 H]**

1.29 Briefly explain why Windows operating system is one of the most popular operating systems. **[LO 1.5 M]**

1.30 Explain the features of MS-DOS operating system. **[LO 1.6 M]**

1.31 Differentiate between internal and external commands of MS-DOS. **[LO 1.6 M]**

1.32 What do you mean by command interpreter? **[LO 1.6 M]**

1.33 Write a short note on the following commands: **[LO 1.6 E]**

A. DIR
B. COPY
C. MD
D. TREE
E. COMP.

1.34 What is the basic use of MS Word? Explain with the help of an example. **[LO 1.6 E]**

1.35 What are the different methods of accessing MS Word? **[LO 1.6 M]**

1.36 What are the basic operations performed on a word document? Explain all of them in detail. **[LO 1.6 E]**

1.37 What do you mean by MS-Excel? Explain the different ways of starting MS-Excel from our computer system? **[LO 1.6 M]**

1.38 What are the different operations possible on a worksheet in MS-Excel? **[LO 1.6 E]**

1.39 What are the different methods of accessing MS PowerPoint? **[LO 1.6 M]**

1.40 What is the difference between creating and designing a new presentation in MS PowerPoint? **[LO 1.6 H]**

1.41 How can a new slide be added to a presentation in MS PowerPoint? **[LO 1.6 E]**

1.42 What is a computer network? **[LO 1.7 M]**

1.43 Describe different types of computer networks with the help of illustrations. **[LO 1.7 M]**

1.44 What is the difference between LAN and WAN? **[LO 1.7 H]**

1.45 What is network topology? **[LO 1.7 E]**

1.46 What are the different types of network topologies? Explain any two network topologies through suitable illustrations. **[LO 1.7 H]**

1.47 How network protocol helps in the communication of messages over the network? **[LO 1.7 H]**

1.48 What is the difference between ring topology and bus topology? **[LO 1.7 M]**

1.49 Differentiate among ring, star, bus and hybrid topology with the help of diagrams. **[LO 1.7 M]**

1.50 What do you understand by positional number system and why is it called a positional system? **[LO 1.8 E]**

1.51 What are the different types of positional number systems? Which of the positional systems is mostly used by the computer systems? **[LO 1.8 M]**

1.52 Explain the different technical terms associated with the binary system. **[LO 1.8 H]**

1.53 What is the weight of digit 5 in the decimal number 9536? **[LO 1.8 M]**

1.54 What is the 4-bit binary representation of the decimal number 12? **[LO 1.9 E]**

1.55 Explain in detail the concept of hexadecimal system. **[LO 1.9 M]**

1.56 Why are binary codes used by computer systems **[LO 1.9 M]**

1.57 What do you understand by digital codes? Explain the two different types of digital codes. **[LO 1.9 M]**

1.58 Why are the number system conversions implemented in a computer system? **[LO 1.9 M]**

1.59 Explain in detail the different categories of number system conversions. **[LO 1.9 M]**

1.60 How is binary number converted into its decimal equivalent? **[LO 1.9 E]**

1.61 What is the hexadecimal representation of octal number 6235? **[LO 1.9 H]**

1.62 What is the binary equivalent of 859.238? **[LO 1.9 H]**

1.63 What do you understand by computer arithmetic? Are the rules for performing computer arithmetic and decimal arithmetic same? **[LO 1.10 M]**

1.64 What are the different computer arithmetic operations? Explain all of them with their associated set of rules. **[LO 1.10 M]**

1.65 Perform the binary addition of 1000010, 0111010 and 11110101. **[LO 1.10 M]**

1.66 Why is binary multiplication considered as the process of repetitive addition? **[LO 1.10 H]**

1.67 Perform the binary multiplication of 15 and 17. **[LO 1.10 H]**

1.68 Perform the binary division of 141 and 21. **[LO 1.10 H]**

1.69 What are the different laws of arithmetic? **[LO 1.10 E]**

1.70 What are logic gates? Why are they important? **[LO 1.11 E]**

1.71 Explain the different types of basic logic gates. **[LO 1.11 E]**

1.72 Explain the basic concept of truth table and also describe the truth tables of all the basic logic gates. **[LO 1.11 M]**

1.73 Explain the basic steps required to convert a Boolean expression into logic gates. **[LO 1.11 M]**

1.74 What is assembly language? What are its main advantages? **[LO 1.12 M]**

1.75 What is high level language? What are the different types of high level languages? **[LO 1.12 M]**

1.76 What do we understand by a compiler and an assembler? **[LO 1.12 H]**

1.77 What is flow chart? How is it different from an algorithm? **[LO 1.13 H]**

1.78 What are the functions of a flow chart? **[LO 1.13 E]**

Fundamentals of C

Chapter 2

LEARNING OBJECTIVES

LO 2.1 Produce an overview of C programming language

LO 2.2 Exemplify the elementary C concepts through sample programs

LO 2.3 Illustrate the use of user-defined functions and math functions through sample programs

LO 2.4 Describe the basic structure of C program

LO 2.5 Recognize the programming style of C language

LO 2.6 Describe how a C program is compiled and executed

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. 2.1.

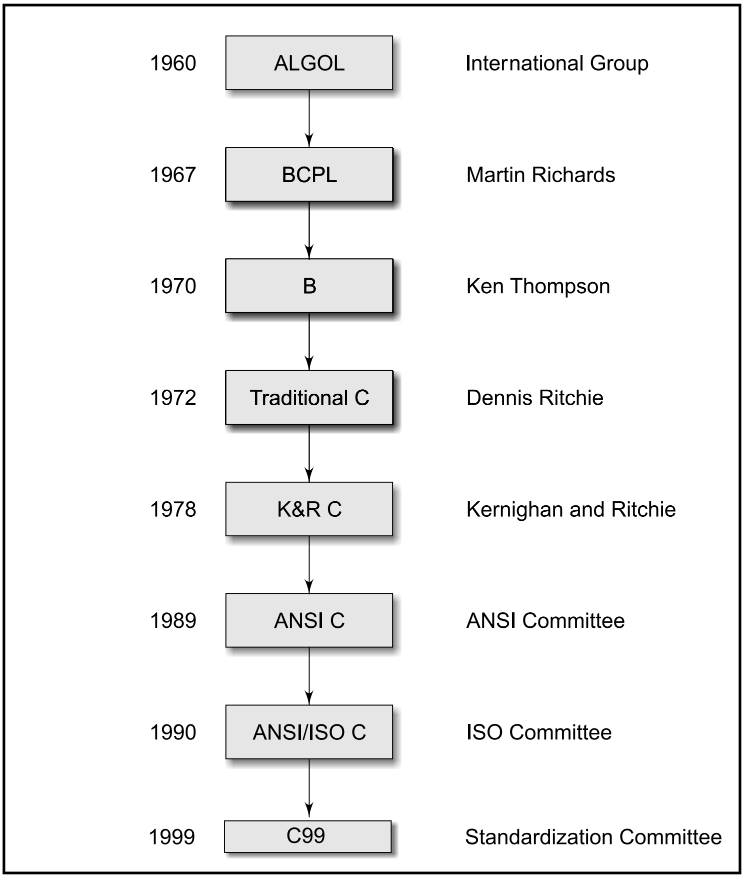


Fig. 2.1 *History of ANSI C*

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

IMPORTANCE OF C

LO 2.1
Produce an overview of C programming language

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.

SAMPLE PROGRAM 1: PRINTING A MESSAGE

LO 2.2
Exemplify the elementary C concepts through sample programs

Consider a very simple program given in Fig. 2.2.

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

Fig. 2.2 *A program to print one line of text*

This program when executed will produce the following output:

```
I see, I remember
```

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

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 5).

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

In this case, the function body contains three statements out of which only the **printf** line is an executable statement. The lines beginning with **/*** and ending with ***/** are known as *comment* lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between **/*** and ***/** is ignored by the compiler. In general, a comment can be inserted wherever blank spaces can occur—at the beginning, middle or end of a line—"but never in the middle of a word".

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

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

is not valid and therefore results 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.

```
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 concepts of compilation and linking are explained later in this chapter. The **printf** function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

```
I see, I remember
```

Note that the print line ends with a semicolon. *Every statement in C should end with a semicolon (;) mark.*

Suppose we want to print the above quotation in two lines as

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

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

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

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

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

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

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

This is similar to the output of the program in Fig. 2.2. However, note that there is no space between, and I.

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 output

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

while the statement

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

will print out

```
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 2.3 highlights the general format of such simple programs. All C programs need a **main** function.

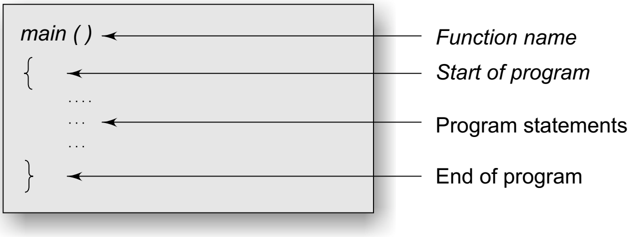


Fig. 2.3 *Format of simple C programs*

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.

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. 2.4.

```
/* Programm 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 */
}                                            /*         line13 */
```

Fig. 2.4 *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;
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. 2.4. All declaration statements end with a semicolon; C supports many other data types.

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.** The statements

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

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**. The print statement

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

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

```
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.

SAMPLE PROGRAM 3: INTEREST CALCULATION

The program in Fig. 2.5 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. 2.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = Value at start of year (1 + 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. 2.5 *Program for investment problem*

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.

```
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. 2.6 *Output of the investment 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 statement

```
PRINCIPAL = 10000.00;
```

is illegal.

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

```
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**. The concept and types of loops are discussed in Chapter 3.

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

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. 2.7 uses a user-defined function. A function defined by the user is equivalent to a subroutine in FORTRAN or subprogram in BASIC.

LO 2.3
Illustrate the use of user-defined functions and math functions through sample programs

Figure 2.7 presents a very simple program that uses a **mul ( )** function. The program will print the following output.

Multiplication of 5 and 10 is 50

```
/*———————— 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. 2.7 *A program using a user-defined function*

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 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. User-defined functions are considered in detail in chapter 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 is of the form

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

math.h is the filename containing the required function. Figure 2.8 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 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. 2.8 *Program using a math function*

Another **#include** instruction that is often required is

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

stdio.h refers to the *standard* I/O header file containing standard input and output functions

The *#include* Directive

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

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

```
#include<filename>
```

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

BASIC STRUCTURE OF C PROGRAMS

LO 2.4
Describe the basic structure of C program

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more *statements* designed to perform a *specific task*. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 2.9.

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.

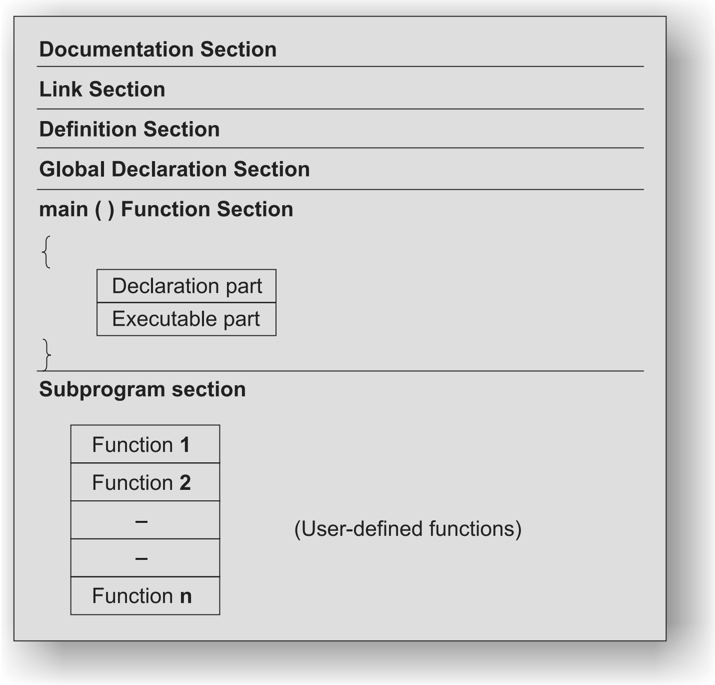


Fig. 2.9 *An overview of a C program*

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.

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.

LO 2.5
Recognize the Programming style of C language

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. Note how the braces are aligned and the statements are indented in the program of Fig. 2.5.

Since C is a free-form language, we can group statements together on one line. The statements

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

can be written on one line as

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

The program

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

may be written in one line like

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

However, this style make the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

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 2.10 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.

UNIX SYSTEM

Creating the Program

LO 2.6
Describe how a C program is compiled and executed

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

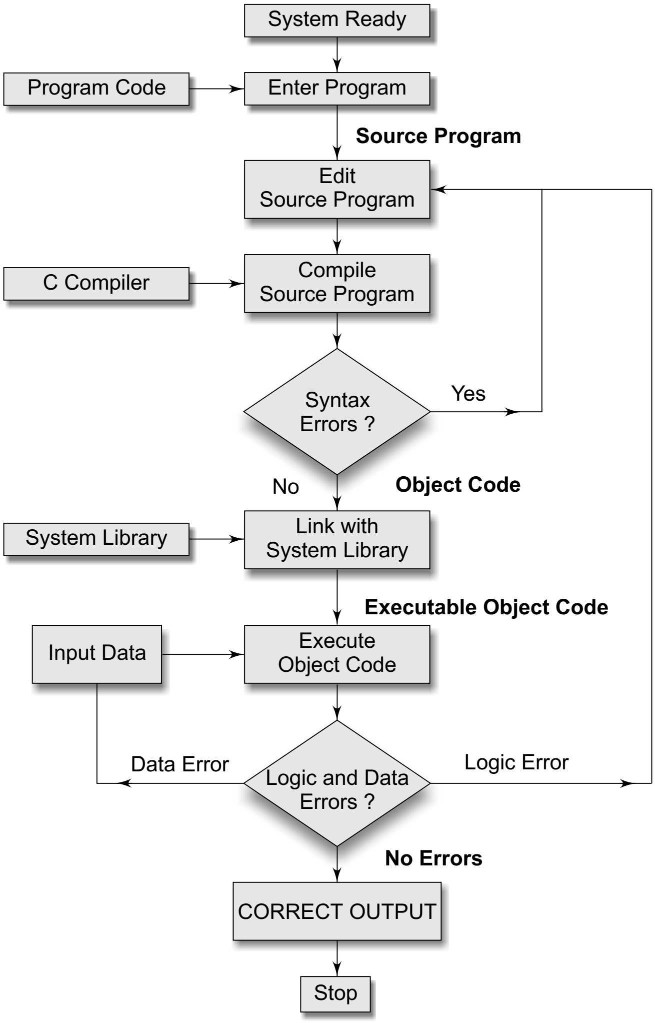


Fig. 2.10 *Process of compiling and runnig a C program*

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

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

ed *filename*

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system manual.)

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

Compiling and Linking

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

```
cc ebg1.c
```

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

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

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

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

Note that some systems use different compilation command for linking mathematical functions.

```
cc filename - lm
```

is the command under UNIPLUS SYSTEM V operating system.

Executing the Program

Execution is a simple task. The command

```
a.out
```

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

Creating Your Own Executable File

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

```
mv a.out name
```

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

```
cc –o name source-file
```

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

Multiple Source Files

To compile and link multiple source program files, we must append all the files names to the **cc** command.

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

These files will be separately compiled into object files called

```
filename-i.o
```

and then linked to produce an executable program file **a.out** as shown in Fig. 2.11.

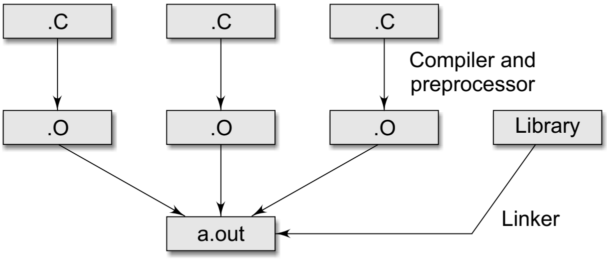


Fig. 2.11 *Compilation of multiple files*

It is also possible to compile each file separately and link them later. For example, the commands

```
cc –c mod1.c
cc –c mod2.c
```

will compile the source files *mod1.c* and *mod2.c* into objects files *mod1.o* and *mod2.o*. They can be linked together by the command

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

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

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

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

MS-DOS SYSTEM

The program can be created using any word processing software in non-document mode. The file name should end with the characters ".c" like **program.c, pay.c,** etc. Then the command

```
MSC pay.c
```

under MS-DOS operating system would load the program stored in the file **pay.c** and generate the **object code.** This code is stored in another file under name **pay.obj**. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the command

```
LINK pay.obj
```

which generates the **executable code** with the filename **pay.exe**. Now the command

```
pay
```

would execute the program and give the results.

KEY CONCEPTS

- **#DEFINE: (Is a preprocessor compiler directive.) [LO 2.2]**
- **PRINTF: (Is a predefined standard C function that writes the output to the stdout (standard output) stream.) [LO 2.2]**
- **SCANF: (Is a predefined standard C function that reads formatted input from stdin (standard input) stream.) [LO 2.2]**
- **PROGRAM: (Is a sequence of instructions written to perform a specific task in the computer.) [LO 2.4]**

ALWAYS REMEMBER

- C is a structured, high-level, machine independent language. **[LO 2.1]**
- ANSI C and C99 are the standardized versions of C language. **[LO 2.1]**
- C combines the capabilities of assembly language with the features of a high level language. **[LO 2.1]**
- C is robust, portable and structured programming language. **[LO 2.1]**
- Every C program requires a **main()** function (Use of more than one **main()** is illegal). The place **main** is where the program execution begins. **[LO 2.2]**
- The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace. **[LO 2.2]**
- C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings. **[LO 2.2]**
- All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark. **[LO 2.2]**
- Every program statement in a C language must end with a semicolon. **[LO 2.2]**
- All variables must be declared for their types before they are used in the program. **[LO 2.2]**
- A comment can be inserted almost anywhere a space can appear. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and * appropriately. **[LO 2.2]**
- Compiler directives such as **define** and **include** are special instructions to the compiler to help it compile a program. They do not end with a semicolon. **[LO 2.2]**
- The sign # of compiler directives must appear in the first column of the line. **[LO 2.2]**
- We must make sure to include header files using **#include** directive when the program refers to special names and functions that it does not define. **[LO 2.3]**
- The structure of a C program comprises of various sections including Documentation, Link, Definition, Global Declaration, main () function and Sub program section. **[LO 2.4]**
- C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program. **[LO 2.5]**
- The execution of a C program involves a series of steps including: creating the program, compiling the program, linking the program with functions from C library and executing the program. **[LO 2.6]**
- The command used for running a C program in UNIX system is *a.out*. **[LO 2.6]**
- The command used for running a C program in MS-DOS system is *file.exe* where file is the name of the program that has already been compiled. **[LO 2.6]**
- When braces are used to group statements, make sure that the opening brace has a corresponding closing brace. **[LO 2.6]**

REVIEW QUESTIONS

2.1 State whether the following statements are *true* or *false*.

(a) Every line in a C program should end with a semicolon. **[LO 2.2 E]**

(b) The closing brace of the **main( )** in a program is the logical end of the program. **[LO 2.2 E]**

(c) Comments cause the computer to print the text enclosed between /* and */ when executed. **[LO 2.2 E]**

(d) Every C program ends with an END word. **[LO 2.2 M]**

(e) A **printf** statement can generate only one line of output. **[LO 2.2 M]**

E for Easy, **M** for Medium and **H** for High

(f) The purpose of the header file such as **stdio.h** is to store the source code of a program. **[LO 2.3 M]**

(g) A line in a program may have more than one statement. **[LO 2.5 M]**

(h) Syntax errors will be detected by the compiler. **[LO 2.6 M]**

(i) In C language lowercase letters are significant. **[LO 2.2 H]**

(j) **main( )** is where the program begins its execution. **[LO 2.2 H]**

2.2 Which of the following statements are *true*?

(a) Every C program must have at least one user-defined function. **[LO 2.3 E]**

(b) Declaration section contains instructions to the computer. **[LO 2.4 E]**

(c) Only one function may be named **main( )**. **[LO 2.2 M]**

2.3 Which of the following statements about comments are *false*?

(a) Comments serve as internal documentation for programmers. **[LO 2.2 E]**

(b) In C, we can have comments inside comments. **[LO 2.2 M]**

(c) Use of comments reduces the speed of execution of a program. **[LO 2.2 H]**

(d) A comment can be inserted in the middle of a statement. **[LO 2.2 H]**

2.4 Fill in the blanks with appropriate words in each of the following statements.

(a) Every program statement in a C program must end with a ___________. **[LO 2.2 E]**

(b) The ____________ Function is used to display the output on the screen. **[LO 2.2 E]**

(c) The ____________ header file contains mathematical functions. **[LO 2.3 M]**

(d) The escape sequence character ____________ causes the cursor to move to the next line on the screen. **[LO 2.2 H]**

2.5 Remove the semicolon at the end of the **printf** statement in the program of Fig. 2.2 and execute it. What is the output? **[LO 2.2 M]**

2.6 In the Sample Program 2, delete line-5 and execute the program. How helpful is the error message? **[LO 2.2 H]**

2.7 Modify the Sample Program 3 to display the following output: **[LO 2.2 M]**

Year	**Amount**
1	5500.00
2	6160.00
—	______________
—	______________
10	14197.11

2.8 Why and when do we use the **#define** directive? **[LO 2.2 E]**

2.9 Why and when do we use the **#include** directive? **[LO 2.3 M]**

2.10 What does **void main(void)** mean? **[LO 2.2 H]**

2.11 Distinguish between the following pairs:

(a) main( ) and void main(void) **[LO 2.2 H]**

(b) int main( ) and void main( ) **[LO 2.2 M]**

2.12 Why do we need to use comments in programs? **[LO 2.2 E]**

2.13 Why is the look of a program is important? **[LO 2.5 E]**

2.14 Where are blank spaces permitted in a C program? **[LO 2.5 M]**

2.15 Describe the structure of a C program. **[LO 2.4 M]**

2.16 Describe the process of creating and executing a C program under UNIX system. **[LO 2.6 M]**

2.17 How do we implement multiple source program files? **[LO 2.6 H]**

DEBUGGING EXERCISES

2.1 Find errors, if any, in the following program: **[LO 2.2 E]**

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

2.2 Find errors, if any, in the following program: **[LO 2.2 M]**

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

2.3 Find errors, if any, in the following program: **[LO 2.3 H]**

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

PROGRAMMING EXERCISES

2.1 Write a program to display the equation of a line in the form

$$ax + by = c$$

for $a = 5$, $b = 8$ and $c = 18$. **[LO 2.2 E]**

2.2 Write a program that will print your mailing address in the following form: **[LO 2.2 M]**

First line	:	Name
Second line	:	Door No, Street
Third line	:	City, Pin code

2.3 Write a program to output the following multiplication table: **[LO 2.2 M]**

$5 \times 1 = 5$
$5 \times 2 = 10$
$5 \times 3 = 15$
• •
• •
$5 \times 10 = 50$

2.4 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 **[LO 2.2 M]**

(b) a = 300, b = 70, c = 70 **[LO 2.2 M]**

Comment on the output in each case.

2.5 Relationship between Celsius and Fahrenheit is governed by the formula

$$F = \frac{9C}{5} + 32$$

Write a program to convert the temperature

(a) from Celsius to Fahrenheit and **[LO 2.2 M]**

(b) from Fahrenheit to Celsius. **[LO 2.2 M]**

2.6 Given the radius of a circle, write a program to compute and display its area. Use a symbolic constant to define the π value and assume a suitable value for radius. **[LO 2.3 M]**

2.7 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: **[LO 2.3 M]**

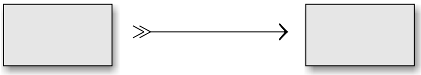
```
20 + 10 = 30
20 – 10 = 10
```

2.8 Modify the above program to provide border lines to the address. **[LO 2.2 H]**

2.9 Write a program using one print statement to print the pattern of asterisks as shown below: **[LO 2.2 H]**

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

2.10 Write a program that will print the following figure using suitable characters. **[LO 2.2 H]**



2.11 Area of a triangle is given by the formula

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

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. **[LO 2.2 H]**

2.12 Write a program to display the following simple arithmetic calculator **[LO 2.2 H]**

x =	□	y =	□
sum	□	Difference =	□
Product =	□	Division =	□

2.13 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. **[LO 2.3 H]**

2.14 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. (Hint: use the formula given in the Ex. 2.11) **[LO 2.3 H]**

2.15 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. **[LO 2.3 H]**

Chapter

3

Control Structure in C

LEARNING OBJECTIVES

LO 3.1 Discuss decision making with if statement

LO 3.2 Describe if...else statement

LO 3.3 Explain switch statement

LO 3.4 Know conditional operator

LO 3.5 Illustrate how goto statement is used for unconditional branching

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*.

We have already used some of these statements in the earlier examples. Here, we shall discuss their features, capabilities and applications in more detail.

DECISION MAKING WITH IF STATEMENT

LO 3.1
Discuss 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.

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 section.

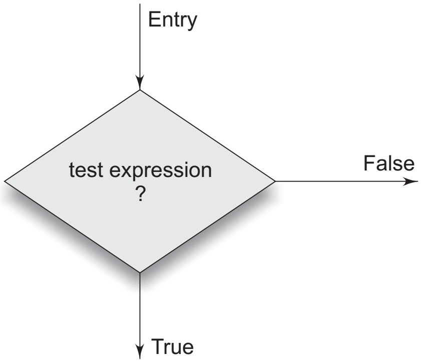


Fig. 3.1 *Two-way branching*

SIMPLE IF STATEMENT

The general form of a simple **if** statement is

```
                if (test expression)
                {
                    statement-block;
                }
                statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x.* Remember, when the condition is true both the statement-block and the statement-x are executed in sequence. This is illustrated in Fig. 3.2.

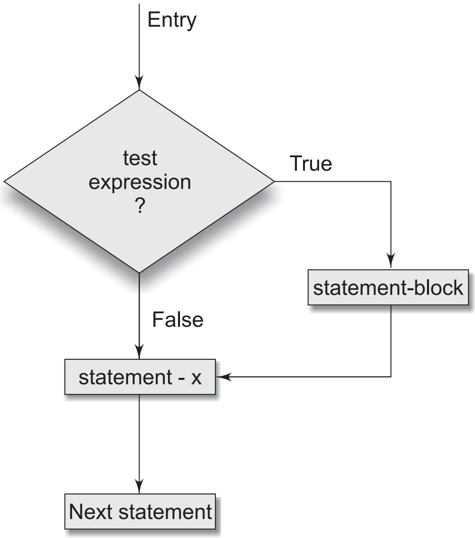


Fig. 3.2 *Flowchart of simple* ***if*** *control*

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);
.........
.........
```

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 adde.

WORKED-OUT PROBLEM 3.1 E

The program in Fig. 3.3 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. 3.3 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);
                    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. 3.3 *Illustration of simple **if** statement*

E for Easy, **M** for Medium and **H** for High

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. The Worked-Out Problem 3.2 illustrates this.

WORKED-OUT PROBLEM 3.2

The program in Fig. 3.4 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. 3.4 *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||!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

THE IF.....ELSE STATEMENT

LO 3.2
Describe 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. 3.5. In both the cases, the control is transferred subsequently to the statement-x.

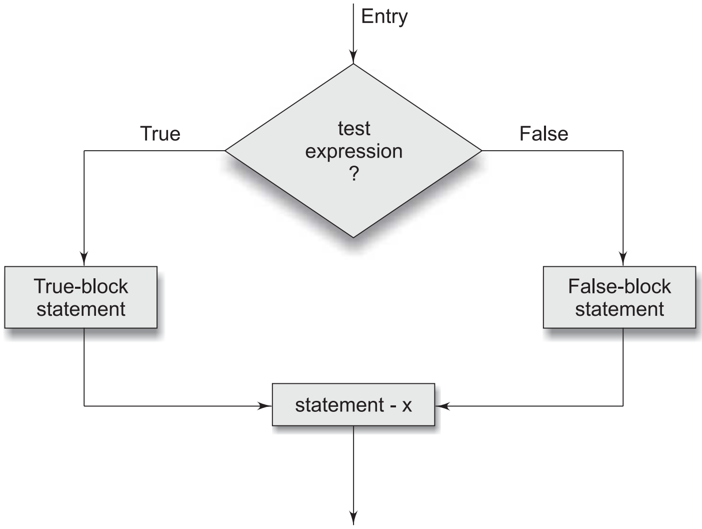


Fig. 3.5 *Flowchart of **if......else** control*

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 **xxxxxx**, 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. 3.3. 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");
..........
.........
```

WORKED-OUT PROBLEM 3.3

H

A program to evaluate the power series.

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

is given in Fig. 3.6. It uses **if......else** to test the accuracy.

The power series contains the recurrence relationship of the type

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

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

$$T_0 = 1$$

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

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

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;
```

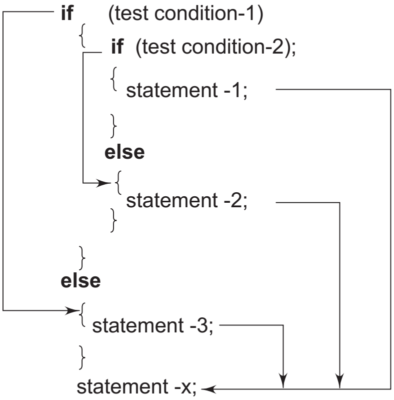
```
                while (n <= 100)
             {
                term = term * x/n;
                sum = sum + term;
                count = count + 1;
                if (term < ACCURACY)
                   n = 999;
                else
                   n = n + 1;
             }
             printf("Terms = %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. 3.6 *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.

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 below:



The logic of execution is illustrated in Fig. 3.7. 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.

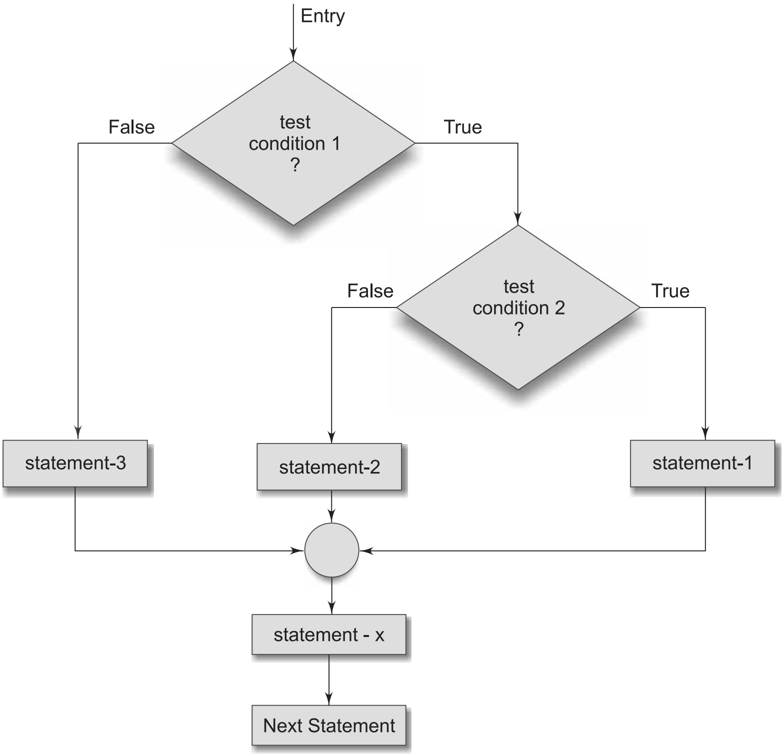


Fig. 3.7 *Flow chart of nested **if…else** statements*

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 ₹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**.

WORKED-OUT PROBLEM 3.4 H

The program in Fig. 3.8 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. 3.8 *Selecting the largest of three numbers*

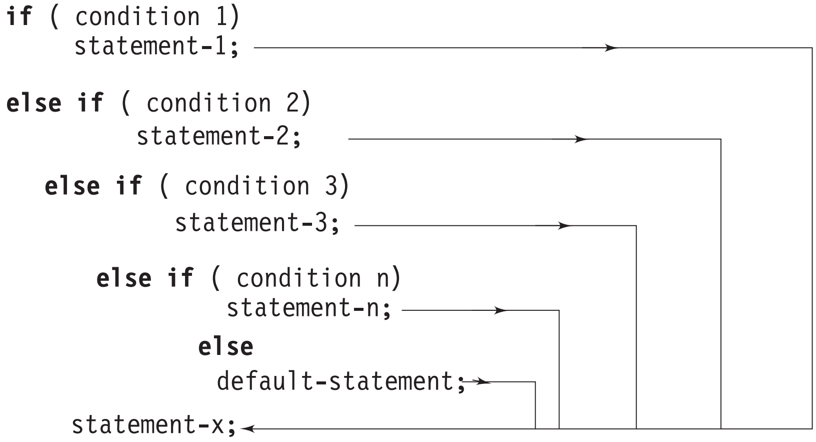
Dangling Else Problem

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**"

THE ELSE IF LADDER

There is another way of putting **if**s together when multipath decisions are involved. A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if**. It takes the following general form:



```
if ( condition 1)
     statement-1;

else if ( condition 2)
         statement-2;

   else if ( condition 3)
            statement-3;

       else if ( condition n)
                statement-n;
            else
             default-statement;

     statement-x;
```

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. 3.9 shows the logic of execution of **else if** ladder statements.

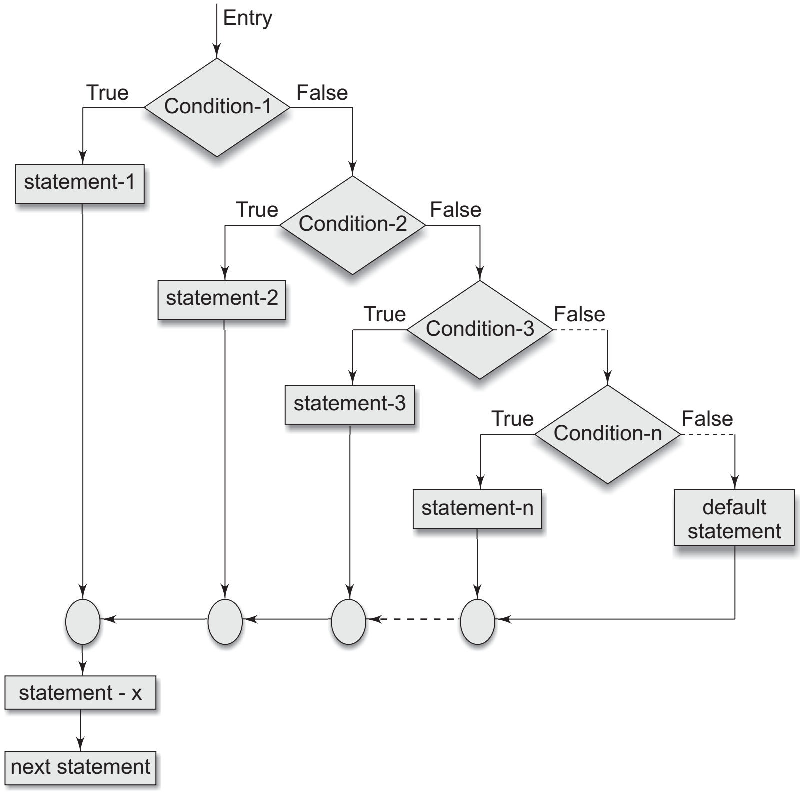


Fig. 3.9 *Flow chart of* **else..if** *ladder*

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.

WORKED-OUT PROBLEM 3.5

H

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

Consumption Units	*Rate of Charge*
0 – 200	₹0.50 per unit
201 – 400	₹100 plus ₹0.65 per unit excess of 200
401 – 600	₹230 plus ₹0.80 per unit excess of 400
601 and above	₹390 plus ₹1.00 per unit excess of 600

The program in Fig. 3.10 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. 3.10 *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 below:

- 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.

THE SWITCH STATEMENT

LO 3.3
Explain 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**.

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. 3.11.

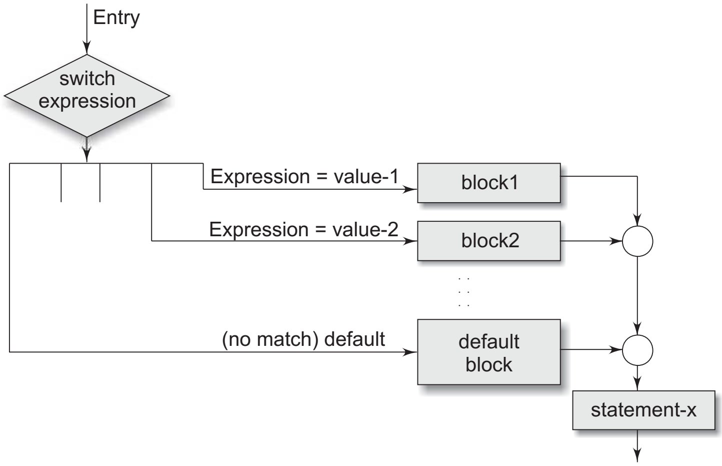


Fig. 3.11 *Selection process of the **switch** statement*

The **switch** statement can be used to grade the students as discussed in the last section. 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.

Marks	**Index**
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;
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.

WORKED-OUT PROBLEM 3.6

E

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

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:

LO 3.4
Know conditional operator

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.

WORKED-OUT PROBLEM 3.7

M

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. 3.13.

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. 3.13 *Illustration of the conditional operator*

The program uses the following variables:

loan3	-	present loan amount requested
loan2	-	previous loan amount pending
loan1	-	previous to previous loan pending
sum23	-	sum of loan2 and loan3
sancloan	-	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.

WORKED-OUT PROBLEM 3.8

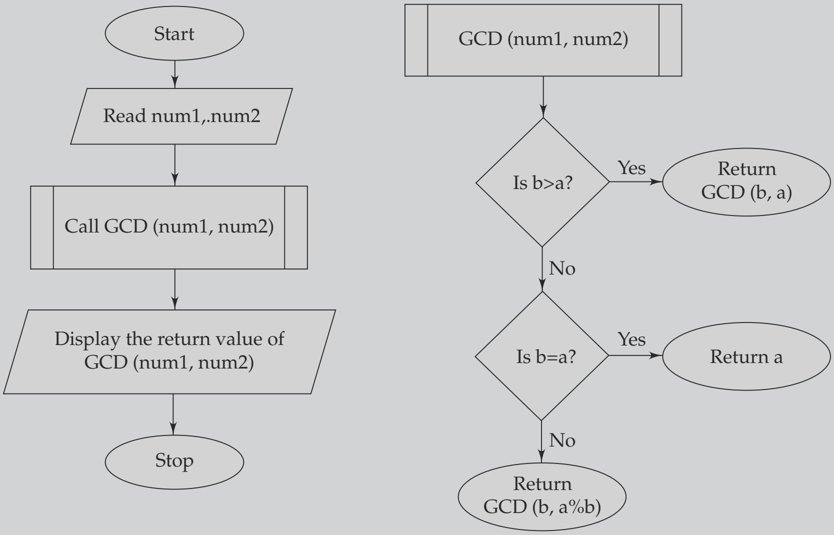
M

Write a program to determine the Greatest Common Divisor (GCD) of two numbers.

```
Algorithm
Step 1 – Start
Step 2 – Accept the two numbers whose GCD is to be found (num1, num2)
Step 3 – Call function GCD(num1,num2)
Step 4 – Display the value returned by the function call GCD(num1, num2)
Step 5 – Stop
GCD(a,b)
Step 1 – Start
Step 2 – If b > a goto Step 3 else goto Step 4
Step 3 – Return the result of the function call GCD(b,a) to the calling function
Step 4 – If b = 0 goto Step 5 else goto Step 6
Step 5 – Return the value a to the calling function
Step 6 – Return the result of the function call GCD(b,a mod b) to the calling function
```

Program

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

int GCD(int m, int n);
void main()
```



```
     {
          int num1,num2;
          clrscr();
          printf("Enter the two numbers whose GCD is to be found: ");
          scanf("%d %d",&num1,&num2);

          printf("\nGCD of %d and %d is %d\n",num1,num2,GCD(num1,num2));
          getch();
     }

     int GCD(int a, int b)
     {
          if(b>a)
             return GCD(b,a);
          if(b==0)
             return a;
          else
             return GCD(b,a%b);
     }
```

```
Output
Enter the two numbers whose GCD is to be found: 18 12
GCD of 18 and 12 is 6
```

Fig. 3.14 *Program to determine GCD of two numbers*

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.

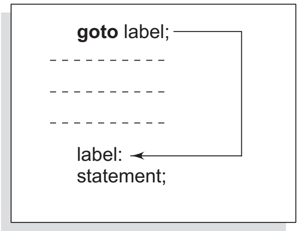
- Avoid compound negative statements. Use positive statements wherever possible.
- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.
- Use the most probable condition first. 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.

THE GOTO STATEMENT

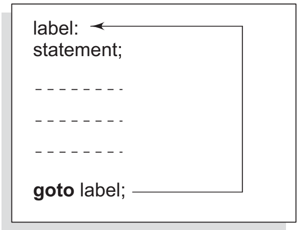
So far we have discussed ways of controlling the flow of execution based on certain specified conditions. Like many other languages, C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of **goto** might be desirable.

The **goto** requires a *label* in order to identify the place where the branch is to be made. A *label* is any valid variable name, and must be followed by a colon. The *label* is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and *label* statements are shown below:

LO 3.5
Illustrate how goto statement is used for unconditional branching



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. Worked-Out Problem 3.9 illustrates how such infinite loops can be eliminated.

WORKED-OUT PROBLEM 3.9

E

Program presented in Fig. 3.15 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)
            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. 3.15 *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;
  ----
  }                                    Jumping
----                                   out of
----                                   loops
}
end_of_program:
```

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.

KEY CONCEPTS

- **DECISION-MAKING STATEMENTS:** Are the statements that control the flow of execution in a program. **[LO 3.1]**
- **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. **[LO 3.3]**
- **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. **[LO 3.4]**
- **GOTO STATEMENT:** Is used for unconditional branching. It transfers the flow of control to the place where matching label is found. **[LO 3.5]**
- **INFINITE LOOP:** Is a condition where a set of instructions is repeatedly executed. **[LO 3.5]**

ALWAYS REMEMBER

- Be aware of any side effects in the control expression such as if(x++). **[LO 3.1]**
- Check the use of =operator in place of the equal operator = =. **[LO 3.1]**
- Do not give any spaces between the two symbols of relational operators = =, !=, >= and <=. **[LO 3.1]**
- Writing !=, >= and <= operators like =!, => and =< is an error. **[LO 3.1]**
- Remember to use two ampersands (&&) and two bars (||) for logical operators. Use of single operators will result in logical errors. **[LO 3.1]**
- Do not forget to place parentheses for the if expression. **[LO 3.1]**
- It is an error to place a semicolon after the if expression. **[LO 3.1]**
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal. **[LO 3.1]**
- 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. **[LO 3.1]**
- Be aware of dangling **else** statements. **[LO 3.2]**
- Use braces to encapsulate the statements in **if** and **else** clauses of an if…. else statement. **[LO 3.2]**
- Do not forget to use a break statement when the cases in a switch statement are exclusive. **[LO 3.3]**
- Although it is optional, it is a good programming practice to use the default clause in a switch statement. **[LO 3.3]**
- It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.) **[LO 3.3]**
- Do not use the same constant in two case labels in a switch statement. **[LO 3.3]**
- Try to use simple logical expressions. **[LO 3.4]**
- Be careful while placing a goto label in a program as it may lead to an infinite loop condition. **[LO 3.5]**

BRIEF CASES

1. Range of Numbers [LO 3.1, 3.2 M]

Problem: A survey of the computer market shows that personal computers are sold at varying costs by the vendors. The following is the list of costs (in hundreds) quoted by some vendors:

35.00, 40.50, 25.00, 31.25, 68.15,
47.00, 26.65, 29.00 53.45, 62.50

Determine the average cost and the range of values.

Problem analysis: Range is one of the measures of dispersion used in statistical analysis of a series of values. The range of any series is the difference between the highest and the lowest values in the series. That is

Range = highest value – lowest value

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

Program: A program to determine the range of values and the average cost of a personal computer in the market is given in Fig. 3.16.

```
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. 3.16 *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 [LO 3.2, 3.5 M]

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

Level	*Perks*	
	Conveyance allowance	*Entertainment allowance*
1	1000	500
2	750	200
3	500	100
4	250	—

An executive's gross salary includes basic pay, house rent allowance at 25% of basic pay and other perks. Income tax is withheld from the salary on a percentage basis as follows:

Gross salary	*Tax rate*
Gross <= 2000	No tax deduction
2000 < Gross <= 4000	3%
4000 < Gross <= 5000	5%
Gross > 5000	8%

Write a program that will read an executive's job number, level number, and basic pay and then compute the net salary after withholding income tax.

***Problem analysis**:*

Gross salary = basic pay + house rent allowance + perks

Net salary = Gross salary – income tax.

The computation of perks depends on the level, while the income tax depends on the gross salary. The major steps are:

1. Read data.
2. Decide level number and calculate perks.
3. Calculate gross salary.
4. Calculate income tax.
5. Compute net salary.
6. Print the results.

Program: A program and the results of the test data are given in Fig. 3.17. 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. 3.17 *Pay-bill calculations*

REVIEW QUESTIONS

3.1 State whether the following are *true* or *false*:

(a) A **switch** expression can be of any type. **[LO 3.3 E]**

(b) A program stops its execution when a **break** statement is encountered. **[LO 3.3 E]**

(c) Each case label can have only one statement. **[LO 3.3 E]**

(d) The **default** case is required in the **switch** statement. **[LO 3.3 E]**

(e) When **if** statements are nested, the last **else** gets associated with the nearest **if** without an **else**. **[LO 3.2 M]**

(f) One **if** can have more than one **else** clause. **[LO 3.2 M]**

(g) Each expression in the **else if** must test the same variable. **[LO 3.2 M]**

(h) A **switch** statement can always be replaced by a series of **if..else** statements. **[LO 3.3 M]**

(i) Any expression can be used for the **if** expression. **[LO 3.1 H]**

(j) The predicate !((x >= 10)¦(y = = 5)) is equivalent to (x < 10) && (y !=5). **[LO 3.1 H]**

3.2 Fill in the blanks in the following statements.

(a) The _______ operator is true only when both the operands are true. **[LO 3.1 E]**

(b) Multiway selection can be accomplished using an **else if** statement or the __________ statement. **[LO 3.3 E]**

(c) The ______ statement when executed in a **switch** statement causes immediate exit from the structure. **[LO 3.3 E]**

(d) The expression ! (x ! = y) can be replaced by the expression ________. **[LO 3.1 M]**

(e) The ternary conditional expression using the operator ?: could be easily coded using ______ statement. **[LO 3.4 M]**

3.3 The following is a segment of a program: **[LO 3.1 M]**

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

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

3.4 Rewrite each of the following without using compound relations:

(a) `if (grade <= 59 && grade >= 50)` **[LO 3.1 E]**

```
second = second + 1;
```

(b) `if (number > 100 || number < 0)` **[LO 3.2 M]**

```
    printf(" Out of range");
else
    sum = sum + number;
```

(c) `if ((M1 > 60 && M2 > 60) || T > 200)` **[LO 3.2 M]**

```
    printf(" Admitted\n");
else
    printf(" Not admitted\n");
```

3.5 Assuming x = 10, state whether the following logical expressions are true or false. **[LO 3.1 H]**

(a) x = = 10 && x > 10 && !x

(b) x = = 10 || x > 10 && ! x

(c) x = = 10 && x > 10 || ! x

(d) x = = 10 || x > 10 || !x

3.6 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 **[LO 3.3 H]**

(a) `switch (y);`

(b) `case 10;`

(c) `switch (x + y)`

(d) `switch (x) {case 2: y = x + y; break};`

3.7 Simplify the following compound logical expressions **[LO 3.1 M]**

(a) !(x <=10)

(b) !(x = = 10) ||! ((y = = 5) || (z < 0))

(c) ! ((x +y = = z) && !(z > 5)

(d) !((x <=5) && (y = = 10) & & (z < 5))

3.8 Assuming that x = 5, y = 0, and z = 1 initially, what will be their values after executing the following code segments? **[LO 3.2 M]**

(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;
```

3.9 Assuming that x = 2, y = 1 and z = 0 initially, what will be their values after executing the following code segments? **[LO 3.3 M]**

(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;
  }
```

3.10 What is the output of the following program? **[LO 3.2 M]**

```
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);
}
```

3.11 What is the output of the following program? **[LO 3.2 M]**

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

```
        else;
        printf(" END");
}
```

3.12 What is the output of the following program? **[LO 3.4 M]**

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

3.13 What is the output of the following program? **[LO 3.5 M]**

```
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);
}
```

3.14 What will be the value of x when the following segment is executed? **[LO 3.4 E]**

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

3.15 What will be the output when the following segment is executed? **[LO 3.2 M]**

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

3.16 What will be the output when the following segment is executed? **[LO 3.3 M]**

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

3.17 What will be the output of the following segment when executed? **[LO 3.2 E]**

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

3.18 What will be output of the following segment when executed? **[LO 3.2 M]**

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

DEBUGGING EXERCISES

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

(a) `if (x + y = z && y > 0)` **[LO 3.1 E]**

```
         printf(" ");
```

(b) `if (p < 0) || (q < 0)` **[LO 3.1 E]**

```
         printf (" sign is negative");
```

(c) `if (code > 1);` **[LO 3.2 M]**

```
         a = b + c
else
         a = 0
```

3.2 Find the error, if any, in the following statements: **[LO 3.1 H]**

(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") ;
```

PROGRAMMING EXERCISES

3.1 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 **[LO 3.1 E]**, and (b) with **else** option. **[LO 3.2 E]**

3.2 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. **[LO 3.1 M]**

3.3 A set of two linear equations with two unknowns x1 and x2 is given below: **[LO 3.2 H]**

$ax_1 + bx_2 = m$

$cx_1 + dx_2 = n$

The set has a unique solution

$$x1 = \frac{md - bn}{ad - cb}$$

$$x2 = \frac{na - mc}{ad - cb}$$

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.

3.4 Given a list of marks ranging from 0 to 100, write a program to compute and print the number of students: **[LO 3.2 M]**

(a) who have obtained more than 80 marks,
(b) who have obtained more than 60 marks,
(c) who have obtained more than 40 marks,
(d) who have obtained 40 or less marks,
(e) in the range 81 to 100,
(f) in the range 61 to 80,
(g) in the range 41 to 60, and
(h) in the range 0 to 40.

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

3.5 Admission to a professional course is subject to the following conditions: **[LO 3.2 M]**

(a) Marks in Mathematics >= 60
(b) Marks in Physics >= 50
(c) Marks in Chemistry >= 40
(d) 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.

3.6 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. **[LO 3.2 H]**

Square Root Table

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

3.7 Shown below is a Floyd's triangle. **[LO 3.2 H]**

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

3.8 A cloth showroom has announced the following seasonal discounts on purchase of items: **[LO 3.1, 3.3 H]**

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.

3.9 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, **[LO 3.1 M]**

(b) **else if** statements, and **[LO 3.2 M]**

(c) conditional operator ? : **[LO 3.4 M]**

3.10 Write a program to compute the real roots of a quadratic equation **[LO 3.2 H]**

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

The roots are given by the equations

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

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

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 $b^2 – 4 ac$ 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.

3.11 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. **[LO 3.2 M]**

3.12 An electricity board charges the following rates for the use of electricity: **[LO 3.2 H]**

For the first 200 units: 80 P per unit

For the next 100 units: 90 P per unit

Beyond 300 units: ₹1.00 per unit

All users are charged a minimum of ₹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.

3.13 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. **[LO 3.2 M]**

3.14 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. **[LO 3.2 M]**

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.

3.15 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 **[LO 3.2 M]**, and (ii) **switch** statement. **[LO 3.3 M]**

Array & String

Chapter

4

LEARNING OBJECTIVES

LO 4.1	Define the concept of arrays
LO 4.2	Determine how one-dimensional array is declared and initialized
LO 4.3	Know the concept of two-dimensional arrays
LO 4.4	Discuss how two-dimensional array is declared and initialized
LO 4.5	Describe multi-dimensional arrays
LO 4.6	Explain dynamic arrays
LO 4.7	Discuss how string variables are declared and initialized
LO 4.8	Explain how strings are read from terminal
LO 4.9	Describe how strings are written to screen
LO 4.10	Illustrate how strings are manipulated

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. However, we shall consider structures in Chapter 7 and lists in Chapter 8.

As we mentioned earlier, an array is a sequenced collection of related data items that share a common name. For instance, we can use an array name *salary* to represent a *set of salaries* of a group of employees in an organization. We can refer to the individual salaries by writing a number called *index* or *subscript* in brackets after the array name. For example,

```
salary [10]
```

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

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

We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions.

In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.

- One-dimensional arrays
- Two-dimensional arrays
- Multidimensional arrays

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:

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include the following:

“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

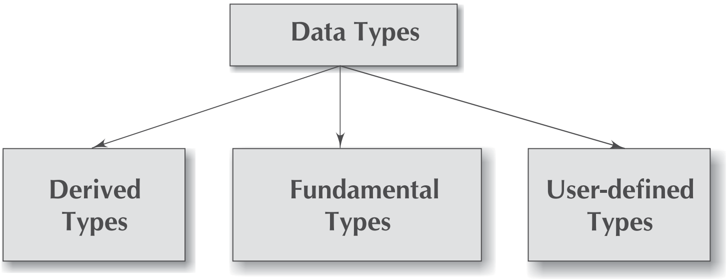
```
Well Done!
```

- 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.

Data Structures

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



- Arrays
- Functions
- Pointers

- Integral Types
- Float Types
- Character Types

- Structures
- Unions
- 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

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

LO 4.1
Define the concept of arrays

$$A = \frac{\sum_{i=1}^{n} x_i}{n}$$

to calculate the average of n values of *x.* The subscripted variable x_i refers to the ith element of *x.* In C, single-subscripted variable x_i can be expressed as

x[1], x[2], x[3],.........x[n]

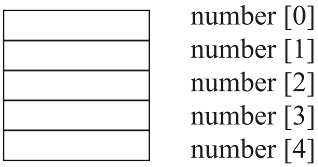
The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35, 40, 20, 57, 19), by an array variable **number**, then we may declare the variable **number** as follows

```
int number[5];
```

and the computer reserves five storage locations as shown below:



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.*

DECLARATION OF ONE-DIMENSIONAL ARRAYS

LO 4.2
Determine how one-dimensional array is declared and initialized

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

type ***variable-name[size];***

The *type* specifies the type of element that will be contained in the array, such as **int, float,** or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember:

- 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*.

WORKED-OUT PROBLEM 4.1 E

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 x1,x2,....are read from the terminal.

Program in Fig. 4.1 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") ;
```

E for Easy, **M** for Medium and **H** for High

```
            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 = %.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. 4.1 *Program to illustrate* ***one-dimensional*** *array*

Note *C99 permits arrays whose size can be specified at run time.*

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

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";
```

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 inilialized 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.

Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
--------
--------
for (i = 0; i < 100; i = i+1)
{
   if   i < 50
         sum[i] = 0.0;           /* assignment statement */
   else
         sum[i] = 1.0;
}
--------
--------
```

The first 50 elements of the array **sum** are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.

We can also use a read function such as **scanf** to initialize an array. For example, the statements

```
int x [3];
scanf("%d%d%d", &x[0], &[1], &x[2]);
```

will initialize array elements with the values entered through the keyboard.

WORKED-OUT PROBLEM 4.2

M

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. 4.2 uses the array **group** containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

Program

```
#define    MAXVAL    50
#define    COUNTER   11
main()
{
      float      value[MAXVAL];
      int           i, low, high;
      int  group[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
      /* . . . . . . . .READING AND COUNTING . . . . . .*/
      for( i = 0 ; i < MAXVAL ; i++ )
      {
      /*. . . . . . . .READING OF VALUES . . . . . . . . */
      scanf("%f", &value[i]) ;
      /*. . . . . .COUNTING FREQUENCY OF GROUPS. . . . . */
         ++ 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. 4.2 *Program for **frequency counting***

Note that we have used an initialization statement.

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

which can be replaced by

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

This will initialize all the elements to zero.

WORKED-OUT PROBLEM 4.3

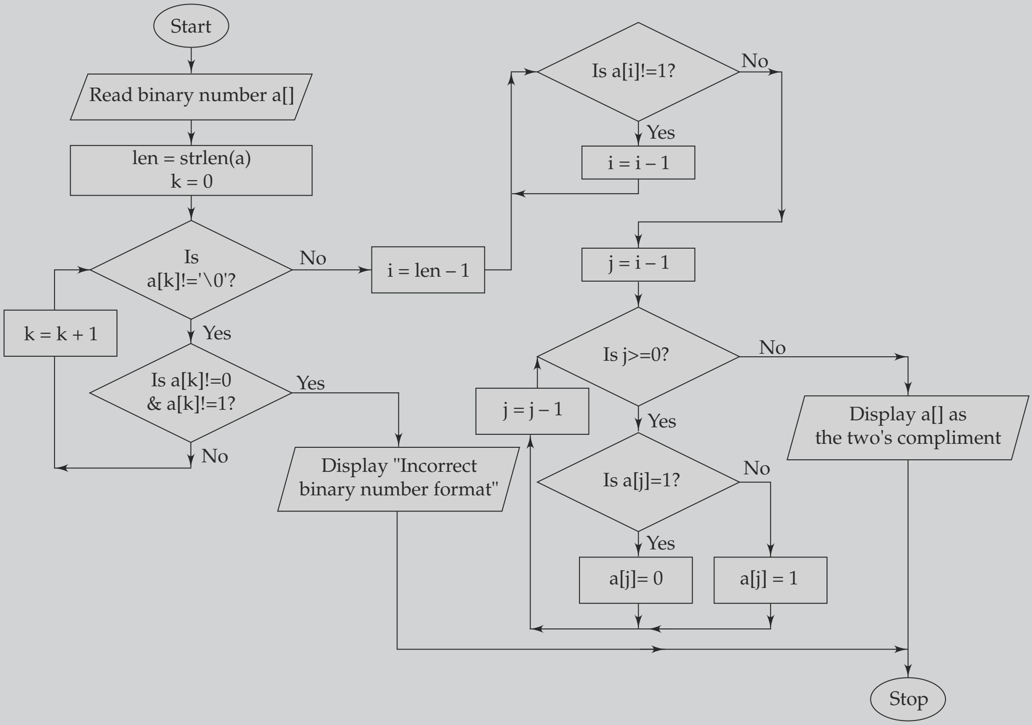
H

The program shown in Fig. 4.3 shows the algorithm, flowchart and the complete C program to find the two's compliment 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'
Step 11 – i = i - 1
Step 12 – Initialize the looping counter j = i - 1
Step 13 – Repeat Step 14-17 while j >= 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. 4.3 *Algorithm, flowchart and C program to find two's compliment of a binary number*

Searching and Sorting

Searching and sorting are the two most frequent operations performed on arrays. Computer Scientists have devised several data structures and searching and sorting techniques that facilitate rapid access to data stored in lists.

Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an *ordered list.* Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available. The three simple and most important among them are:

- Bubble sort
- Selection sort
- Insertion sort

Other sorting techniques include Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the *search key.* If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are:

- Sequential search
- Binary search

TWO-DIMENSIONAL ARRAYS

LO 4.3
Know the concept of 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 $\mathbf{v}_{\mathbf{ij}}$. Here **v** denotes the entire matrix and $\mathbf{v}_{\mathbf{ij}}$ refers to the value in the i^{th} row and j^{th} column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as

v[4][3]

Two-dimensional arrays are declared as follows:

type *array_name* [row_size][column_size];

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

Two-dimensional arrays are stored in memory, as shown in Fig. 4.4. 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.

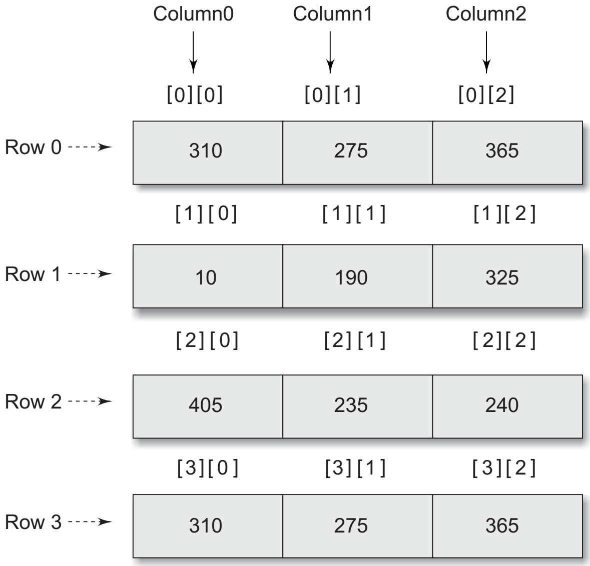


Fig. 4.4 *Representation of a **two-dimensional** array in memory*

WORKED-OUT PROBLEM 4.4

E

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. 4.5 uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested for loops as follows:

```
product[i] [j] = row * column
```

where i denotes rows and j denotes columns of the product table. Since the indices i and j range from 0 to 4, we have introduced the following transformation:

row = i+1
column = j+1

Program

```
#define  ROWS       5
#define  COLUMNS    5
main()
{
      int row, column, product[ROWS][COLUMNS] ;
      int i, j ;
      printf(" MULTIPLICATION TABLE\n\n") ;
      printf(" ") ;
      for( j = 1 ; j <= COLUMNS ; j++ )
         printf("%4d" , j ) ;
      printf("\n") ;
      printf("---------------------------------\n");
      for( i = 0 ; i < ROWS ; i++ )
      {
            row = i + 1 ;
            printf("%2d |", row) ;
            for( j = 1 ; j <= COLUMNS ; j++ )
            {
                  column = j ;
                  product[i][j] = row * column ;
                  printf("%4d", product[i][j] ) ;
                  }
                  printf("\n") ;
   }
}
```

Output

```
      MULTIPLICATION TABLE
      1     2     3     4     5
------------------------------------
1  |  1     2     3      4      5
2  |  2     4     6      8     10
3  |  3     6     9     12     15
4  |  4     8    12     16     20
5  |  5    10    15     20     25
```

Fig. 4.5 *Program to print multiplication table using two-dimensional array*

WORKED-OUT PROBLEM 4.5

H

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. 4.6. 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) Total sales by mth girl = $\sum_{i=0}^{2}$ value [m][j] (girl_total[m])

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

(c) Grand total = $\sum_{i=0}^{3} \sum_{i=0}^{2}$ value[i][j]

$$= \sum_{i=0}^{3} \text{girl\_total}[i]$$

$$= \sum_{j=0}^{2} \text{item\_total}[j]$$

Program

```
#define  MAXGIRLS   4
#define  MAXITEMS   3
main()
{
   int  value[MAXGIRLS][MAXITEMS];
   int  girl_total[MAXGIRLS] , item_total[MAXITEMS];
   int  i, j, grand_total;
/*.......READING OF VALUES AND COMPUTING girl_total ...*/

   printf("Input data\n");
   printf("Enter values, one at a time, row-wise\n\n");

   for( i = 0 ; i < MAXGIRLS ; i++ )
   {
        girl_total[i] = 0;
        for( j = 0 ; j < MAXITEMS ; j++ )
        {
             scanf("%d", &value[i][j]);
             girl_total[i] = girl_total[i] + value[i][j];
        }
   }
/*.......COMPUTING item_total.........................*/

   for( j = 0 ; j < MAXITEMS ; j++ )
   {
        item_total[j] = 0;
        for( i =0 ; i < MAXGIRLS ; i++ )
             item_total[j] = item_total[j] + value[i][j];
   }
/*.......COMPUTING grand_total........................*/
   grand_total = 0;
   for( i =0 ; i < MAXGIRLS ; i++ )
     grand_total = grand_total + girl_total[i];
/* .......PRINTING OF RESULTS..........................*/

   printf("\n GIRLS TOTALS\n\n");

   for( i = 0 ; i < MAXGIRLS ; i++ )
        printf("Salesgirl[%d] = %d\n", i+1, girl_total[i] );
   printf("\n ITEM TOTALS\n\n");
   for( j = 0 ; j < MAXITEMS ; j++ )
        printf("Item[%d] = %d\n", j+1 , item_total[j] );
   printf("\nGrand Total = %d\n", grand_total);
}
```

Output

```
Input data
Enter values, one at a time, row_wise
310 257 365
210 190 325
405 235 240
260 300 380
GIRLS TOTALS
Salesgirl[1] = 950
Salesgirl[2] = 725
Salesgirl[3] = 880
Salesgirl[4] = 940
ITEM TOTALS
```

```
Item[1] = 1185
Item[2] = 1000
Item[3] = 1310
Grand Total = 3495
```

Fig. 4.6 *Illustration of two-dimensional arrays*

INITIALIZING TWO-DIMENSIONAL ARRAYS

LO 4.4
Discuss how two-dimensional array is declared and initialized

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};
```

WORKED-OUT PROBLEM 4.6

M

A survey to know the popularity of four cars (Ambassador, Fiat, Dolphin and Maruti) was conducted in four cities (Bombay, Calcutta, Delhi and Madras). Each person surveyed was asked to give his city and the type of car he was using. The results, in coded form, are tabulated as follows:

M	1	C	2	B	1	D	3	M	2	B	4
C	1	D	3	M	4	B	2	D	1	C	3
D	4	D	4	M	1	M	1	B	3	B	3
C	1	C	1	C	2	M	4	M	4	C	2
D	1	C	2	B	3	M	1	B	1	C	2
D	3	M	4	C	1	D	2	M	3	B	4

Codes represent the following information:

M – Madras	1 – Ambassador
D – Delhi	2 – Fiat
C – Calcutta	3 – Dolphin
B – Bombay	4 – Maruti

Write a program to produce a table showing popularity of various cars in four cities.

A two-dimensional array **frequency** is used as an accumulator to store the number of cars used, under various categories in each city. For example, the element **frequency** [i][j] denotes the number of cars of type j used in city i. The **frequency** is declared as an array of size 5 × 5 and all the elements are initialized to zero.

The program shown in Fig. 4.7 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

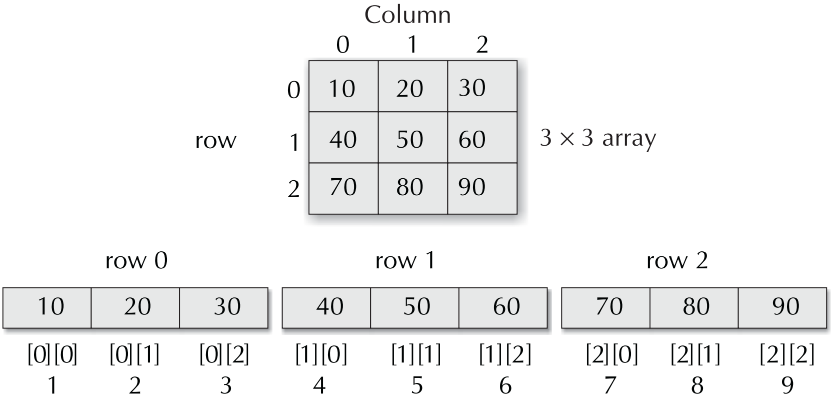
```
For each person, enter the city code
followed by the car code.
Enter the letter X to indicate end.
M 1  C  2  B  1  D  3  M  2  B  4
C 1  D  3  M  4  B  2  D  1  C  3
D 4  D  4  M  1  M  1  B  3  B  3
C 1  C  1  C  2  M  4  M  4  C  2
D 1  C  2  B  3  M  1  B  1  C  2
D 3  M  4  C  1  D  2  M  3  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. 4.7 *Program to tabulate a survey data*

Memory Layout

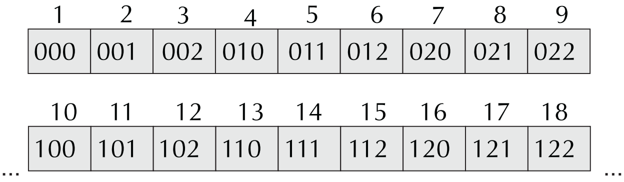
The subscripts in the definition of a two-dimensional array represent rows and columns. This format maps the way that data elements are laid out in the memory. The elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we consider the memory as a row of bytes, with the lowest address on the left and the highest address on the right, a simple array will be stored in memory with the first element at the left end and the last element at the right end. Similarly, a two-dimensional array is stored “row-wise, starting from the first row and ending with the last row, treating each row like a simple array. This is illustrated below.



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



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.

WORKED-OUT PROBLEM 4.7

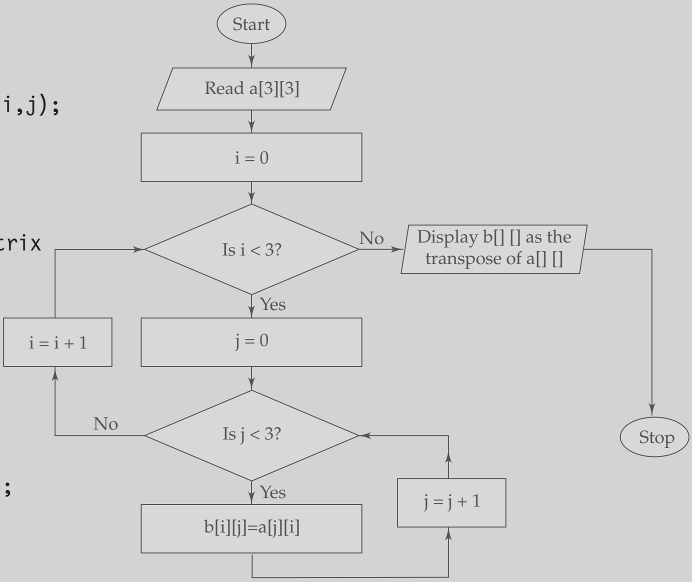
E

The program in Fig. 4.8 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
```

Flowchart



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
is: \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
     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. 4.8 *Program to find transpose of a matrix*

WORKED-OUT PROBLEM 4.8

M

The program in Fig. 4.9 shows how to multiply the elements of two N × 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. 4.9 *Program for N × N matrix multiplication*

MULTI-DIMENSIONAL ARRAYS

LO 4.5
Describe 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 ith 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.

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.

LO 4.6
Explain dynamic arrays

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. We discuss in detail pointers and pointer variables in Chapter 6 and creating and managing linked lists in Chapter 8.

MORE ABOUT ARRAYS

What we have discussed in this chapter are the basic concepts of arrays and their applications to a limited extent. There are some more important aspects of application of arrays. They include:

- using printers for accessing arrays;
- passing arrays as function parameters;
- arrays as members of structures;
- using structure type data as array elements;
- arrays as dynamic data structures; and
- manipulating character arrays and strings.

These aspects of arrays are covered later in the following chapters:

Chapter 5	:	Functions
Chapter 7	:	Structures
Chapter 6	:	Pointers
Chapter 8	:	Linked Lists

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:

LO 4.7
Discuss how string variables are declared and initialized

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.

READING STRINGS FROM TERMINAL

LO 4.8
Explain how strings are read from terminal

Using scanf Function

The familiar input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[10]
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal:

NEW YORK

then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the reading of string.

The **scanf** function automatically terminates the string that is read with a null character and therefore, the character array should be large enough to hold the input string plus the null character. Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

The **address** array is created in the memory as shown below:

N	E	W	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Note that the unused locations are filled with garbage.

If we want to read the entire line “NEW YORK”, then we may use two character arrays of appropriate sizes. That is,

```
char adr1[5], adr2[5];
scanf("%s %s", adr1, adr2);
```

with the line of text

NEW YORK

will assign the string “NEW” to **adr1** and “YORK” to **adr2**.

WORKED-OUT PROBLEM 4.9

Write a program to read a series of words from a terminal using scanf function.

The program shown in Fig. 4.10 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. 4.10 *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

Reading a Line of Text

We have seen just now that **scanf** with %**s** or %**ws** can read only strings without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the *edit set conversion code* %[. .] that can be used to read a line containing a variety of characters, including whitespaces. Recall that we have used this conversion code in Chapter 4. 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.

Using getchar and gets Functions

A single character can be read from the terminal, using the function **getchar**. We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array. The reading is terminated when the newline character ('\n') is entered and the null character is then inserted at the end of the string. The **getchar** function call takes the following form:

```
char ch;
ch = getchar( );
```

*Note that the **getchar** function has no parameters.*

WORKED-OUT PROBLEM 4.10

M

Write a program to read a line of text containing a series of words from the terminal.

The program shown in Fig. 4.11 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. 4.11 *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.

WORKED-OUT PROBLEM 4.11

M

Write a program to copy one string into another and count the number of characters copied.

The program is shown in Fig. 4.12. We use a **for** loop to copy the characters contained inside **string2** into the **string1**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **string1.**

Program

```
main( )
{
     char string1[80], string2[80];
     int i;
     printf("Enter a string \n");
     printf("?");
     scanf("%s", string2);
     for( i=0 ; string2[i] != '\0'; i++)
          string1[i] = string2[i];
     string1[i] = '\0';
     printf("\n");
     printf("%s\n", string1);
     printf("Number of characters = %d\n", i );
}
```

Output

```
Enter a string
?Manchester

Manchester
Number of characters = 10

Enter a string
?Westminster

Westminster
Number of characters = 11
```

Fig. 4.12 *Copying one string into another*

WORKED-OUT PROBLEM 4.12

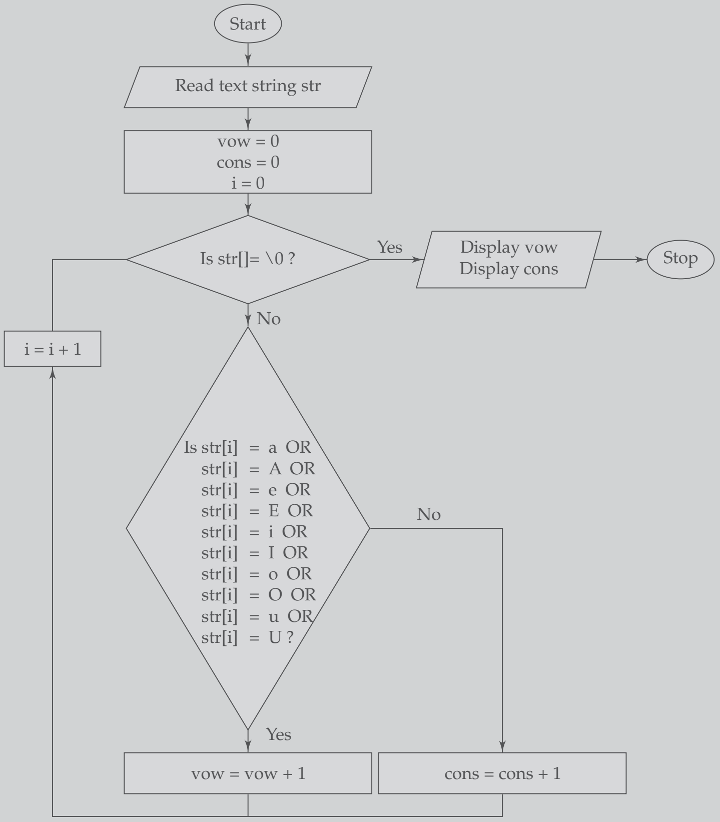
H

The program in Fig. 4.13 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++;
       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. 4.13 *Program to find the number of vowel and consonants in a text string*

WRITING STRINGS TO SCREEN

LO 4.9
Describe how strings are written to screen

Using printf Function

We have used extensively the **printf** function with %s format to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character. For example, the statement

```
printf("%s", name);
```

can be used to display the entire contents of the array **name**.

We can also specify the precision with which the array is displayed. For instance, the specification

```
%10.4
```

indicates that the *first four* characters are to be printed in a field width of 10 columns.

However, if we include the minus sign in the specification (e.g., %-10.4s), the string will be printed left-justified. The Program 4.4 illustrates the effect of various %s specifications.

WORKED-OUT PROBLEM 4.13

M

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. 4.14. 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("%15.0s\n", country);
   printf("%.3s\n", country);
   printf("%s\n", country);
   printf("————— \n");
}
```

Output

```
*123456789012345*
-----
United Kingdom
United Kingdom
        United
United
Uni
United Kingdom
-----
```

Fig. 4.14 *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. Program 4.5 illustrates this.

WORKED-OUT PROBLEM 4.14

M

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. 4.15, for variable specifications **%12.*s, %.*s,** and **%*.1s** are shown in Fig. 4.16, 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. 4.15 *Illustration of variable field specifications by printing sequences of characters*

```
           C
          CP
         CPr
        CPro
       CProg
      CProgr
     CProgra
    CProgram
   CProgramm
  CProgrammi
 CProgrammin
CProgramming
CProgramming
 CProgrammin
  CProgrammi
   CProgramm
    CProgram
     CProgra
      CProgr
       CProg
        CPro
         CPr
          CP
           C

  (a) %12.*s
```

```
C|
CP|
CPr|
CPro|
CProg|
CProgr|
CProgra|
CProgram|
CProgramm|
CProgrammi|
CProgrammin|
CProgramming|
CProgramming|
CProgrammin|
CProgrammi|
CProgramm|
CProgram|
CProgra|
CProgr|
CProg|
CPro|
CPr|
CP|
C|

    (b) %.*s
```

```
 C|
  C|
   C|
    C|
     C|
      C|
       C|
        C|
         C|
          C|
            C|
            C|
           C|
          C|
         C|
        C|
       C|
      C|
     C|
    C|
   C|
  C|
 C|
C|

   (c) %*.1s
```

Fig. 4.16 *Further illustrations of variable specifications*

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);
```

putchar function is used 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.

ARITHMETIC OPERATIONS ON CHARACTERS

LO 4.10
Illustrate how strings are manipulated

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>.

WORKED-OUT PROBLEM 4.15

E

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. 4.17. 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. 4.17 *Printing of the alphabet set in decimal and character form*

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*. Program 4.18 illustrates the concatenation of three strings.

WORKED-OUT PROBLEM 4.16

M

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. 4.18. 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. 4.18 *Concatenation of strings*

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");
```

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.

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:

	0	1	2	3	4	5	6	7	8	9	0	1
Part1 =	V	E	R	Y		\0						

	0	1	2	3	4	5	6
Part2 =	G	O	O	D	\0		

	0	1	2	3	4	5	6
Part3 =	B	A	D	\0			

Execution of the statement

```
strcat(part1, part2);
```

will result in:

	0	1	2	3	4	5	6	7	8	9	0	1	2
Part1 =	V	E	R	Y		G	O	O	D	\0			

	0	1	2	3	4	5	6
Part2 =	G	O	O	D	\0		

while the statement

will result in:

	0	1	2	3	4	5	6	7	8	9	0	1	2
Part1 =	V	E	R	Y		B	A	D	\0				

	0	1	2	3	4	5	6
Part3 =	B	A	D	\0			

We must make sure that the size of **string1** (to which **string2** is appended) is large enough to accommodate the final string.

strcat function may also append a string constant to a string variable. The following is valid:

```
strcat(part1,"GOOD");
```

C permits nesting of **strcat** functions. For example, the statement

```
strcat(strcat(string1,string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in **string1.**

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**.

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.**

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.

WORKED-OUT PROBLEM 4.17

M

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. 4.19. 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. 4.19 *Illustration of string handling functions*

WORKED-OUT PROBLEM 4.18

M

The program in Fig. 4.20 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. 4.20 *Program to check if a string is palindrome or not*

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:

S1 :	B	A	L	A	\0							

S2 :	G	U	R	U	S	A	M	Y	\0

After **strncat** (s1, s2, 4); execution:

S1 :	B	A	L	A	G	U	R	U	\0

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');
```

will locate the first occurrence of the character ‘m’ and the call

```
strrchr(s1, 'm');
```

will locate the last occurrence of the character ‘m’ in the string **s1**.

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.

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 ith 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.

WORKED-OUT PROBLEM 4.19

H

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. 4.21. 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. 4.21 *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.

OTHER FEATURES OF STRINGS

Other aspects of strings we have not discussed in this chapter include the following:

- Manipulating strings using pointers.
- Using string as function parameters.
- Declaring and defining strings as members of structures.

These topics will be dealt with later when we discuss functions, structures, and pointers.

KEY CONCEPTS

- **ARRAY:** Is a fixed-size sequenced collection part of elements of the same data type. **[LO 4.1]**
- **ONE-DIMENSIONAL ARRAY:** Is a list of items that has one variable name and one subscript to access the items. **[LO 4.1]**
- **STRUCTURED DATA TYPES:** Represent data values that have a structure of some sort. For example, arrays, structures, etc. **[LO 4.1]**
- **SEARCHING:** Is the process of finding the location of the specified element in the list. **[LO 4.2]**
- **SORTING:** Is the process of rearranging elements in the list as per ascending or descending order. **[LO 4.2]**
- **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. **[LO 4.3]**
- **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. **[LO 4.5]**
- **DYNAMIC ARRAYS:** Are the arrays declared using dynamic memory allocation technique. **[LO 4.6]**
- **DYNAMIC MEMORY ALLOCATION:** Is the process of allocating memory at run time. **[LO 4.6]**
- **STATIC ARRAYS:** Are the arrays declared using static memory allocation technique. **[LO 4.6]**
- **STATIC MEMORY ALLOCATION:** Is the process of allocating memory at compile time. **[LO 4.6]**
- **STRING:** Is a sequence of characters that is considered as a single data item. **[LO 4.7]**
- **STRCAT:** Concatenates two strings. **[LO 4.10]**
- **STRCMP:** Compares two strings and determines whether they are equal or not. **[LO 4.10]**
- **STRCPY:** Copies one string into another. **[LO 4.10]**
- **STRSTR:** Determines whether one string is a subset of another. **[LO 4.10]**

ALWAYS REMEMBER

- We need to specify three things, namely, name, type and size, when we declare an array. **[LO 4.1]**
- Use of invalid subscript is one of the common errors. An incorrect or invalid index may cause unexpected results. **[LO 4.1]**
- Always remember that subscripts begin at 0 (not 1) and end at size –1. **[LO 4.2]**
- Defining the size of an array as a symbolic constant makes a program more scalable. **[LO 4.2]**
- 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. **[LO 4.2]**
- Do not forget to initialize the elements; otherwise they will contain "garbage". **[LO 4.2]**
- Supplying more initializers in the initializer list is a compile time error. **[LO 4.2]**
- 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. **[LO 4.2]**
- 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+ +)
  ```
  [LO 4.2]
- Referring a two-dimensional array element like x[i, j] instead of x[i][j] is a compile time error. **[LO 4.3]**
- Leaving out the subscript reference operator [ ] in an assignment operation is compile time error. **[LO 4.3]**
- When initializing character arrays, provide enough space for the terminating null character. **[LO 4.4]**
- Make sure that the subscript variables have been properly initialized before they are used. **[LO 4.4]**
- During initialization of multi-dimensional arrays, it is an error to omit the array size for any dimension other than the first. **[LO 4.5]**
- 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. **[LO 4.6]**
- Character constants are enclosed in single quotes and string constants are enclosed in double quotes. **[LO 4.7]**
- Allocate sufficient space in a character array to hold the null character at the end. **[LO 4.7]**
- Avoid processing single characters as strings. **[LO 4.7]**
- It is a compile time error to assign a string to a character variable. **[LO 4.7]**
- The header file <stdio.h> is required when using standard I/O functions. **[LO 4.7]**
- The header file <stdlib.h> is required when using general utility functions. **[LO 4.7]**
- Using the address operator **&** with a **string** variable in the **scanf** function call is an error. **[LO 4.8]**
- Use %s format for printing strings or character arrays terminated by null character. **[LO 4.9]**
- Using a string variable name on the left of the assignment operator is illegal. **[LO 4.10]**
- When accessing individual characters in a string variable, it is logical error to access outside the array bounds. **[LO 4.10]**
- Strings cannot be manipulated with operators. Use string functions. **[LO 4.10]**
- Do not use string functions on an array **char** type that is not terminated with the null character. **[LO 4.10]**
- 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. **[LO 4.10]**
- Be aware the return values when using the functions **strcmp** and **strncmp** for comparing strings. **[LO 4.10]**

- 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. **[LO 4.10]**
- The header file <ctype.h> is required when using character handling functions. **[LO 4.10]**
- The header file <string.h> is required when using string manipulation functions. **[LO 4.10]**

BRIEF CASES

1. Median of a List of Numbers [LO 4.2, M]

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.

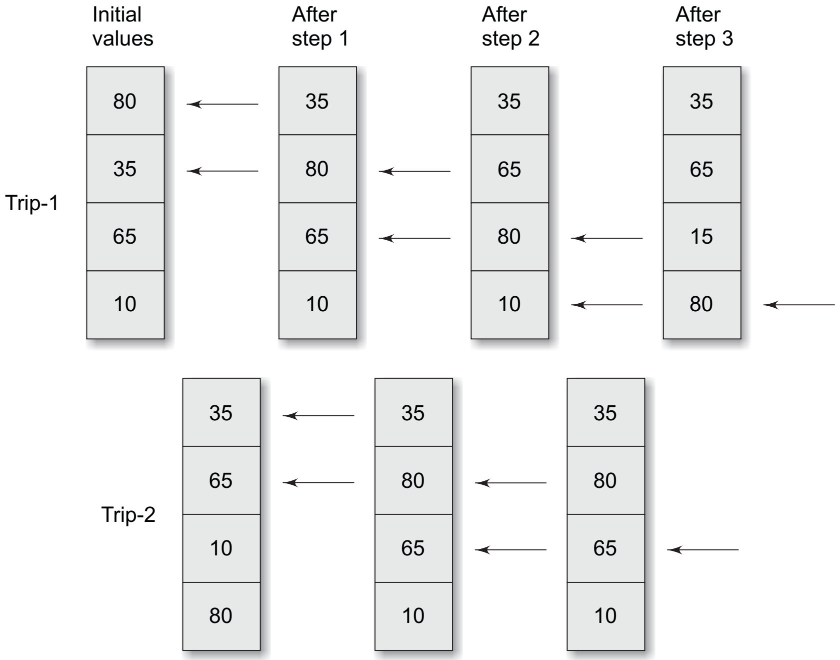
The major steps for finding the median are as follows:

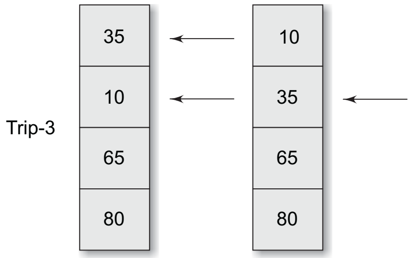
1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

The program and sample output are shown in Fig. 4.22. The sorting algorithm used is as follows:

1. Compare the first two elements in the list, say a[1], and a[2]. If a[2] is smaller than a[1], then interchange their values.
2. Compare a[2] and a[3]; interchange them if a[3] is smaller than a[2].
3. Continue this process till the last two elements are compared and interchanged.
4. Repeat the above steps n–1 times.

In repeated trips through the array, the smallest elements ‘bubble up’ to the top. Because of this bubbling up effect, this algorithm is called *bubble sorting*. The bubbling effect is illustrated below for four items.





During the first trip, three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered any further. Therefore, trip-2 requires only two pairs to be compared. This time, the number 65 (the second largest value) has been moved down the list. Notice that each trip brings the smallest value 10 up by one level.

The number of steps required in a trip is reduced by one for each trip made. The entire process will be over when a trip contains only one step. If the list contains **n** elements, then the number of comparisons involved would be **n(n–1)/2.**

Program

```
#define N 10
main( )
{
   int i,j,n;
   float median,a[N],t;
   printf("Enter the number of items\n");
   scanf("%d", &n);
/* Reading items into array a */
   printf("Input %d values \n",n);
   for (i = 1; i <= n ; i++)
      scanf("%f", &a[i]);
/* Sorting begins */
   for (i = 1 ; i <= n–1 ; i++)
   {    /* Trip-i begins */
      for (j = 1 ; j <= n–i ; j++)
      {
            if (a[j] <= a[j+1])
            { /* Interchanging values */
               t = a[j];
               a[j] = a[j+1];
               a[j+1] = t;
            }
            else
            continue ;
      }
   } /* sorting ends */
/* calculation of median */
   if ( n % 2 == 0)
      median = (a[n/2] + a[n/2+1])/2.0 ;
   else
      median = a[n/2 + 1];
/* Printing */
   for (i = 1 ; i <= n ; i++)
         printf("%f ", a[i]);
   printf("\n\nMedian is %f\n", median);
}
```

Output

```
Enter the number of items
5
Input 5 values
1.111 2.222 3.333 4.444 5.555
5.555000 4.444000 3.333000 2.222000 1.111000
Median is 3.333000
Enter the number of items
6
Input 6 values
3 5 8 9 4 6
9.000000 8.000000 6.000000 5.000000 4.000000 3.000000
Median is 5.500000
```

Fig. 4.22 *Program to sort a list of numbers and to determine median*

2. Calculation of Standard Deviation [LO 4.2, M]

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. 4.23.

Program

```
#include <math.h>
#define MAXSIZE 100
main( )
{
     int i,n;
     float value [MAXSIZE], deviation,
          sum,sumsqr,mean,variance,stddeviation;
     sum = sumsqr = n = 0 ;
     printf("Input values: input –1 to end \n");
     for (i=1; i< MAXSIZE ; i++)
     {
        scanf("%f", &value[i]);
        if (value[i] == -1)
          break;
        sum += value[i];
        n += 1;
     }
     mean = sum/(float)n;
     for (i = 1 ; i<= n; i++)
     {
        deviation = value[i] – mean;
        sumsqr += deviation * deviation;
     }
     variance = sumsqr/(float)n ;
     stddeviation = sqrt(variance) ;
     printf("\nNumber of items : %d\n",n);
     printf("Mean : %f\n", mean);
     printf("Standard deviation : %f\n", stddeviation);
}
```

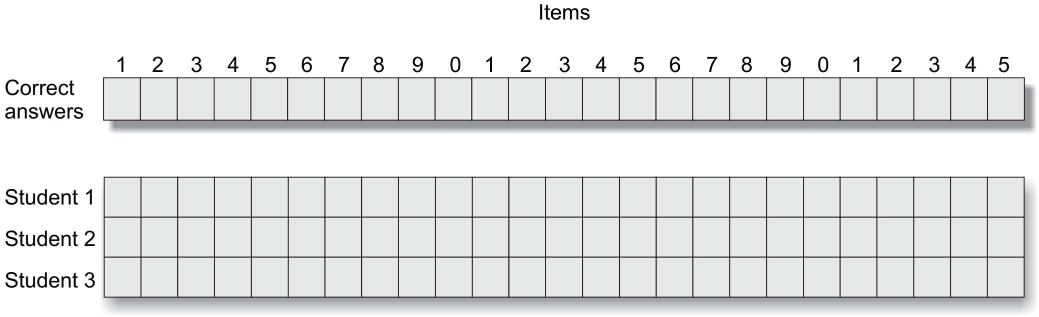
Output

```
Input values: input -1 to end
65 9 27 78 12 20 33 49 -1
Number of items : 8

Mean : 36.625000
Standard deviation : 23.510303
```

Fig. 4.23 *Program to calculate standard deviation*

3. Evaluating a Test [LO 4.2, H]

A test consisting of 25 multiple-choice items is administered to a batch of 3 students. Correct answers and student responses are tabulated as shown below:



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. 4.24. 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
abcdabcdabcdabcdabcdabcda
Input responses of student-1
abcdabcdabcdabcdabcdabcda
Student-1
Score is 25 out of 25
Response to the following items are wrong
NIL
Input responses of student-2
```

```
abcddcbaabcdabcdddddddddd
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
aaaaaaaaaaaaaaaaaaaaaaaaa
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. 4.24 *Program to evaluate responses to a multiple-choice test*

4. Production and Sales Analysis [LO 4.3, 4.4, H]

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,

	M11	M12	M13	M14	M15
M =	M21	M22	M23	M24	M25
	M31	M32	M33	M34	M35
	M41	M42	M43	M44	M45

	S11	S12	S13	S14	S15
S =	S21	S22	S23	S24	S25
	S31	S32	S33	S34	S35
	S41	S42	S43	S44	S45

where Mij represents the number of jth type product manufactured in ith week and Sij the number of jth product sold in ith 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 Cj is the cost of jth 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 x Cj
Svalue[i][j] = Sij x Cj

A program to generate the required outputs for the review meeting is shown in Fig. 4.25. The following additional variables are used:

Mweek[i] = Value of all the products manufactured in week i

$$= \sum_{J=1}^{5} \text{Mvalue[i][j]}$$

Sweek[i] = Value of all the products in week i

$$= \sum_{J=1}^{5} \text{Svalue[i][j]}$$

Mproduct[j] = Value of jth type product manufactured during the month

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

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

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

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

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

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

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

Program

```
main( )
{
   int M[5][6],S[5][6],C[6],
      Mvalue[5][6],Svalue[5][6],
      Mweek[5], Sweek[5],
      Mproduct[6], Sproduct[6],
      Mtotal, Stotal, i,j,number;
/*   Input data      */
printf (" Enter products manufactured week_wise \n");
printf (" M11,M12,—, M21,M22,— etc\n");
for(i=1; i<=4; i++)
   for(j=1;j<=5; j++)
      scanf("%d",&M[i][j]);
printf (" Enter products sold week_wise\n");
printf (" S11,S12,—, S21,S22,— etc\n");
```

```
for(i=1; i<=4; i++)
   for(j=1; j<=5; j++)
      scanf("%d", &S[i][j]);
printf(" Enter cost of each product\n");
   for(j=1; j <=5; j++)
      scanf("%d",&C[j]);
/* Value matrices of production and sales */
   for(i=1; i<=4; i++)
      for(j=1; j<=5; j++)
      {
           Mvalue[i][j] = M[i][j] * C[j];
           Svalue[i][j] = S[i][j] * C[j];
      }
/* Total value of weekly production and sales */
   for(i=1; i<=4; i++)
   {
      Mweek[i] = 0 ;
      Sweek[i] = 0 ;
      for(j=1; j<=5; j++)
   {
      Mweek[i] += Mvalue[i][j];
      Sweek[i] += Svalue[i][j];
   }
}
/* Monthly value of product_wise production and sales */
   for(j=1; j<=5; j++)
   {
        Mproduct[j] = 0 ;
        Sproduct[j] = 0 ;
        for(i=1; i<=4; i++)
        {
           Mproduct[j] += Mvalue[i][j];
           Sproduct[j] += Svalue[i][j];
        }
   }
```

```
/* Grand total of production and sales values */
  Mtotal = Stotal = 0;
  for(i=1; i<=4; i++)
  {
    Mtotal += Mweek[i];
    Stotal += Sweek[i];
  }
  /**********************************************
  Selection and printing of information required
  **********************************************/
  printf("\n\n");
  printf(" Following is the list of things you can\n");
  printf(" request for. Enter appropriate item number\n");
  printf(" and press RETURN Key\n\n");
  printf(" 1.Value matrices of production & sales\n");
  printf(" 2.Total value of weekly production & sales\n");
  printf(" 3.Product_wise monthly value of production &");
  printf(" sales\n");
  printf(" 4.Grand total value of production & sales\n");
  printf(" 5.Exit\n");
  number = 0;
  while(1)
  {    /* Beginning of while loop */
    printf("\n\n ENTER YOUR CHOICE:");
    scanf("%d",&number);
    printf("\n");
    if(number == 5)
    {
    printf(" GOOD  BYE\n\n");
    break;
  }
    switch(number)
  {  /* Beginning of switch */
    /* VALUE  MATRICES */
      case 1:
      printf(" VALUE MATRIX OF PRODUCTION\n\n");
      for(i=1; i<=4; i++)
      {
        printf(" Week(%d)\t",i);
        for(j=1; j <=5; j++)
          printf("%7d", Mvalue[i][j]);
        printf("\n");
      }
      printf("\n VALUE MATRIX OF SALES\n\n");
      for(i=1; i <=4; i++)
      {
      printf(" Week(%d)\t",i);
      for(j=1; j <=5; j++)
          printf("%7d", Svalue[i][j]);
        printf("\n");
      }
        break;
      /* WEEKLY ANALYSIS */
        case 2:
          printf(" TOTAL WEEKLY   PRODUCTION & SALES\n\n");
          printf("                PRODUCTION   SALES\n");
          printf("                -----        --    \n");
          for(i=1; i <=4; i++)
          {
            printf(" Week(%d)\t", i);
            printf("%7d\t%7d\n", Mweek[i], Sweek[i]);
          }
          break;
      /* PRODUCT WISE ANALYSIS */
        case 3:
          printf(" PRODUCT_WISE TOTAL PRODUCTION &");
          printf(" SALES\n\n");
          printf("                   PRODUCTION SALES\n");
          printf("                   -----        --    \n");
          for(j=1; j <=5; j++)
          {
            printf(" Product(%d)\t", j);
            printf("%7d\t%7d\n",Mproduct[j],Sproduct[j]);
          }
          break;
      /* GRAND TOTALS */
        case 4:
          printf(" GRAND TOTAL OF PRODUCTION & SALES\n");
          printf("\n Total production = %d\n", Mtotal);
          printf(" Total sales = %d\n", Stotal);
          break;
      /* DEFAULT */
        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. 4.25 *Program for production and sales analysis*

5. Counting Words in a Text [LO 4.7, 4.8 M]

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. 4.26. 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. 4.26 *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.

6. Processing of a Customer List [LO 4.7, 4.8, 4.9, 4.10 M]

Telephone numbers of important customers are recorded as follows:

Full name	Telephone number
Joseph Louis Lagrange	869245
Jean Robert Argand	900823
Carl Freidrich Gauss	806788
-----	-----
-----	-----

It is desired to prepare a revised alphabetical list with surname (last name) first, followed by a comma and the initials of the first and middle names. For example,

Argand, J.R

We create a table of strings, each row representing the details of one person, such as first_name, middle_name, last_name, and telephone_number. The columns are interchanged as required and the list is sorted on the last_name. Figure 4.27 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. 4.27 *Program to alphabetize a customer list*

REVIEW QUESTIONS

4.1 State whether the following statements are *true* or *false*.

(a) An array can store infinite data of similar type. **[LO 4.1 E]**

(b) In declaring an array, the array size can be a constant or variable or an expression. **[LO 4.2 E]**

(c) The declaration int x[2] = {1,2,3}; is illegal. **[LO 4.2 E]**

(d) When an array is declared, C automatically initializes its elements to zero. **[LO 4.2, 4.5 M]**

(e) An expression that evaluates to an integral value may be used as a subscript. **[LO 4.1, 4.2 M]**

(f) In C, by default, the first subscript is zero. **[LO 4.1, 4.2 M]**

(g) When initializing a multidimensional array, not specifying all its dimensions is an error. **[LO 4.1, 4.2 M]**

(h) When we use expressions as a subscript, its result should be always greater than zero. **[LO 4.1, 4.2 M]**

(i) In C, we can use a maximum of 4 dimensions for an array. **[LO 4.1, 4.2 M]**

(j) Accessing an array outside its range is a compile time error. **[LO 4.2 H]**

(k) A **char** type variable cannot be used as a subscript in an array. **[LO 4.2 H]**

(l) An unsigned long int type can be used as a subscript in an array. **[LO 4.2 H]**

(m) When initializing a string variable during its declaration, we must include the null character as part of the string constant, like “GOOD\0”. **[LO 4.7 M]**

(n) The **gets** function automatically appends the null character at the end of the string read from the keyboard. **[LO 4.8 E]**

(o) When reading a string with **scanf**, it automatically inserts the terminating null character. **[LO 4.8 E]**

(p) The input function **gets** has one string parameter. **[LO 4.8 E]**

(q) The function **scanf** cannot be used in any way to read a line of text with the white-spaces. **[LO 4.8 M]**

(r) The function **getchar** skips white-space during input. **[LO 4.8 M]**

(s) In C, strings cannot be initialized at run time. **[LO 4.8 H]**

(t) String variables cannot be used with the assignment operator. **[LO 4.10 E]**

(u) We cannot perform arithmetic operations on character variables. **[LO 4.10 E]**

(v) The ASCII character set consists of 128 distinct characters. **[LO 4.10 E]**

(w) In the ASCII collating sequence, the uppercase letters precede lowercase letters. **[LO 4.10 M]**

(x) In C, it is illegal to mix character data with numeric data in arithmetic operations. **[LO 4.10 M]**

(y) The function call **strcpy(s2, s1);** copies string s2 into string s1. **[LO 4.10 M]**

(z) The function call **strcmp(“abc”, “ABC”);** returns a positive number. **[LO 4.10 M]**

(aA) We can assign a character constant or a character variable to an **int** type variable. **[LO 4.10 H]**

4.2 Fill in the blanks in the following statements.

(a) The variable used as a subscript in an array is popularly known as _________ variable. **[LO 4.1 E]**

(b) An array that uses more than two subscript is referred to as ______ array. **[LO 4.2 E]**

(c) _______ is the process of arranging the elements of an array in order. **[LO 4.3 E]**

(d) An array can be initialized either at compile time or at ________. **[LO 4.6 M]**

(e) An array created using **malloc** function at run time is referred to as ______ array. **[LO 4.6 M]**

(f) We can use the conversion specification _______in **scanf** to read a line of text. **[LO 4.8 E]**

(g) The function _______does not require any conversion specification to read a string from the keyboard. **[LO 4.8 E]**

(h) The **printf** may be replaced by ______function for printing strings. **[LO 4.9 E]**

(i) The function **strncat** has _____ parameters. **[LO 4.10 E]**

(j) The function _______ is used to determine the length of a string. **[LO 4.10 E]**

(k) We can initialize a string using the string manipulation function_______. **[LO 4.8 M]**

(l) To use the function **atoi** in a program, we must include the header file ____. **[LO 4.10 M]**

(m) The _________string manipulation function determines if a character is contained in a string. **[LO 4.10 M]**

(n) The function call **strcat (s2, s1);** appends _____ to ______. **[LO 4.10 M]**

(o) The function _____is used to sort the strings in alphabetical order. **[LO 4.10 H]**

4.3 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. **[LO 4.4 M]**

1	0	0	0	0	. . . . .	0
0	1	0	0	0	. . . . .	0
0	0	1	0	0	. . . . .	0
.	.	.	.	.		.

.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
0	0	0	0	0	. . . . .	1

4.4 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? **[LO 4.4 M]**

(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];`

4.5 Which of the following initialization statements are correct? **[LO 4.2 H]**

(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";`

4.6 What is a data structure? Why is an array called a data structure? **[LO 4.1 E]**

4.7 What is a dynamic array? How is it created? Give a typical example of use of a dynamic array. **[LO 4.6 M]**

4.8 What happens when an array with a specified size is assigned **[LO 4.2 H]**

(a) with values fewer than the specified size; and
(b) with values more than the specified size.

4.9 Discuss how initial values can be assigned to a multidimensional array. **[LO 4.5 E]**

4.10 Describe the limitations of using **getchar** and **scanf** functions for reading strings. **[LO 4.8 M]**

4.11 Character strings in C are automatically terminated by the *null* character. Explain how this feature helps in string manipulations. **[LO 4.10 H]**

4.12 Strings can be assigned values as follows: **[LO 4.7, 4.8, 4.10 E]**

(a) During type declaration	char string[ ] = {“.......”};
(b) Using **strcpy** function	strcpy(string, “.......”);
(c) Reading using **scanf** function	scanf(“%s”, string);
(d) Reading using **gets** function	gets(string);

Compare them critically and describe situations where one is superior to the others.

4.13 Assuming the variable **string** contains the value “The sky is the limit.”, determine what output of the following program segments will be. **[LO 4.9 H]**

(a) printf(“%s”, string);
(b) printf(“%25.10s”, string);
(c) printf(“%s”, string[0]);
(d) for (i=0; string[i] != “.”; i++)
        printf(“%c”, string[i]);
(e) for (i=0; string[i] != ‘\0’; i++;)
    printf(“%d\n”, string[i]);

(f)
```
for (i=0; i <= strlen[string]; ;)
{
            string[i++] = i;
    printf(“%s\n”, string[i]);
}
```

(g) printf(“%c\n”, string[10] + 5);

(h) printf(“%c\n”, string[10] + 5’)

4.14 Which of the following statements will correctly store the concatenation of strings **s1** and **s2** in string **s3**? **[LO 4.10 M]**

(a) s3 = strcat (s1, s2);

(b) strcat (s1, s2, s3);

(c) strcat (s3, s2, s1);

(d) strcpy (s3, strcat (s1, s2));

(e) strcmp (s3, strcat (s1, s2));

(f) strcpy (strcat (s1, s2), s3);

4.15 What will be the output of the following statement? **[LO 4.9 M]**

```
printf ("%d", strcmp ("push", "pull"));
```

4.16 Assume that s1, s2 and s3 are declared as follows: **[LO 4.10 H]**

```
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));
```

4.17 What will be the output of the following segment? **[LO 4.10 E]**

```
char s1[ ] = "Kolkotta" ;
char s2[ ] = "Pune" ;
strcpy (s1, s2) ;
printf("%s", s1) ;
```

4.18 What will be the output of the following segment? **[LO 4.10 E]**

```
char s1[ ] = "NEW DELHI" ;
char s2[ ] = "BANGALORE" ;
strncpy (s1, s2, 3) ;
printf("%s", s1) ;
```

4.19 What will be the output of the following code? **[LO 4.10 E]**

```
char s1[ ] = "Jabalpur" ;
char s2[ ] = "Jaipur" ;
printf(strncmp(s1, s2, 2) );
```

4.20 What will be the output of the following code? **[LO 4.10 E]**

```
char s1[ ] = "ANIL KUMAR GUPTA";
char s2[ ] = "KUMAR";
printf (strstr (s1, s2) );
```

4.21 Compare the working of the following functions: **[LO 4.10 M]**

(a) strcpy and strncpy;
(b) strcat and strncat; and
(c) strcmp and strncmp.

DEBUGGING EXERCISES

4.1 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);` **[LO 4.2, 4.4 E]**
(b) `float values [10,15];` **[LO 4.2, 4.4 E]**
(c) `char name[15];` **[LO 4.2, 4.4 E]**
(d) `float average[ROW],[COLUMN];` **[LO 4.2, 4.4 M]**
(e) `double salary [i + ROW]` **[LO 4.2, 4.4 M]**
(f) `long int number [ROW]` **[LO 4.2, 4.4 M]**
(g) `int sum[ ];` **[LO 4.2, 4.4 H]**
(h) `int array x[COLUMN];` **[LO 4.2, 4.4 H]**

4.2 Identify errors, if any, in each of the following initialization statements.

(a) `int number[ ] = {0,0,0,0,0};` **[LO 4.2, 4.4 M]**
(b) `float item[3][2] = {0,1,2,3,4,5};` **[LO 4.2, 4.4 M]**
(c) `char word[ ] = {'A','R', 'R', 'A', 'Y'};` **[LO 4.2, 4.4 M]**
(d) `int m[2,4] = {(0,0,0,0)(1,1,1,1)};` **[LO 4.2, 4.4 M]**
(e) `float result[10] = 0;` **[LO 4.2 H]**

4.3 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]);
```
[LO 4.2 E]

(b)
```
for (i=1; i<=5; i++)
for(j=1; j<=4; j++)
A[i][j] = 0;
```
[LO 4.2, 4.4 M]

(c)
```
for (i=0; i<=4; i++)
B[i] = B[i]+i;
```
[LO 4.2, 4.4 M]

(d)
```
for (i=4; i>=0; i––)
for (j=0; j<4; j++)
A[i][j] = B[j] + 1.0;
```
[LO 4.2, 4.4 M]

4.4 What is the error in the following program? **[LO 4.2 M]**

```
main ( )
{
        int x ;
        float y [ ] ;
        ......
  }
```

4.5 What is the output of the following program? **[LO 4.2 M]**

```
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) ;
}
```

4.6 What is the output of the following program? **[LO 4.2 M]**

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

4.7 Find errors, if any, in the following code segments: **[LO 4.10 M]**

(a) char str[10]
strncpy(str, “GOD”, 3);
printf(“%s”, str);

(b) char str[10];
strcpy(str, “Balagurusamy”);

(c) if strstr(“Balagurusamy”, “guru”) == 0);
printf(“Substring is found”);

(d) char s1[5], s2[10],
gets(s1, s2);

PROGRAMMING EXERCISES

4.1 Write a program for fitting a straight line through a set of points (x_i, y_i), i = 1,....,n. **[LO 4.4 H]**
The straight line equation is

$$y = mx + c$$

and the values of m and c are given y

$$m = \frac{n\,\Sigma\left(x_1 y_i\right) - \left(\Sigma x_1\right)\left(\Sigma y_i\right)}{n\left(\Sigma x_i^2\right) - \left(\Sigma x_i\right)^2}$$

$$c = \frac{1}{n}\left(\Sigma\, y_i - m\,\Sigma\, x_i\right)$$

All summations are from 1 to n.

4.2 The daily maximum temperatures recorded in 10 cities during the month of January (for all 31 days) have been tabulated as follows: **[LO 4.4 M]**

	City			
Day	1	2	3 -	10
1			- -	
2				
3				
–				
–				
–				
–				
31				

Write a program to read the table elements into a two-dimensional array **temperature**, and to find the city and day corresponding to

(a) the highest temperature and

(b) the lowest temperature.

4.3 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. **[LO 4.2 M]**

4.4 The following set of numbers is popularly known as Pascal's triangle. **[LO 4.5 H]**

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.

4.5 The annual examination results of 100 students are tabulated as follows: **[LO 4.2 M]**

Roll No.	*Subject 1*	*Subject 2*	*Subject 3*
. . .			

Write a program to read the data and determine the following:

(a) Total marks obtained by each student.

(b) The highest marks in each subject and the Roll No. of the student who secured it.

(c) The student who obtained the highest total marks.

4.6 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. **[LO 4.2 H]**

4.7 Two matrices that have the same number of rows and columns can be multiplied to produce a third matrix. Consider the following two matrices. **[LO 4.4 M]**

$$A = \begin{bmatrix} a_{11}\ a_{12}.....a_{1n} \\ a_{12}\ a_{22}.....a_{2n} \\ . \qquad\quad . \\ . \qquad\quad . \\ . \qquad\quad . \\ a_{n1} \ldots\ldots\ a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11}\ b_{12}.....b_{1n} \\ b_{12}\ b_{22}.....b_{2n} \\ . \qquad\quad . \\ . \qquad\quad . \\ . \qquad\quad . \\ b_{n1} \ldots\ldots\ b_{nn} \end{bmatrix}$$

The product of **A** and **B** is a third matrix C of size n×n where each element of C is given by the following equation.

$$\mathbf{C}_{ij} \sum_{k=1}^{n} = a_{ik}b_{kj}$$

Write a program that will read the values of elements of A and B and produce the product matrix **C.**

4.8 Write a program that fills a five-by-five matrix as follows: **[LO 4.4 M]**

- 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

4.9 Selection sort is based on the following idea:

Selecting the largest array element and swapping it with the last array element leaves an unsorted list whose size is 1 less than the size of the original list. If we repeat this step again on the unsorted list we will have an ordered list of size 2 and an 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. **[LO 4.2 M]**

4.10 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 4.9 or use any other sorted list. **[LO 4.2 H]**

4.11 Write a program that will compute the length of a given character string. **[LO 4.2 E]**

4.12 Write a program that will count the number occurrences of a specified character in a given line of text. Test your program. **[LO 4.2 E]**

4.13 Write a program to read a matrix of size m × n and print its transpose. **[LO 4.2 E]**

4.14 Every book published by international publishers should carry an International Standard Book Number (ISBN). It is a 10 character 4 part number as shown below.

0-07-041183-2

The first part denotes the region, the second represents publisher, the third identifies the book and the fourth is the check digit. The check digit is computed as follows:

Sum = (1 × first digit) + (2 × second digit) + (3 × third digit) + - - - - + (9 × ninth digit).

Check digit is the remainder when sum is divided by 11. Write a program that reads a given ISBN number and checks whether it represents a valid ISBN. **[LO 4.2 M]**

4.15 Write a program to read two matrices A and B and print the following: **[LO 4.4 L]**
(a) A + B; and
(b) A – B.

4.16 Write a program, which reads your name from the keyboard and outputs a list of ASCII codes, which represent your name. **[LO 4.10 M]**

4.17 Write a program to do the following: **[LO 4.9 M]**
(a) To output the question “Who is the inventor of C ?”
(b) To accept an answer.
(c) To print out “Good” and then stop, if the answer is correct.
(d) To output the message ‘try again’, if the answer is wrong.
(e) To display the correct answer when the answer is wrong even at the third attempt and stop.

4.18 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. **[LO 4.10 M]**

4.19 Write a program which will read a text and count all occurrences of a particular word. **[LO 4.10 H]**

4.20 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. **[LO 4.10 H]**

4.21 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.” **[LO 4.10 H]**

4.22 A Maruti car dealer maintains a record of sales of various vehicles in the following form: **[LO 4.10 M]**

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).

4.23 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). **[LO 4.10 M]**

4.24 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”. **[LO 4.9 H]**

4.25 Develop a program that will read and store the details of a list of students in the format **[LO 4.10 H]**

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)

4.26 Write a program to read two strings and compare them using the function **strncmp( )** and print a message that the first string is equal, less, or greater than the second one. **[LO 4.10 E]**

4.27 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 ( ). [LO 4.10 M]**

4.28 Write a program that will copy m consecutive characters from a string s1 beginning at position n into another string s2. **[LO 4.10 E]**

4.29 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. **[LO 4.10 M]**

4.30 Given a string

char str [ ] = “123456789” ;

Write a program that displays the following: **[LO 4.9 M]**

```
        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
```

Chapter 5

Functions

LEARNING OBJECTIVES

LO 5.1	Outline user-defined functions
LO 5.2	Identify the elements of user-defined functions
LO 5.3	Explain the different categories of functions
LO 5.4	Know the concept of recursion
LO 5.5	Describe how arrays are passed to functions
LO 5.6	Discuss the relevance of storage classes on scope, visibility and lifetime of variables

INTRODUCTION

We have mentioned earlier that one of the strengths of C language is C functions. They are easy to define and use. We have used functions in every program that we have discussed so far. However, they have been primarily limited to the three functions, namely, **main, printf**, and **scanf**. In this chapter, we shall consider in detail the following:

- How a function is designed?
- How a function is integrated into a program?
- How two or more functions are put together? and
- How they communicate with one another?

C functions can be classified into two categories, namely, *library* functions and *user-defined* functions. **main** is an example of user-defined functions. **printf** and **scanf** belong to the category of library functions. We have also used other library functions such as **sqrt, cos, strcat,** etc. The main distinction between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library. In fact, this is one of the strengths of C language.

NEED FOR USER-DEFINED FUNCTIONS

LO 5.1
Outline 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'**.

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. 5.1. 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.

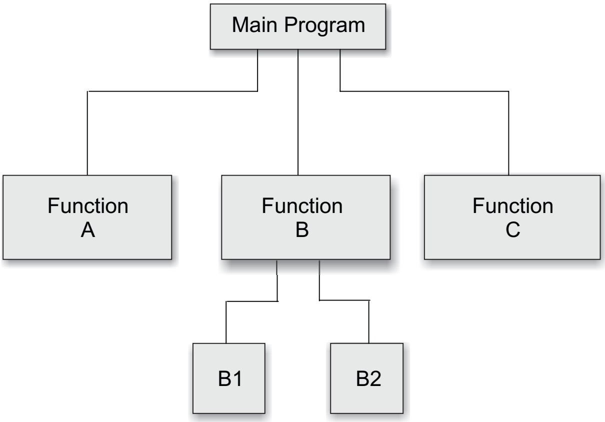


Fig. 5.1 *Top-down modular programming using functions*

A MULTI-FUNCTION PROGRAM

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. All that the program knows about a function is: What goes in and what comes out. Every C program can be designed using a collection of these black boxes known as *functions*.

Consider a set of statements as shown below:

```
void  printline(void)
{
     int i;
     for (i=1; i<40; i++)
            printf("–");
     printf("\n");
}
```

The above set of statements defines a function called **printline,** which could print a line of 39-character length. This function can be used in a program as follows:

```
void printline(void); /* declaration  */
   main( )
  {
       printline( );
       printf("This illustrates the use of C functions\n");
       printline();
  }
     void printline(void)
  {
       int i;
      for(i=1; i<40; i++)
      printf("–");
      printf("\n");
  }
```

This program will print the following output:

```
–––––––––––––––––––––––––––––––––––––––
This illustrates the use of C functions
–––––––––––––––––––––––––––––––––––––––
```

The above program contains two user-defined functions:

main() function
printline() function

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

```
printline( );
```

which indicates that the function **printline** is to be executed. At this point, the program control is transferred to the function **printline.** After executing the **printline** function, which outputs a line of 39 character length, the control is transferred back to the **main.** Now, the execution continues at the point where the function call was executed. After executing the **printf** statement, the control is again transferred to the **printline** function for printing the line once more.

The **main** function calls the user-defined **printline** function two times and the library function **printf** once. We may notice that the **printline** function itself calls the library function **printf** 39 times repeatedly.

Any function can call any other function. In fact, it can call itself. A 'called function' can also call another function. A function can be called more than once. In fact, this is one of the main features of using functions. Figure 5.2 illustrates the flow of control in a multi-function program.

Except the starting point, there are no other predetermined relationships, rules of precedence, or hierarchies among the functions that make up a complete program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end. See the box "Modular Programming".

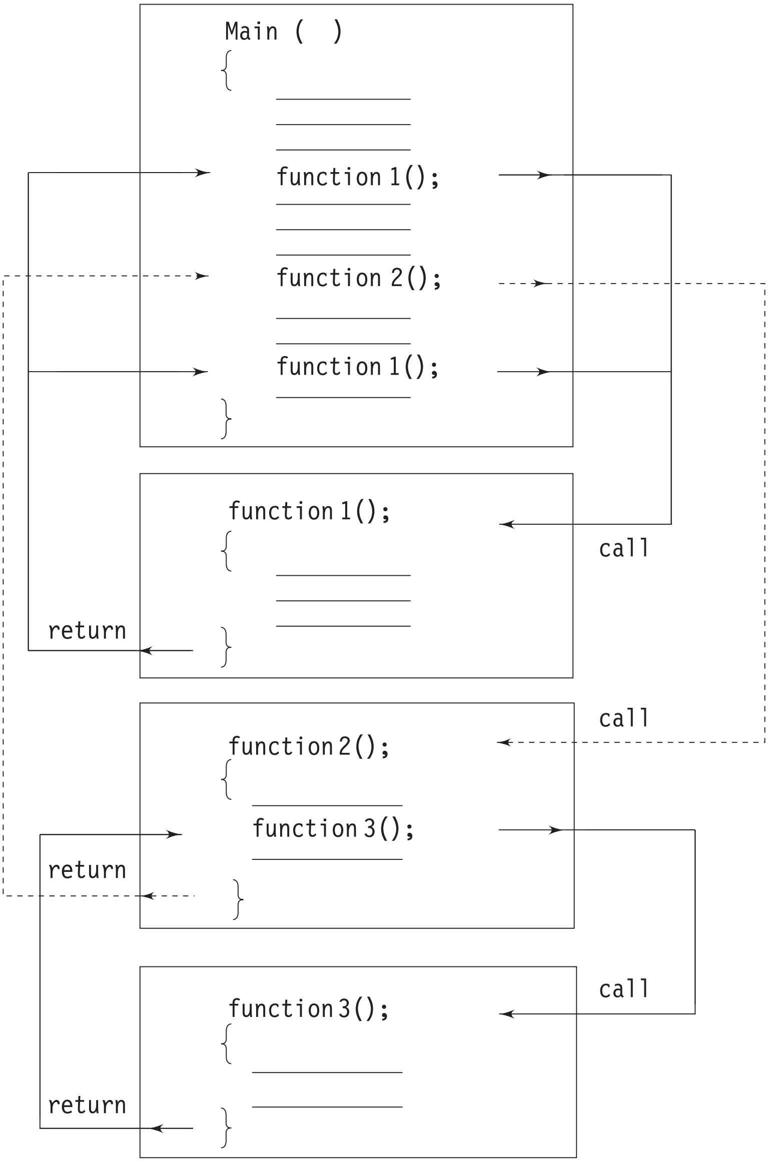


Fig. 5.2 *Flow of control in a multi-function program*

Modular Programming

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called ***modules*** that are separately named and individually callable *program units*. These modules are carefully integrated to become a software system that satisfies the system requirements. It is basically a "divide-and-conquer" approach to problem solving.

Modules are identified and designed such that they can be organized into a top-down hierarchical structure (similar to an organization chart). In C, each module refers to a function that is responsible for a single task.

Some characteristics of modular programming are 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.

ELEMENTS OF USER-DEFINED FUNCTIONS

LO 5.2
Identify the 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.*

DEFINITION OF FUNCTIONS

A *function definition,* also known as *function implementation* shall include the following elements:

1. function name;
2. function type;

3. list of parameters;
4. local variable declarations;
5. function statements; and
6. a return statement.

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

- function header (First three elements); and
- function body (Second three elements).

A general format of a function definition to implement these two parts is given below:

```
function_type  function_name(parameter list)
  {
     local variable declaration;
     executable statement1;
     executable statement2;
     . . . . .
     . . . . .
     return statement;
  }
```

The first line **function_type function_name(parameter list)** is known as the *function header* and the statements within the opening and closing braces constitute the *function body,* which is a compound statement.

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.

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.

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 nill parameter list.

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.*

RETURN VALUES AND THEIR TYPES

As pointed out earlier, a function may or may not send back any value to the calling function. If it does, it is done through the **return** statement. While it is possible to pass to the called function any number of values, the called function can only return *one value* per call, at the most.

The **return** statement can take one of the following forms:

```
return;
or
return(expression);
```

The first, the 'plain' **return** does not return any value; it acts much as the closing brace of the function. When a **return** is encountered, the control is immediately passed back to the calling function. An example of the use of a simple **return** is as follows:

```
if(error)
return;
```

Note *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);
}
```

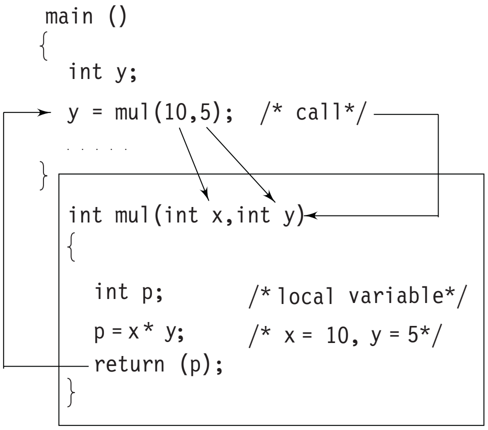
will return the value 7, only the integer part of the result.

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:



```
main ()
{
  int y;
  y = mul(10,5);  /* call*/
  . . . . .
}
int mul(int x,int y)
{
   int p;        /* local variable*/
   p = x* y;     /* x = 10, y = 5*/
   return (p);
}
```

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 **printline( )** discussed in earlier section belongs to this category. Such functions may be called in by simply stating their names as independent statements.

Example:

```
main( )
{
   printline( );
}
```

Note the presence of a semicolon at the end.

Function Call

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

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.

Prototypes: Yes or No

Prototype declarations are not essential. If a function has not been declared before it is used, C will assume that its details available at the time of linking. Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definitions. If these assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations, preferably in global declaration section.

Parameters Everywhere!

Parameters (also known as arguments) are used in 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.

CATEGORY OF FUNCTIONS

LO 5.3
Explain the different categories 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.

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

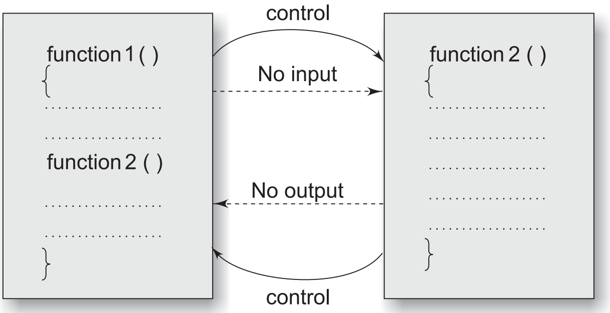


Fig. 5.3 *No data communication between functions*

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.

WORKED-OUT PROBLEM 5.1

E

Write a program with multiple functions that do not communicate any data between them.

E for Easy, **M** for Medium and **H** for High

A program with three user-defined functions is given in Fig. 5.4. **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:

value = principal(1+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;
          }
          printf("\n%8.2f %5.2f %5d %12.2f\n",
                 principal,inrate,period,sum);
      }
```

Output

```
      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        Principal amount?      5000
        Interest rate?         0.12
        Period?                5

        5000.00  0.12          5      8811.71
      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Fig. 5.4 *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.

ARGUMENTS BUT NO RETURN VALUES

In Fig. 5.4 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. 5.5.

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.

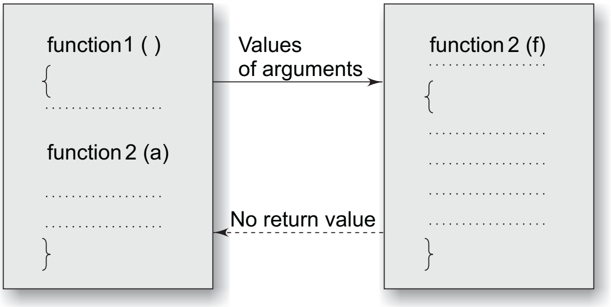


Fig. 5.5 *One-way data communication*

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. 5.6.

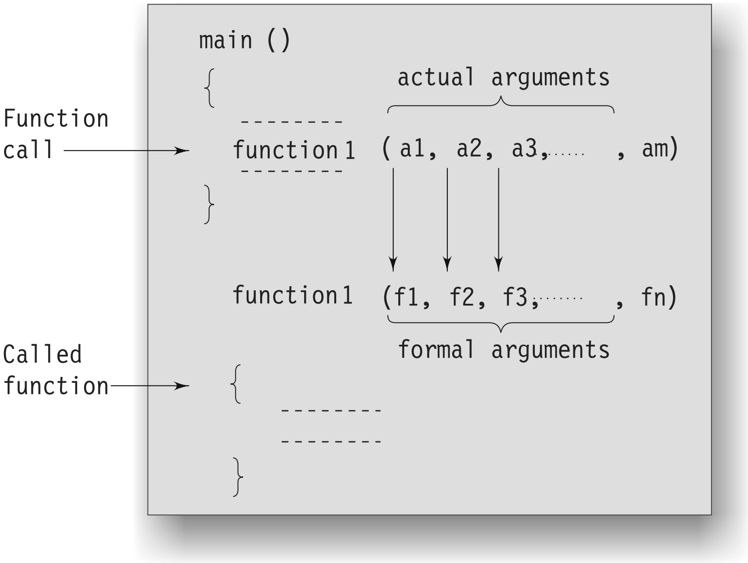

main ()
{
actual arguments
Function call
function1 (a1, a2, a3, ... , am)
}
function1 (f1, f2, f3, ... , fn)
formal arguments
Called function
{
}

Fig. 5.6 *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.

WORKED-OUT PROBLEM 5.2

M

Modify the program of Program 5.1 to include the arguments in the function calls.

The modified program with function arguments is presented in Fig. 5.7. Most of the program is identical to the program in Fig. 5.4. 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;
                   while(year <= n)
                   {
                       sum = sum * (1+r);
                       year = year +1;
                   }
                      printf("%f\t%f\t%d\t%f\n",p,r,n,sum);
              }

Output

              Enter principal amount, interest rate, and period
              5000 0.12  5
              ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
              5000.000000     0.120000        5       8811.708984
              CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Fig. 5.7 *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 **printline** 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.

ARGUMENTS WITH RETURN VALUES

The function **value** in Fig. 5.7 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. 5.8.

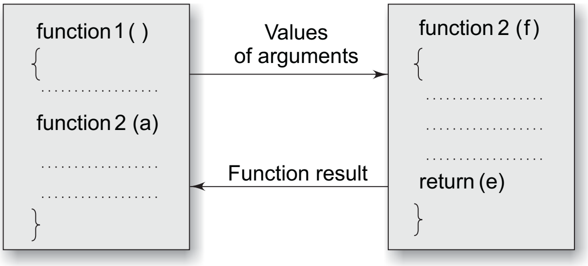


Fig. 5.8 *Two-way data communication between functions*

We shall modify the program in Fig. 5.7 to illustrate the use of two-way data communication between the *calling* and the *called functions*.

WORKED-OUT PROBLEM 5.3

M

In the program presented in Fig. 5.7 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. 5.9. One major change is the movement of the **printf** statement from **value** to **main.**

```
Program
          void printline (char ch, int len);
          value (float, float, int);

              main( )
              {
                float principal, inrate, amount;
                int period;
                printf("Enter principal amount, interest");
                printf("rate, and period\n");
                scanf(%f %f %d", &principal, &inrate, &period);
                printline ('*' , 52);
                amount = value (principal, inrate, period);
                printf("\n%f\t%f\t%d\t%f\n\n",principal,
                  inrate,period,amount);
```

```
       printline('=',52);
   }
       void printline(char ch, int len)
       {
          int i;
          for (i=1;i<=len;i++) printf("%c",ch);
          printf("\n");
       }
       value(float p, float r, int n) /* default return type */
       {
          int year;
          float sum;
          sum = p; year = 1;
          while(year <=n)
          {
             sum = sum * (l+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. 5.9 *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 **printline** function to receive the value of length of the line from the calling function. Thus, the function call

```
printline('*', 52);
```

will transfer the control to the function **printline** and assign the following values to the formal arguments **ch**, and **len:**

```
ch = '*' ;
len = 52;
```

Returning Float Values

We mentioned earlier that a C function returns a value of the type **int** as the default case when no other type is specified explicitly. For example, the function **value** of Program 5.3 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.

WORKED-OUT PROBLEM 5.4

E

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

Figure 5.10 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. 5.10 *Power functions: Illustration of return of float values*

WORKED-OUT PROBLEM 5.5

H

The program in Fig. 5.11 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. 5.11 *Program to return the position of the first minimum value in an array*

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**.

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);
}
```

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 6.

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 *.

NESTING OF FUNCTIONS

C permits nesting of functions freely. **main** can call **function1,** which calls **function2,** which calls **function3,** ………. and so on. There is in principle no limit as to how deeply functions can be nested.

Consider the following program:

```
float ratio (int x, int y, int z);
int difference (int x, int y);
main( )
{
   int a, b, c;
   scanf("%d %d %d", &a, &b, &c);
   printf("%f \n", ratio(a,b,c));
}

float ratio(int x, int y, int z)
{
   if(difference(y, z))
      return(x/(y-z));
   else
      return(0.0);
}
```

```
int difference(int p, int q)
{
    if(p != q)
      return (1);
    else
      return(0);
}
```

The above program calculates the ratio

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

and prints the result. We have the following three functions:

main( )
ratio( )
difference( )

main reads the values of a, b, and c and calls the function **ratio** to calculate the value a/(b–c). This ratio cannot be evaluated if (b–c) = 0. Therefore, **ratio** calls another function **difference** to test whether the difference (b–c) is zero or not; **difference** returns 1, if b is not equal to c; otherwise returns zero to the function **ratio**. In turn, **ratio** calculates the value a/(b–c) if it receives 1 and returns the result in **float**. In case, **ratio** receives zero from **difference**, it sends back 0.0 to **main** indicating that (b–c) = 0.

Nesting of function calls is also possible. For example, a statement like

```
P = mul(mul(5,2),6);
```

is valid. This represents two sequential function calls. The inner function call is evaluated first and the returned value is again used as an actual argument in the outer function call. If **mul** returns the product of its arguments, then the value of **p** would be 60 (= 5 × 2 × 6).

Note that the nesting does not mean defining one function within another. Doing this is illegal.

RECURSION

LO 5.4
Know the concept of 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 × 3 × 2 × 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.

PASSING ARRAYS TO FUNCTIONS

One-Dimensional Arrays

LO 5.5
Describe how arrays are passed to functions

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 an array to a called function, it is sufficient to list the name of the array, *without any subscripts,* and the size of the array as arguments. For example, the call

largest(a,n)

will pass the whole array **a** to the called function. The called function expecting this call must be appropriately defined. The **largest** function header might look like:

float largest(float array[ ], int size)

The function **largest** is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```
float array[ ];
```

The pair of brackets informs the compiler that the argument **array** is an array of numbers. It is not necessary to specify the size of the **array** here.

Let us consider a problem of finding the largest value in an array of elements. The program is as follows:

```
main( )
{
    float largest(float a[ ], int n);
    float value[4] = {2.5,-4.75,1.2,3.67};
    printf("%f\n", largest(value,4));
}
float largest(float a[], int n)
{
    int i;
    float max;
    max = a[0];
    for(i = 1; i < n; i++)
        if(max < a[i])
    max = a[i];
  return(max);
}
```

When the function call **largest**(value,4) is made, the values of all elements of array **value** become the corresponding elements of array **a** in the called function. The **largest** function finds the largest value in the array and returns the result to the **main**.

In C, the name of the array represents the address of its first element. By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the functions is referred to as *pass by address* (or pass by pointers). Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

WORKED-OUT PROBLEM 5.6

H

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 give by

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

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. 5.12 *Passing of arrays to a function*

A multifunction program consisting of **main, std_dev,** and **mean** functions is shown in Fig. 5.12. **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 5.6 highlights these concepts.

WORKED-OUT PROBLEM 5.7

M

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. 5.13. 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. 5.13 *Sorting of array elements using a function*

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]=
          {
            {1,2},
            {3,4},
            {5,6}
          };

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

PASSING STRINGS TO FUNCTIONS

The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is defined. Example:

   ```
   void display(char item_name[ ])
   {
         ......
         ......
   }
   ```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

   ```
   void display(char str[ ]);
   ```

3. A call to the function must have a string array name without subscripts as its actual argument. Example:

   ```
   display (names);
   ```

   where **names** is a properly declared string array in the calling function.

   We must note here that, like arrays, strings in C cannot be passed by value to functions.

Pass by Value versus Pass by Pointers

The technique used to pass data from one function to another is known as *parameter passing*. Parameter passing can be done in 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.

THE SCOPE, VISIBILITY, AND LIFETIME OF VARIABLES

LO 5.6
Discuss the relevance of storage classes on 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.

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.

WORKED-OUT PROBLEM 5.8

M

Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 5.14. **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. 5.14 *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.

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.

WORKED-OUT PROBLEM 5.9

E

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

A program to illustrate the properties of global variables is presented in Fig. 5.15. 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. 5.15 *Illustration of properties of global variables*

Once a variable has been declared as global, any function can use it and change its value. Then, subsequent functions can reference only that new value.

Global Variables as Parameters

Since all functions in a program source file can access global variables, they can be used for passing values between the functions. However, using global variables as parameters for passing values poses certain problems.

- The values of global variables which are sent to the called function may be changed inadvertently by the called function.
- Functions are supposed to be independent and isolated modules. This character is lost, if they use global variables.
- It is not immediately apparent to the reader which values are being sent to the called function.
- A function that uses global variables suffers from reusability.

One other aspect of a global variable is that it is available only from the point of declaration to the end of the program. Consider a program segment as shown below:

```
main( )
{
     y = 5;
     . . . .
     . . . .
}
int y;       /* global declaration */
func1( )
{
     y = y+1;
}
```

We have a problem here. As far as **main** is concerned, **y** is not defined. So, the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement

```
y = y+1;
```

in **fun1** will, therefore, assign 1 to y.

External Declaration

In the program segment above, the **main** cannot access the variable y as it has been declared after the **main** function. This problem can be solved by declaring the variable with the storage class **extern.**

For example:

```
main( )
{
     extern int y;   /* external declaration */
     . . . . .
     . . . . .
}
func1( )
{
     extern int y;   /* external declaration */
     . . . . .
     . . . . .
}
int y;                  /* definition */
```

Although the variable **y** has been defined after both the functions, the *external declaration* of **y** inside the functions informs the compiler that y is an integer type defined somewhere else in the program. Note that **extern** declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.

Example:

```
main( )
{    int i;
     void print_out(void);
     extern float height [ ];
     . . . . .
     . . . . .
     print_out( );
}
void print_out(void)
{
     extern float height [ ];
     int i;
     . . . . .
     . . . . .
}
float height[SIZE];
```

An **extern** within a function provides the type information to just that one function. We can provide type information to all functions within a file by placing external declarations before any of them.

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.

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.

WORKED-OUT PROBLEM 5.10

E

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

The program in Fig. 5.16 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. 5.16 *Illustration of static variable*

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to **stat, x** is incremented to 1. Because **x** is static, this value persists and therefore, the next call adds another 1 to **x** giving it a value of 2. The value of **x** becomes three when the third call is made.

Had we declared **x** as an **auto** variable, the output would have been:

```
x = 1
x = 1
x = 1
```

This is because each time **stat** is called, the auto variable x is initialized to zero. When the function terminates, its value of 1 is lost.

An external **static** variable is declared outside of all functions and is available to all the functions in that program. The difference between a **static** external variable and a simple external variable is that the **static** external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining 'that' function with the storage class **static**.

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

Table 5.1 *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:

```
main( )
{
        int a = 20;
        int b = 10;
        . . . . .                                   Outer
        {                                           block
                int a = 0;              Inner
                int c = a + b;          block
                 . . . . .
        }
        b = a;
}
```

When this program is executed, the value c will be 10, not 30. The statement b = a; assigns a value of 20 to **b** and not zero. Although the scope of **a** extends up to the end of **main** it is not “visible” inside the inner block where the variable **a** has been declared again. The inner **a** hides the visibility of the outer **a** in the inner block. However, when we leave the inner block, the inner **a** is no longer in scope and the outer **a** becomes visible again.

Remember, the variable **b** is not re-declared in the inner block and therefore it is visible in both the blocks. That is why when the statement `int c = a + b;` is evaluated, **a** assumes a values of 0 and **b** assumes a value of 10.

Although main's variables are visible inside the nested block, the reverse is not true.

Scope Rules

Scope

The region of a program in which a variable is available for use.

Visibility

The program's ability to access a variable from the memory.

Lifetime

The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an **auto** variable declared in **main** is the entire program execution time, although its scope is only the **main** function.
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A **static** local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.
7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

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 5.17 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

main( )
{
         extern int m;
         int i;
         . . . . .
         . . . . .
}
function1( )
{
         int j;
         . . . . .
         . . . . .
  }
```

```
file2.c

int m /* global variable */
function2( )
{
       int i;
       . . . . .
       . . . . .
}
function3( )
{
       int count;
     . . . . . . . .
     . . . . . . . . .
}
```

Fig. 5.17 *Use of extern in a multifile program*

The multifile program shown in Fig. 5.18 can be modified as shown in Fig. 5.17.

```
file1.c
int m; /* global variable */

main( )
{        {
         int i;
         . . . . .
}
function1( )
{
         int j;
         . . . . .
}
```

```
file2.c
extern int m;

function2( )

          int i;
          . . . . .
}
function3( )
{
int count;
. . . . .
}
```

Fig. 5.18 *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.**

KEY CONCEPTS

- **ARGUMENTS:** Are the set of values that are passed to a function to enable the function to perform the desired task. **[LO 5.1]**
- **BLOCK STATEMENT:** Is a set of statements enclosed within a set of braces. **[LO 5.1]**
- **FUNCTION:** Is an independently coded subprogram that performs a specific task. **[LO 5.1]**
- **MODULAR PROGRAMMING:** Is a software development approach that organizes a large program into small, independent program segments called modules. **[LO 5.1]**
- **CALLING PROGRAM:** Is the program or function that calls another function. **[LO 5.2]**
- **FUNCTION BODY:** Contains the statement block for performing the required task. **[LO 5.2]**
- **FUNCTION TYPE:** Specifies the type of value that the function will return. **[LO 5.2]**
- **PARAMETER LIST:** Is a list of variables that will receive data values at the time of function call. **[LO 5.2]**
- **PROGRAM DEFINITION:** Is an independent program module that is written to perform specific task. It is also referred as function definition. **[LO 5.2]**
- **RECURSION:** Is a scenario where a function calls itself. **[LO 5.4]**
- **EXTERNAL VARIABLE:** Is a variable that is active throughout the program. It is also referred as global variable. **[LO 5.6]**
- **LOCAL VARIABLE:** Is a variable that is active only within a specific function or statement block. It is also referred as internal variable. **[LO 5.6]**

ALWAYS REMEMBER

- A function that returns a value can be used in expressions like any other C variable. **[LO 5.1]**
- A function that returns a value cannot be used as a stand-alone statement. **[LO 5.1]**
- Where more functions are used, they may be placed in any order. **[LO 5.1]**
- It is a syntax error if the types in the declaration and function definition do not match. **[LO 5.2]**
- It is a syntax error if the number of actual parameters in the function call do not match the number in the declaration statement. **[LO 5.2]**
- It is a logic error if the parameters in the function call are placed in the wrong order. **[LO 5.2]**
- Placing a semicolon at the end of header line is illegal. **[LO 5.2]**
- Forgetting the semicolon at the end of a prototype declaration is an error. **[LO 5.2]**
- A **return** statement can occur anywhere within the body of a function. **[LO 5.2]**
- A function definition may be placed either after or before the **main** function. **[LO 5.2]**
- A **return** statement is required if the return type is anything other than **void**. **[LO 5.3]**
- If a function does not return any value, the return type must be declared **void**. **[LO 5.3]**
- If a function has no parameters, the parameter list must be declared **void**. **[LO 5.3]**
- Using **void** as return type when the function is expected to return a value is an error. **[LO 5.3]**
- Trying to return a value when the function type is marked **void** is an error. **[LO 5.3]**
- Defining a function within the body of another function is not allowed. **[LO 5.3]**

- It is an error if the type of data returned does not match the return type of the function. **[LO 5.3]**
- It will most likely result in logic error if there is a mismatch in data types between the actual and formal arguments. **[LO 5.3]**
- Functions return integer value by default. **[LO 5.3]**
- A function without a return statement cannot return a value, when the parameters are passed by value. **[LO 5.3]**
- When the value returned is assigned to a variable, the value will be converted to the type of the variable receiving it. **[LO 5.3]**
- Function cannot be the target of an assignment. **[LO 5.3]**
- 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. **[LO 5.3]**
- A function can have more than one return statement. **[LO 5.3]**
- A recursive function must have a condition that forces the function to return without making the recursive call; otherwise the function will never return. **[LO 5.4]**
- It is illegal to use the name of a formal argument as the name of a local variable. **[LO 5.5]**
- 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). **[LO 5.5]**
- Use parameter passing by values as far as possible to avoid inadvertent changes to variables of calling function in the called function. **[LO 5.5]**
- Although not essential, include parameter names in the prototype declarations for documentation purposes. **[LO 5.5]**
- A global variable used in a function will retain its value for future use. **[LO 5.6]**
- A local variable defined inside a function is known only to that function. It is destroyed when the function is exited. **[LO 5.6]**
- A global variable is visible only from the point of its declaration to the end of the program. **[LO 5.6]**
- 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. **[LO 5.6]**
- A local variable declared **static** retains its value even after the function is exited. **[LO 5.6]**
- Static variables are initialized at compile time and therefore, they are initialized only once. **[LO 5.6]**
- Avoid the use of names that hide names in outer scope. **[LO 5.6]**

BRIEF CASES

Calculation of Area under a Curve [LO 5.1, 5.2, 5.3, 5.6 M]

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

$$\text{Area} = 0.5 * (h1 + h2) * b$$

where h1 and h2 are the heights of two sides and b is the width as shown in Fig. 5.19.

The program in Fig. 5.21 calculates the area for a curve of the function

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

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

Input

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

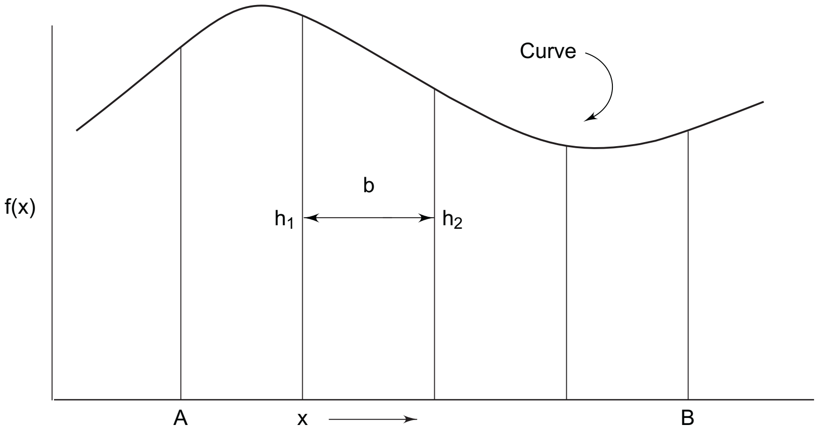


Fig. 5.19 *Area under a curve*

Output

Total area under the curve between the given limits.

Algorithm

1. Input the lower and upper limits and the number of trapezoids.
2. Calculate the width of trapezoids.
3. Initialize the total area.
4. Calculate the area of trapezoid and add to the total area.
5. Repeat step-4 until all the trapezoids are completed.
6. Print total area.

The algorithm is implemented in top-down modular form as in Fig. 5.20.

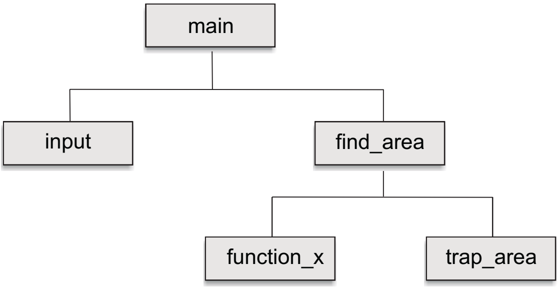


Fig. 5.20 *Modular chart*

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. 5.21 *Computing area under a curve*

REVIEW QUESTIONS

5.1 State whether the following statements are *true* or *false*.

(a) Any name can be used as a function name. **[LO 5.1 E]**

(b) A function without a **return** statement is illegal. **[LO 5.2 E]**

(c) A function prototype must always be placed outside the calling function. **[LO 5.2 E]**

(d) The variable names used in prototype should match those used in the function definition. **[LO 5.2 E]**

(e) The return type of a function is **int** by default. **[LO 5.2 M]**

(f) When variable values are passed to functions, a copy of them are created in the memory. **[LO 5.2 M]**

(g) A function can be defined within the **main** function. **[LO 5.2 M]**

(h) A function can be defined and placed before the **main** function. **[LO 5.2 M]**

(i) C functions can return only one value under their function name. **[LO 5.3 E]**

(j) A function in C should have at least one argument. **[LO 5.3 E]**

(k) Only a **void** type function can have **void** as its argument. **[LO 5.3 E]**

(l) Program execution always begins in the main function irrespective of its location in the program. **[LO 5.3 M]**

(m) In parameter passing by pointers, the formal parameters must be prefixed with the symbol * in their declarations. **[LO 5.3 M]**

(n) In parameter passing by pointers, the actual parameters in the function call may be variables or constants. **[LO 5.3 H]**

(o) An user-defined function must be called at least once; otherwise a warning message will be issued. **[LO 5.3 H]**

(p) A function can call itself. **[LO 5.4 E]**

(q) In passing arrays to functions, the function call must have the name of the array to be passed without brackets. **[LO 5.5 E]**

(r) In passing strings to functions, the actual parameter must be name of the string post-fixed with size in brackets. **[LO 5.5 M]**

(s) Global variables are visible in all blocks and functions in the program. **[LO 5.6 E]**

(t) Global variables cannot be declared as **auto** variables. **[LO 5.6 H]**

5.2 Fill in the blanks in the following statements.

(a) The parameters used in a function call are called ______. **[LO 5.1 E]**

(b) In prototype declaration, specifying ____ is optional. **[LO 5.2 E]**

(c) A _______ aids the compiler to check the matching between the actual arguments and the formal ones. **[LO 5.2 H]**

(d) In passing by pointers, the variables of the formal parameters must be prefixed with _______ in their declaration. **[LO 5.3 M]**

(e) By default, ______ is the return type of a C function. **[LO 5.3 M]**

(f) A function that calls itself is known as a _______ function. **[LO 5.4 E]**

(g) A variable declared inside a function is called _________. **[LO 5.6 E]**

(h) _________ refers to the region where a variable is actually available for use. **[LO 5.6 M]**

(i) If a local variable has to retain its value between calls to the function, it must be declared as _________. **[LO 5.6 M]**

(j) A variable declared inside a function by default assumes ______ storage class. **[LO 5.6 M]**

5.3 The **main** is a user-defined function. How does it differ from other user-defined functions? **[LO 5.1 H]**

5.4 Describe the two ways of passing parameters to functions. When do you prefer to use each of them? **[LO 5.5 M]**

5.5 What is prototyping? Why is it necessary? **[LO 5.2 E]**

5.6 Distinguish between the following:

(a) Actual and formal arguments **[LO 5.2 M]**

(b) & operator and * operator **[LO 5.3 M]**

(c) Global and local variables **[LO 5.6 E]**

(d) Automatic and static variables **[LO 5.6 M]**

(e) Scope and visibility of variables **[LO 5.6 M]**

5.7 Explain what is likely to happen when the following situations are encountered in a program. **[LO 5.3 H]**

(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.

5.8 Which of the following prototype declarations are invalid? Why? **[LO 5.2 M]**

(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);`

5.9 Which of the following header lines are invalid? Why? **[LO 5.2 M]**

(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)`

5.10 A function to divide two floating point numbers is as follows: **[LO 5.3 M]**

```
          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)

5.11 What will be the effect on the above function calls if we change the header line as follows: **[LO 5.3 H]**

(a) int divide (int x, int y)

(b) double divide (float x, float y)

5.12 Determine the output of the following program? **[LO 5.3 H]**

```
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);
     }
```

5.13 What will be the output of the following program? **[LO 5.3 H]**

```
     void test (int *a);
     main ( )
     {
          int x = 50;
          test ( &x);
          printf("%d\n", x);
     }
     void test (int *a);
     {
          *a = *a + 50;
     }
```

5.14 The function **test** is coded as follows: **[LO 5.3 M]**

```
     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);
```

5.15 Enumerate the rules that apply to a function call. **[LO 5.2 M]**

5.16 Summarize the rules for passing parameters to functions by pointers. **[LO 5.3 M]**

5.17 What are the rules that govern the passing of arrays to function? **[LO 5.5 M]**

5.18 State the problems we are likely to encounter when we pass global variables as parameters to functions. **[LO 5.6 H]**

DEBUGGING EXERCISES

5.1 Find errors, if any, in the following function definitions: **[LO 5.3 H]**

(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;
}
```

5.2 Find errors in the following function calls: **[LO 5.3 M]**

(a) `void xyz ( );`

(b) `xyx ( void );`

(c) `xyx ( int x, int y);`

(d) `xyzz ( );`

(e) `xyz ( ) + xyz ( );`

PROGRAMMING EXERCISES

5.1 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. **[LO 5.2, 5.6 M]**

5.2 Write a function **space(x)** that can be used to provide a space of x positions between two output numbers. Demonstrate its application. **[LO 5.2 M]**

5.3 Use recursive function calls to evaluate **[LO 5.4 H]**

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

5.4 An n_order polynomial can be evaluated as follows: **[LO 5.5 H]**

$$P = (.....(((a_0x+a_1)x+a_2)x+a_3)x+...+a_n)$$

Write a function to evaluate the polynomial, using an array variable. Test it using a main program.

5.5 The Fibonacci numbers are defined recursively as follows: **[LO 5.3 M]**

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1}+F_{n-2},\ n > 2$$

Write a function that will generate and print the first n Fibonacci numbers. Test the function for n = 5, 10, and 15.

5.6 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. **[LO 5.2 E]**

5.7 Write a function **prime** that returns 1 if its argument is a prime number and returns zero otherwise. **[LO 5.2 E]**

5.8 Write a function that will scan a character string passed as an argument and convert all lowercase characters into their uppercase equivalents. **[LO 5.5 H]**

5.9 Develop a top_down modular program to implement a calculator. The program should request the user to input two numbers and display one of the following as per the desire of the user: **[LO 5.2, 5.3 H]**

(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.

5.10 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. **[LO 5.2, 5.3 H]**

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

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

5.11 Write a function that can be called to find the largest element of an m by n matrix. **[LO 5.5 M]**

5.12 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. **[LO 5.5 M]**

5.13 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. **[LO 5.3, 5.5 M]**

5.14 Develop a top-down modular program that will perform the following tasks: **[LO 5.3, 5.5 H]**

(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.

5.15 Develop your own functions for performing following operations on strings: **[LO 5.3, 5.5 M]**
(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.

5.16 Write a program that invokes a function called **find( )** to perform the following tasks: **[LO 5.5 M]**
(a) Receives a character array and a single character.
(b) Returns 1 if the specified character is found in the array, 0 otherwise.

5.17 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**. **[LO 5.5 H]**

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).

5.18 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. **[LO 5.3 M]**

5.19 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. **[LO 5.2 E]**

What modifications are required if we want to use the function in preparing the actual calendar?

5.20 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). **[LO 5.2 M]**

Pointers

Chapter
6

LEARNING OBJECTIVES

LO 6.1 Know the concept of pointers

LO 6.2 Determine how pointer variables are used in a program

LO 6.3 Describe chain of pointers

LO 6.4 Illustrate pointer expressions

LO 6.5 Discuss pointers and arrays

LO 6.6 Explain how pointers are used with functions and structures

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. Chapter 8 examines the use of pointers for creating and managing linked lists.

UNDERSTANDING POINTERS

The computer's memory is a sequential collection of *storage cells* as shown in Fig. 6.1. 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.

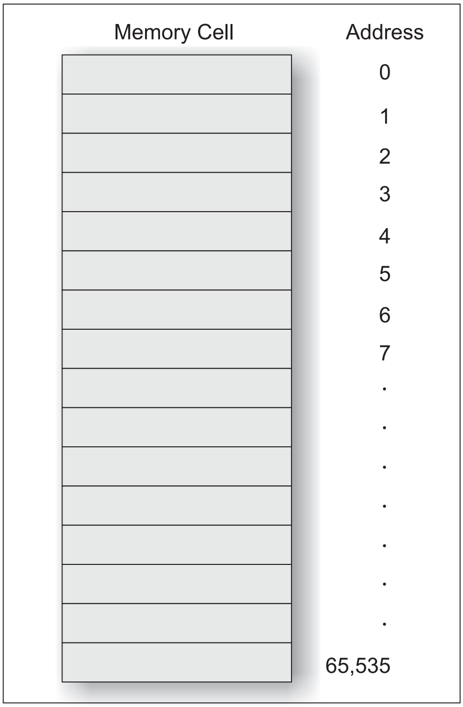


Fig. 6.1 *Memory organisation*

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. 6.2. (Note that the address of a variable is the address of the first bye occupied by that variable.)

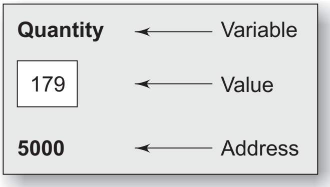


Fig. 6.2 *Representation of a variable*

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. 6.3. The address of **p** is 5048.

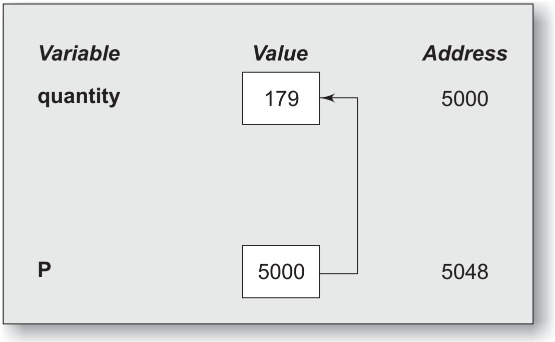
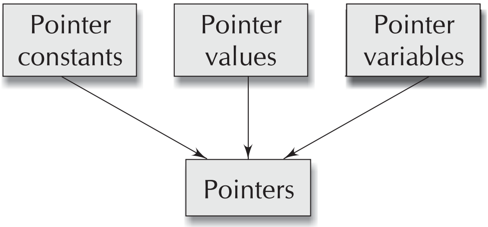


Fig. 6.3 *Pointer variable*

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**.)

Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below:



Memory addresses within a computer are referred to as *pointer constants.* We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as *pointer value.* The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a *pointer variable.*

ACCESSING THE ADDRESS OF A VARIABLE

LO 6.1
Know the concept of pointers

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**.

WORKED-OUT PROBLEM 6.1

E

Write a program to print the address of a variable along with its value.

The program shown in Fig. 6.4, 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);

}
```

E for Easy, **M** for Medium and **H** for High

```
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. 6.4 *Accessing the address of a variable*

DECLARING POINTER VARIABLES

LO 6.2
Determine how pointer variables are used in a program

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

***data_type* *pt_name;**

This tells the compiler three things about the variable **pt_name**.

1. The asterisk (*) tells that the variable **pt_name** is a pointer variable.
2. **pt_name** needs a memory location.
3. **pt_name** points to a variable of type *data_type*.

For example,

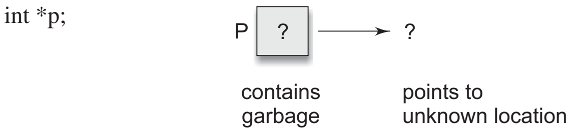
```
int *p; /* integer pointer */
```

declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

```
float *x;   / * float pointer */
```

declares **x** as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables **p** and **x**. Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:



Pointer Declaration Style

Pointer variables are declared similarly as normal variables except for the addition of the unary * operator. This symbol can appear anywhere between the type name and the printer variable name. Programmers use the following styles:

```
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;
```

INITIALIZATION OF POINTER VARIABLES

The process of assigning the address of a variable to a pointer variable is known as *initialization*. As pointed out earlier, all uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;
int *p;                    /* declaration  */
p = &quantity;             /* initialization  */
```

We can also combine the initialization with the declaration. That is,

```
int *p = &quantity;
```

is allowed. The only requirement here is that the variable **quantity** must be declared before the initialization takes place. Remember, this is an initialization of **p** and not ***p**.

We must ensure that the pointer variables always point to the corresponding type of data. For example,

```
float a, b;
int x, *p;
p = &a;            /* wrong */
b = *p;
```

will result in erroneous output because we are trying to assign the address of a **float** variable to an **integer pointer**. When we declare a pointer to be of **int** type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

```
int x, *p = &x;            /* three in one */
```

is perfectly valid. It declares **x** as an integer variable and **p** as a pointer variable and then initializes **p** to the address of **x**. And also remember that the target variable **x** is declared first. The statement

```
int *p = &x, x;
```

is not valid.

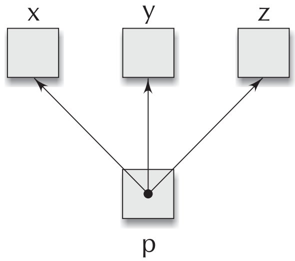
We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued

```
int *p = NULL;
int *p = 0;
```

Pointer Flexibility

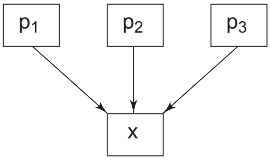
Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
        int x, y, z, *p;
        . . . . .
        p = &x;
        . . . . .
        p = &y;
        . . . . .
        p = &z;
        . . . . .
```



We can also use different pointers to point to the same data variable. Example;

```
        int x;
        int *p1 = &x;
        int *p2 = &x;
        int *p3 = &x;
        . . . . .
        . . . . .
```



With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;          / *absolute address */
```

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. Program 6.2 illustrates the distinction between pointer value and the value it points to.

WORKED-OUT PROBLEM 6.2

E

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. 6.5. The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
x = *(&x) = *ptr = y
&x = &*ptr
```

Program

```
main()
{
        int   x, y;
        int   *ptr;
        x = 10;
        ptr = &x;
        y = *ptr;
        printf("Value of x is %d\n\n",x);
        printf("%d is stored at addr %u\n", x, &x);
        printf("%d is stored at addr %u\n", *&x, &x);
        printf("%d is stored at addr %u\n", *ptr, ptr);
        printf("%d is stored at addr %u\n", ptr, &ptr);
        printf("%d is stored at addr %u\n", y, &y);
        *ptr = 25;
        printf("\nNow x = %d\n",x);

   }
```

Output

```
Value of x is 10
10   is stored at addr 4104
10   is stored at addr 4104
10   is stored at addr 4104
4104 is stored at addr 4106
10   is stored at addr 4108
Now x = 25
```

Fig. 6.5 *Accessing a variable through its pointer*

The actions performed by the program are illustrated in Fig. 6.6. The statement **ptr = &x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y**.

Note the use of the assignment statement

```
*ptr = 25;
```

This statement puts the value of 25 at the memory location whose address is the value of **ptr**. We know that the value of **ptr** is the address of **x** and therefore, the old value of **x** is replaced by 25. This, in effect, is equivalent to assigning 25 to **x**. This shows how we can change the value of a variable *indirectly* using a pointer and the *indirection operator.*

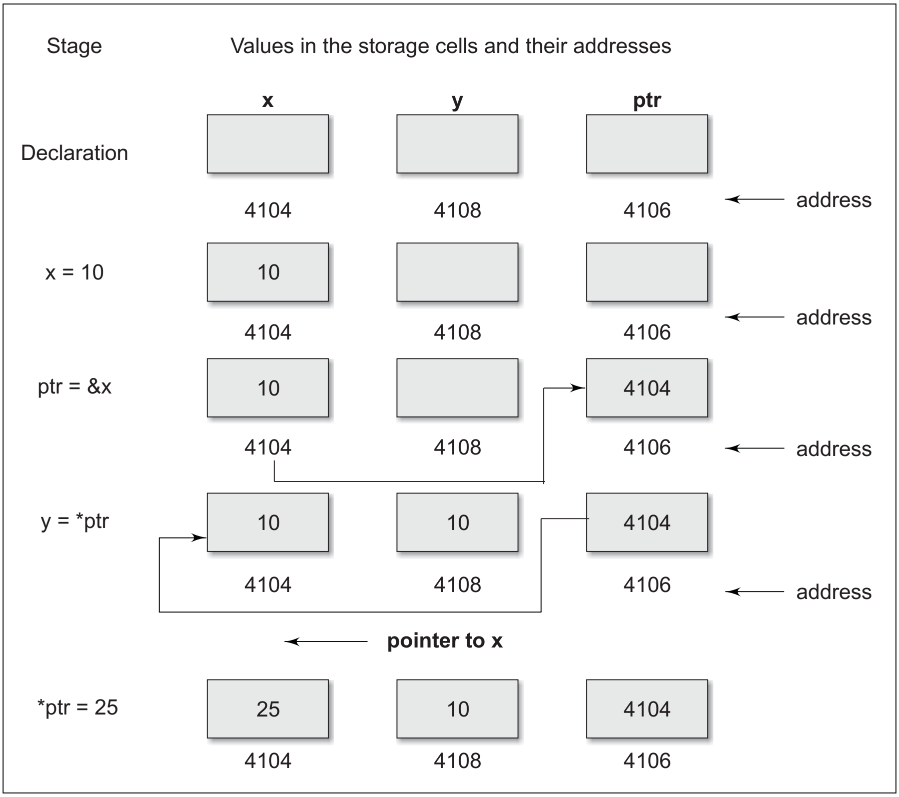
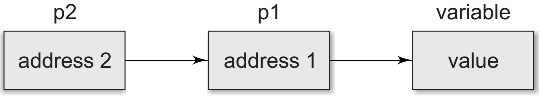


Fig. 6.6 *Illustration of pointer assignments*

CHAIN OF POINTERS

LO 6.3
Describe 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.

POINTER EXPRESSIONS

LO 6.4
Illustrate 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.

WORKED-OUT PROBLEM 6.3

Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig. 6.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

4* – *p2 / *p1 + 10

is evaluated as follows:

((4 * (–(*p2))) / (*p1)) + 10

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. 6.7 *Evaluation of pointer expressions*

POINTER INCREMENTS AND SCALE FACTOR

We have seen that the pointers can be incremented like

```
p1 = p2 + 2;
p1 = p1 + 1;
```

and so on. Remember, however, an expression like

```
p1++;
```

will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1 = p1 + 1**, the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the 'length' of the data type that it points to. This length called the *scale factor.*

For an IBM PC, the length of various data types are as follows:

characters	1 byte
integers	2 bytes
floats	4 bytes
long integers	4 bytes
doubles	8 bytes

The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if **x** is a variable, then **sizeof(x**) returns the number of bytes needed for the variable. (Systems like Pentium use 4 bytes for storing integers and 2 bytes for short integers.)

Rules of Pointer Operations

The following rules apply when performing operations on pointer variables:

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
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).

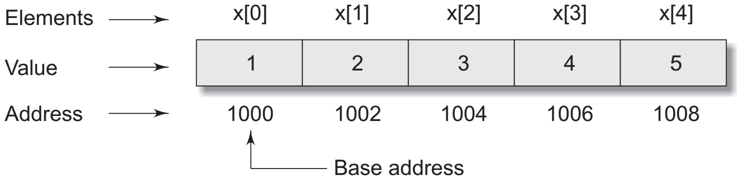
POINTERS AND ARRAYS

LO 6.5
Discuss 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:



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:

p = &x[0] (= 1000)
p+1 = &x[1] (= 1002)
p+2 = &x[2] (= 1004)
p+3 = &x[3] (= 1006)
p+4 = &x[4] (= 1008)

You may notice that the address of an element is calculated using its index and the scale factor of the data type. For instance,

address of **x[3]** = base address + (3 x scale factor of **int**)
= 1000 + (3 x 2) = 1006

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.

The Worked-Out Problem 6.4 illustrates the use of pointer accessing method.

WORKED-OUT PROBLEM 6.4

M

Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig. 6.8 illustrates how a pointer can be used to traverse an array element. Since incrementing an array pointer causes it to point to the next element, we need only to add one to **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. 6.8 *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++;
}
.....
```

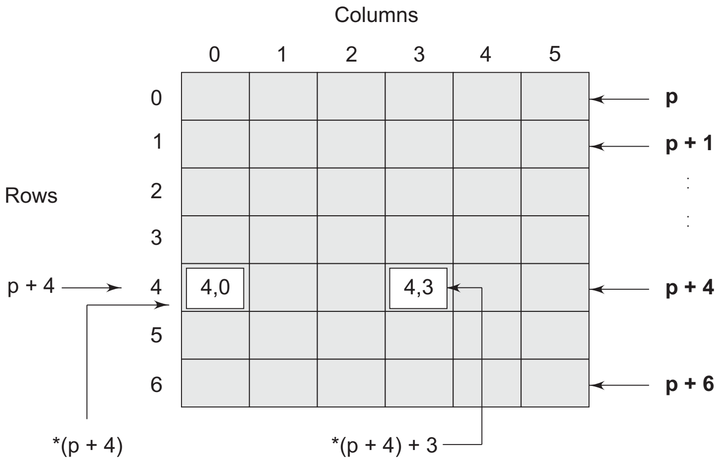
Here, we compare the pointer **p** with the address of the last element to determine when the array has been traversed.

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)`



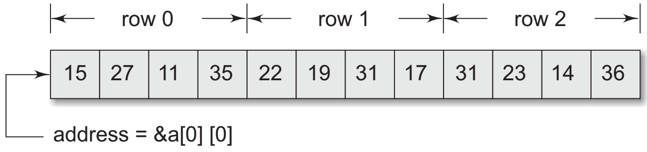
p	→	pointer to first row
p + i	→	pointer to ith row
*(p + i)	→	pointer to first element in the ith row
*(p + i) + j	→	pointer to jth element in the ith row
((p + i) + j)	→	value stored in the cell (i,j) (ith row and jth column)

Fig. 6.9 *Pointers to two-dimensional arrays*

Figure 6.9 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:



15	27	11	35	22	19	31	17	31	23	14	36

If we declare **p** as an **int** pointer with the initial address of &a[0][0], then

a[i][j] is equivalent to *(p+4 × i+j)

You may notice that, if we increment **i** by 1, the **p** is incremented by 4, the size of each row. Then the element **a[2][3]** is given by ***(p+2 × 4+3) = *(p+11)**.

This is the reason why, when a two-dimensional array is declared, we must specify the size of each row so that the compiler can determine the correct storage mapping.

POINTERS AND CHARACTER STRINGS

We have seen in Chapter 4 that strings are treated like character arrays and therefore, they are declared and initialized as follows:

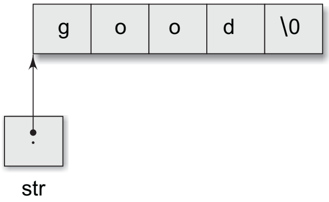
```
char str [5] = "good";
```

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.

(As pointed out in Chapter 4, 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 Worked-Out Problem 6.5.

WORKED-OUT PROBLEM 6.5

E

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. 6.10. The statement

```
char *cptr = name;
```

declares **cptr** as a pointer to a character and assigns the address of the first character of **name** as the initial value. Since a string is always terminated by the null character, the statement

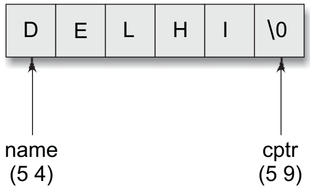
```
while(*cptr != '\0')
```

is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr – name;
```

gives the length of the string **name**.



The output also shows the address location of each character. Note that each character occupies one memory cell (byte).

Program

```
main()
{
     char  *name;
     int   length;
     char  *cptr = name;
     name  = "DELHI";
     printf ("%s\n", name);
     while(*cptr != '\0')
     {
          printf("%c is stored at address %u\n", *cptr, cptr);
          cptr++;
     }
     length = cptr - name;
     printf("\nLength of the string = %d\n", length);
}
```

Output

```
DELHI
D is stored at address 54
E is stored at address 55
L is stored at address 56
H is stored at address 57
I is stored at address 58

Length of the string = 5
```

Fig. 6.10 *String handling by pointers*

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

```
char *name;
name = "Delhi";
```

These statements will declare **name** as a pointer to character and assign to **name** the constant character string “Delhi”. You might remember that this type of assignment does not apply to character arrays. The statements like

```
char name[20];
name = "Delhi";
```

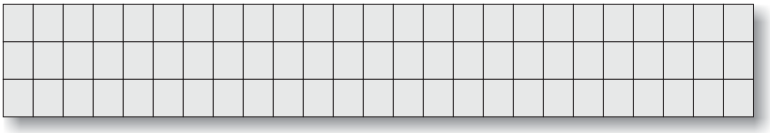
do not work.

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

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	i	a	\0						

The following statement would print out all the three names:

```
for(i = 0; i <= 2; i++)
   printf("%s\n", name[i]);
```

To access the jth character in the ith 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.

POINTERS AS FUNCTION ARGUMENTS

LO 6.6
Explain how pointers are used with functions and structures

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 (see Chapter 5).

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as '*call by reference*'. (You know, the process of passing the actual value of variables is known as "call by value".) The function which is called by 'reference' can change the value of the variable used in the call.

Consider the following code:

```
main()
{
        int x;
        x = 20;
        change(&x);  /* call by reference or address */
        printf("%d\n",x);
}
change(int *p)
{
        *p = *p + 10;
}
```

When the function **change( )** is called, the address of the variable **x**, not its value, is passed into the function **change()**. Inside **change( )**, the variable **p** is declared as a pointer and therefore **p** is the address of the variable **x**. The statement,

```
*p = *p + 10;
```

means 'add 10 to the value stored at the address **p'**. Since **p** represents the address of **x,** the value of **x** is changed from 20 to 30. Therefore, the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function. Note that this mechanism is also known as "*call by address*" or "*pass by pointers*".

Note *C99 adds a new qualifier* ***restrict*** *to the pointers passed as function parameters.*

WORKED-OUT PROBLEM 6.6

M

Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig. 6.11 shows how the contents of two locations can be exchanged using their address locations. The function **exchange()** receives the addresses of the variables **x** and **y** and exchanges their contents.

Program

```
void exchange (int *, int *);    /* prototype */
main()
{
     int  x, y;
     x = 100;
     y = 200;
     printf("Before exchange  : x = %d   y = %d\n\n", x, y);
     exchange(&x,&y);     /* call */
     printf("After exchange   : x = %d   y = %d\n\n", x, y);
}
exchange (int *a, int *b)
{
     int t;
     t = *a;    /* Assign the value at address a to t */
     *a = *b;   /* put b into a */
     *b = t;    /* put t into b */
}
```

Output

```
Before exchange  : x = 100   y = 200
After exchange   : x = 200   y = 100
```

Fig. 6.11 *Passing of pointers as function parameters*

You may note the following points:

1. The function parameters are declared as pointers.
2. The dereferenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

The use of pointers to access array elements is very common in C. We have used a pointer to traverse array elements in Program 6.4. We can also use this technique in designing user-defined functions discussed in Chapter 5. Let us consider the problem sorting an array of integers discussed in Program 5.6.

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 earlier in this chapter). 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.

WORKED-OUT PROBLEM 6.7

M

The program of Fig. 6.12 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;
     clrstr();
     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. 6.12 *Program to pass the arguments using call by reference method*

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.

POINTERS TO FUNCTIONS

A function, like a variable, has a type and an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type (*fptr) ();
```

This tells the compiler that **fptr** is a pointer to a function, which returns *type* value. The parentheses around ***fptr** are necessary. Remember that a statement like

```
type *gptr();
```

would declare **gptr** as a function returning a pointer to *type.*

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.**

WORKED-OUT PROBLEM 6.8

H

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. 6.13. The printing is done by the function **table** by evaluating the function passed to it by the **main.**

With **table,** we declare the parameter **f** as a pointer to a function as follows:

```
double (*f)();
```

The value returned by the function is of type **double.** When **table** is called in the statement

```
table (y, 0.0, 2, 0.5);
```

we pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list.

During the execution of **table,** the statement

```
value = (*f)(a);
```

calls the function **y** which is pointed to by **f,** passing it the parameter **a.** Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

```
table (cos, 0.0, PI, 0.5);
```

passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

Program

```
        #include  <math.h>
        #define  PI  3.1415926
        double y(double);
```

```
double cos(double);
double table (double(*f)(), double, double, double);

main()
{  printf("Table of y(x) = 2*x*x–x+1\n\n");
   table(y, 0.0, 2.0, 0.5);
   printf("\nTable of cos(x)\n\n");
   table(cos, 0.0, PI, 0.5);
}
double table(double(*f)(),double min, double max, double step)
{    double a, value;
     for(a = min; a <= max; a += step)
     {
        value = (*f)(a);
        printf("%5.2f  %10.4f\n", a, value);
     }
}
double y(double x)
{
return(2*x*x-x+1);
}
```

Output

```
Table of y(x) = 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. 6.13 *Use of pointers to functions*

Compatibility and Casting

A variable declared as a pointer is not just a *pointer type* variable. It is also a pointer to a *specific* fundamental data type, such as a character. A pointer therefore always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as *incompatibility* of pointers.

All the pointer variables store memory addresses, which are compatible, but what is not compatible is the underlying data type to which they point to. We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using **cast** operator, as we do with the fundamental types. Example:

```
int x;
char *p;
p = (char *) & x;
```

In such cases, we must ensure that all operations that use the pointer **p** must apply casting properly.

We have an exception. The exception is the void pointer (void *). The void pointer is a ***generic pointer*** that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

```
void *vp;
```

Remember that since a void pointer has no object type, it cannot be de-referenced.

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 –> 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–>** 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 *.

WORKED-OUT PROBLEM 6.9

H

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. 6.14. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent)** is also used as the loop control index in **for** loops.

Program

```
struct invent
{
   char    *name[20];
   int     number;
   float price;
};
main()
{
   struct invent product[3], *ptr;
   printf("INPUT\n\n");
   for(ptr = product; ptr < product+3; ptr++)
      scanf("%s %d %f", ptr–>name, &ptr–>number, &ptr–>price);
   printf("\nOUTPUT\n\n");
      ptr = product;
      while(ptr < product + 3)
      {
            printf("%–20s %5d %10.2f\n",
                     ptr–>name,
                     ptr–>number,
                     ptr–>price);
            ptr++;
      }
   }
```

Output

```
INPUT
Washing_machine    5    7500
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. 6.14 *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.**

TROUBLES WITH POINTERS

Pointers give us tremendous power and flexibility. However, they could become a nightmare when they are not used correctly. The major problem with wrong use of pointers is that the compiler may not detect the error in most cases and therefore the program is likely to produce unexpected results. The output may not give us any clue regarding the use of a bad pointer. Debugging therefore becomes a difficult task.

We list here some pointer errors that are more commonly committed by the programmers.

- Assigning values to uninitialized pointers

```
int * p, m = 100 ;
*p = m ;          /* Error */
```

- Assigning value to a pointer variable

```
int *p, m = 100 ;
p = m;            /* Error */
```

- Not dereferencing a pointer when required

```
int *p, x = 100;
p = &x;
printf("%d",p);   /* Error */
```

- Assigning the address of an uninitialized variable

```
int m, *p
p = &m;            /* Error */
```

- Comparing pointers that point to different objects

```
char name1 [ 20 ], name2 [ 30 ];
  char *p1 = name1;
  char *p2 = name2;
  if(p1 > p2).......        /* Error */
```

We must be careful in declaring and assigning values to pointers correctly before using them. We must also make sure that we apply the address operator & and referencing operator * correctly to the pointers. That will save us from sleepless nights.

KEY CONCEPTS

- **MEMORY:** This is a sequential collection of storage cells with each cell having an address value associated with it. **[LO 6.1]**
- **POINTER:** It is used to store the memory address as value. **[LO 6.1]**
- **POINTER VARIABLE:** It is a variable that stores the memory address of another variable. **[LO 6.2]**
- **CALL BY REFERENCE:** It is the process of calling a function using pointers to pass the addresses of variables. **[LO 6.6]**
- **CALL BY VALUE:** It is the process of passing the actual value of variables. **[LO 6.6]**

ALWAYS REMEMBER

- Only an address of a variable can be stored in a pointer variable. **[LO 6.1]**
- Do not store the address of a variable of one type into a pointer variable of another type. **[LO 6.1]**
- The value of a variable cannot be assigned to a pointer variable. **[LO 6.1]**
- 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. **[LO 6.1]**

- Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing. **[LO 6.1]**
- 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. **[LO 6.2]**
- It is an error to assign a numeric constant to a pointer variable. **[LO 6.2]**
- It is an error to assign the address of a variable to a variable of any basic data types. **[LO 6.2]**
- 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. **[LO 6.4]**
- 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). **[LO 6.4]**
- 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. **[LO 6.5]**
- 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. **[LO 6.6]**
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable. **[LO 6.6]**
- 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). **[LO 6.6]**

BRIEF CASES

1. Processing of Examination Marks [LO 6.2, 6.5, 6.6 H]

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. 6.15 stores the student names in the array **name** and the marks in the array **marks.** After computing the total marks obtained by all the students, the program prepares and prints the rank list. The declaration

```
int marks[STUDENTS][SUBJECTS+1];
```

defines **marks** as a pointer to the array's first row. We use **rowptr** as the pointer to the row of **marks.** The **rowptr** is initialized as follows:

```
int (*rowptr)[SUBJECTS+1] = array;
```

Note that **array** is the formal argument whose values are replaced by the values of the actual argument **marks.** The parentheses around ***rowptr** makes the **rowptr** as a pointer to an array of **SUBJECTS+1** integers. Remember, the statement

```
int *rowptr[SUBJECTS+1];
```

would declare **rowptr** as an array of **SUBJECTS+1** elements.

When we increment the **rowptr** (by **rowptr+1**), the incrementing is done in units of the size of each row of **array,** making **rowptr** point to the next row. Since **rowptr** points to a particular row, **(*rowptr)[x]** points to the xth element in the row.

Program

```
#define  STUDENTS  5
#define  SUBJECTS  4
#include <string.h>

main()
{
   char name[STUDENTS][20];
   int  marks[STUDENTS][SUBJECTS+1];

   printf("Input students names & their marks in four subjects\n");
   get_list(name, marks, STUDENTS, SUBJECTS);
   get_sum(marks, STUDENTS, SUBJECTS+1);
   printf("\n");
   print_list(name,marks,STUDENTS,SUBJECTS+1);
   get_rank_list(name, marks, STUDENTS, SUBJECTS+1);
   printf("\nRanked List\n\n");
   print_list(name,marks,STUDENTS,SUBJECTS+1);
   }
     /*   Input student name and marks        */
   get_list(char *string[ ],
           int array [ ] [SUBJECTS +1], int m, int n)
   {
        int   i, j, (*rowptr)[SUBJECTS+1] = array;
        for(i = 0; i < m; i++)
        {
               scanf("%s", string[i]);
              for(j = 0; j < SUBJECTS; j++)
                  scanf("%d", &(*(rowptr + i))[j]);
         }
   }
   /*   Compute total marks obtained by each student       */
   get_sum(int array [ ] [SUBJECTS +1], int m, int n)
   {
        int   i, j, (*rowptr)[SUBJECTS+1] = array;
        for(i = 0; i < m; i++)
        {
           (*(rowptr + i))[n-1] = 0;
           for(j =0; j < n-1; j++)
              (*(rowptr + i))[n-1] += (*(rowptr + i))[j];
     }
   }
```

```
  /*    Prepare rank list based on total marks      */

  get_rank_list(char *string [ ],
            int array [ ] [SUBJECTS + 1]
            int m,
            int n)
  {
    int i, j, k, (*rowptr)[SUBJECTS+1] = array;
    char *temp;

    for(i = 1; i <= m–1; i++)
      for(j = 1; j <= m–i; j++)
        if( (*(rowptr + j–1))[n–1] < (*(rowptr + j))[n–1])
        {
        swap_string(string[j-1], string[j]);

        for(k = 0; k < n; k++)
        swap_int(&(*(rowptr + j–1))[k],&(*(rowptr+j))[k]);
        }
}
/*      Print out the ranked list         */
print_list(char *string[ ],
        int array [] [SUBJECTS + 1],
        int m,
        int n)
{
    int  i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
        printf("%–20s", string[i]);
        for(j = 0; j < n; j++)
            printf("%5d", (*(rowptr + i))[j]);
            printf("\n");
    }
}
/*    Exchange of integer values          */
swap_int(int *p, int *q)
{
    int  temp;
    temp = *p;
```

```
          *p   = *q;
          *q   = temp;
     }

     /*     Exchange of strings         */
     swap_string(char s1[ ], char s2[ ])
     {
          char  swaparea[256];
          int   i;
          for(i = 0; i < 256; i++)
             swaparea[i] = '\0';
          i = 0;
          while(s1[i] != '\0' && i < 256)
          {
             swaparea[i] = s1[i];
             i++;
          }
          i = 0;
          while(s2[i] != '\0' && i < 256)
          {
             s1[i] = s2[i];
             s1[++i] = '\0';
          }
          i = 0;
          while(swaparea[i] != '\0')
          {
             s2[i] = swaparea[i];
             s2[++i] = '\0';
          }
     }
```

Output

```
     Input students names & their marks in four subjects
     S.Laxmi 45 67 38 55
     V.S.Rao 77 89 56 69
     A.Gupta 66 78 98 45
     S.Mani 86 72 0 25
     R.Daniel 44 55 66 77

     S.Laxmi                45   67   38   55  205
     V.S.Rao                77   89   56   69  291
```

```
A.Gupta                 66   78   98   45  287
S.Mani                  86   72    0   25  183
R.Daniel                44   55   66   77  242

Ranked List
V.S.Rao                 77   89   56   69  291
A.Gupta                 66   78   98   45  287
R.Daniel                44   55   66   77  242
S.Laxmi                 45   67   38   55  205
S.Mani                  86   72    0   25  183
```

Fig. 6.15 *Preparation of the rank list of a class of students*

2. Inventory Updating [LO 6.2, 6.6 M]

The price and quantity of items stocked in a store changes every day. They may either increase or decrease. The program in Fig. 6.16 reads the incremental values of price and quantity and computes the total value of the items in stock.

The program illustrates the use of structure pointers as function parameters. **&item,** the address of the structure **item,** is passed to the functions **update()** and **mul().** The formal arguments **product** and **stock,** which receive the value of **&item,** are declared as pointers of type **struct stores.**

```
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. 6.16 *Use of structure pointers as function parameters*

REVIEW QUESTIONS

6.1 State whether the following statements are *true* or *false*.

(a) Pointer constants are the addresses of memory locations. **[LO 6.1 M]**

(b) The underlying type of a pointer variable is void. **[LO 6.1 M]**

(c) Pointer variables are declared using the address operator. **[LO 6.2 E]**

(d) It is possible to cast a pointer to float as a pointer to integer. **[LO 6.2 M]**

(e) Pointers to pointers is a term used to describe pointers whose contents are the address of another pointer. **[LO 6.3 E]**

(f) A pointer can never be subtracted from another pointer. **[LO 6.4 E]**

(g) An integer can be added to a pointer. **[LO 6.4 M]**

(h) Pointers cannot be used as formal parameters in headers to function definitions. **[LO 6.6 E]**

(i) When an array is passed as an argument to a function, a pointer is passed. **[LO 6.6 M]**

(j) Value of a local variable in a function can be changed by another function. **[LO 6.6 M]**

6.2 Fill in the blanks in the following statements:

(a) A pointer variable contains as its value the _____ of another variable. **[LO 6.1 E]**

(b) The _____operator returns the value of the variable to which its operand points. **[LO 6.1 E]**

(c) The ______operator is used with a pointer to de-reference the address contained in the pointer. **[LO 6.2 M]**

(d) The pointer that is declared as ______cannot be de-referenced. **[LO 6.2 M]**

(e) The only integer that can be assigned to a pointer variable is ______. **[LO 6.4 M]**

6.3 What is a pointer? How can it be initialized? **[LO 6.1, 6.2 E]**

6.4 A pointer in C language is **[LO 6.1 E]**

(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.

6.5 Explain the effects of the following statements: **[LO 6.2 M]**

(a) `int a, *b = &a;`

(b) `int p, *p;`

(c) `char *s;`

(d) `a = (float *) &x);`

(e) `double(*f)();`

6.6 Distinguish between (*m)[5] and *m[5]. **[LO 6.5 M]**

6.7 Given the following declarations: **[LO 6.4 H]**

```
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

6.8 Describe typical applications of pointers in developing programs. **[LO 6.1 E]**

6.9 What are the arithmetic operators that are permitted on pointers? **[LO 6.4 E]**

6.10 What is printed by the following program? **[LO 6.2, 6.3 M]**

```
int m = 100';
int * p1 = &m;
int **p2 = &p1;
printf("%d", **p2);
```

6.11 Assuming **name** as an array of 15 character length, what is the difference between the following two expressions? **[LO 6.4, 6.5 E]**

(a) name + 10; and

(b) *(name + 10).

6.12 What is the output of the following segment? **[LO 6.5 H]**

```
int m[2];
*(m+1) = 100;
*m = *(m+1);
printf("%d", m [0]);
```

6.13 What is the output of the following code? **[LO 6.4, 6.5 M]**

```
int m [2];
int *p = m;
m [0] = 100 ;
m [1] = 200 ;
printf("%d %d", ++*p, *p);
```

6.14 What is the output of the following program? **[LO 6.5, 6.6 M]**

```
int f(char *p);
main ( )
{
     char str[ ] = "ANSI";
     printf("%d", f(str) );
}
int f(char *p)
{
     char *q = p;
     while (*++p)
             ;
     return (p-q);
}
```

6.15 Given below are two different definitions of the function **search( ) [LO 6.3, 6.6 M]**

a) void search (int* m[ ], int x)
```
{
}
```
b) void search (int ** m, int x)
```
{
}
```
Are they equivalent? Explain.

6.16 Do the declarations **[LO 6.5 M]**

```
char s [ 5 ] ;
char *s;
represent the same? Explain.
```

6.17 Which one of the following is the correct way of declaring a pointer to a function? Why? **[LO 6.5, 6.6 E]**

(a) int (*p) (void);

(b) int *p (void);

DEBUGGING EXERCISES

6.1 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. **[LO 6.2, 6.4 M]**

(a) `p1 = &m;`

(b) `p2 = n;`

(c) `*p1 = &n;`

(d) `p2 = &*&m;`

(e) `m = p2–p1;`

(f) `p1 = &p2;`

(g) `m = *p1 + *p2++;`

6.2 Find the error, if any, in each of the following statements: **[LO 6.2, 6.3 E]**

(a) `int x = 10;`

(b) `int *y = 10;`

(c) `int a, *b = &a;`

(d)
```
int m;
int **x = &m;
```

6.3 What is wrong with the following code? **[LO 6.2, 6.3 M]**

```
int **p1, *p2;
p2 = &p1;
```

PROGRAMMING EXERCISES

6.1 Write a program using pointers to read in an array of integers and print its elements in reverse order. **[LO 6.5 M]**

6.2 We know that the roots of a quadratic equation of the form **[LO 6.6 M]**

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

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.

6.3 Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place. **[LO 6.5, 6.6 H]**

6.4 Write a function using pointers to add two matrices and to return the resultant matrix to the calling function. **[LO 6.5, 6.6 M]**

6.5 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. **[LO 6.5, 6.6 H]**

6.6 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. **[LO 6.5, 6.6 M]**

6.7 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. **[LO 6.5, 6.6 H]**

6.8 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. **[LO 6.2, 6.4, 6.6 H]**

(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.)

6.9 Write a function (using a pointer parameter) that reverses the elements of a given array. **[LO 6.5, 6.6 H]**

6.10 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. **[LO 6.5, 6.6 M]**

Chapter 7

Structure

LEARNING OBJECTIVES

LO 7.1 Explain how structures are used in a program

LO 7.2 Describe how structure variables and members are manipulated

LO 7.3 Discuss structures and arrays

LO 7.4 Illustrate nested structures and 'structures and functions'

LO 7.5 Determine how structures and unions differ in terms of their storage technique

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.

DEFINING A STRUCTURE

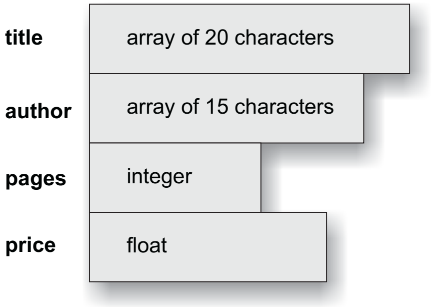
LO 7.1
Explain how structures are used in a program

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:

```
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 which are as follows:

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.

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 the following 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.

WORKED-OUT PROBLEM 7.1 M

Explain how complex number can be represented using structures. Write two C functions: one to return the sum of to complex numbers passed as parameters.

A complex number has two parts: real and imaginary. Structures can be used to realize complex numbers in C, as shown below:

```
struct complex /*Declaring the complex number datatype using structure*/
{
  double real;/*Real part*/
  double img;/*Imaginary part*/
};
```

E for Easy, **M** for Medium and **H** for High

Function to return the sum of two complex numbers

```
struct complex add(struct complex c1, struct complex c1)
{
  struct complex c3;
  c3.real=c1.real+c2.real;
  c3.img=c1.img+c2.img;
  return(c3);
}
```

Function to return the product of two complex numbers

```
struct complex product(struct complex c1, struct complex c1)
{
  struct complex c3;
  c3.real=c1.real*c2.real-c1.img*c2.img;
  c3.img=c1.real*c2.img+c1.img*c2,real;
  return(c3);
}
```

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.

WORKED-OUT PROBLEM 7.2

E

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. 7.1. 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. 7.1 *Defining and accessing structure members*

STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```
main()
{
     struct
     {
        int weight;
        float height;
```

```
         }
         student = {60, 180.75};
         .....
         .....
         }
```

This assigns the value 60 to **student. weight** and 180.75 to **student. height.** There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
   struct st_record
   {
   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 which are as follows:

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.

COPYING AND COMPARING STRUCTURE VARIABLES

LO 7.2
Describe how structure variables and members are manipulated

Two variables of the same structure type can be copied the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following statements are valid:

```
person1 = person2;
person2 = person1;
```

However, the statements such as

```
person1 == person2
person1 != person2
```

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

WORKED-OUT PROBLEM 7.3

M

Write a program to illustrate the comparison of structure variables.

The program shown in Fig. 7.2 illustrates how a structure variable can be copied into another of the same type. It also performs member-wise comparison to decide whether two structure variables are identical.

Program

```
struct class
{
     int  number;
     char name[20];
     float marks;
};

main()
{
     int  x;
     struct class student1 = {111,"Rao",72.50};
```

```
              struct class student2 = {222,"Reddy", 67.00};
              struct class student3;

              student3 = student2;

              x = ((student3.number ==  student2.number) &&
                 (student3.marks  ==  student2.marks)) ? 1 : 0;

          if(x == 1)
          {
            printf("\nstudent2 and student3 are same\n\n");
            printf("%d %s %f\n", student3.number,
                                 student3.name,
                                 student3.marks);
            }
            else
                printf("\nstudent2 and student3 are different\n\n");

          }
```

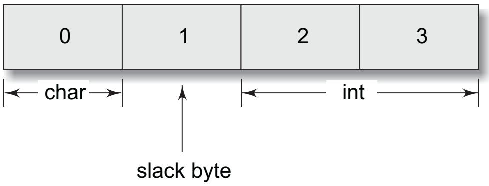
Output

```
student2 and student3 are same

222 Reddy 67.000000
```

Fig. 7.2 *Comparing and copying structure variables*

Word Boundaries and Slack Bytes

Computer stores structures using the concept of “word boundary”. The size of a word boundary is machine dependent. In a computer with two bytes word boundary, the members of a structure are stored left_aligned on the word boundary, as shown below. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as the *slack byte.*



When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined. Due to this, even if the members of two variables are equal, their structures do not necessarily compare equal. C, therefore, does not permit comparison of structures. However, we can design our own function that could compare individual members to decide whether the structures are equal or not.

OPERATIONS ON INDIVIDUAL MEMBERS

As pointed out earlier, the individual members are identified using the member operator, the *dot*. A member with the *dot operator* along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators. Consider the program in Fig. 7.2. We can perform the following operations:

```
if (student1.number == 111)
   student1.marks += 8.00;
float sum = student1.marks + student2.marks;
student2.marks * = 0.5;
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

```
student1.number ++;
++ student1.number;
```

The precedence of the *member* operator is higher than all *arithmetic* and *relational* operators and therefore no parentheses are required.

Three Ways to Access Members

We have used the dot operator to access the members of structure variables. In fact, there are two other ways. Consider the following structure:

```
typedef struct
{
        int x;
        int y;
}  VECTOR;
VECTOR v, *ptr;
ptr = & v;
```

The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable n. Now, the members can be accessed in the following three ways:

• using dot notation	:	**v.x**
• using indirection notation	:	(*ptr).x
• using selection notation	:	ptr –> x

The second and third methods will be considered in Chapter 6.

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:

LO 7.3
Discuss structures and arrays

```
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.

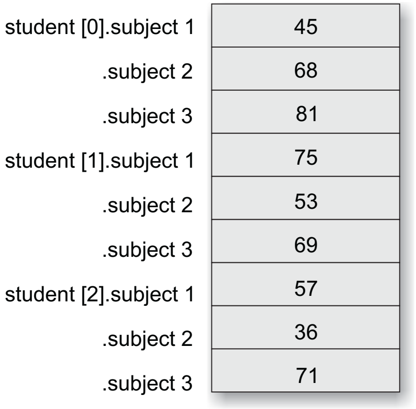
An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array **student** actually looks as shown in Fig. 7.3.

WORKED-OUT PROBLEM 7.4

M

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. 7.4. We have declared a four-member structure, the fourth one for keeping the student-totals. We have also declared an **array** total to keep the subject-totals and the grand-total. The grand-total is given by **total.total.** Note that a member name can be any valid C name and can be the same as an existing structure variable name. The linked name **total.total** represents the **total** member of the structure variable **total**.



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. 7.3 *The array student inside memory*

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. 7.4 *Arrays of structures: Illustration of subscripted structure variables*

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.

WORKED-OUT PROBLEM 7.5

E

Rewrite the program of Program 7.4 using an array member to represent the three subjects.

The modified program is shown in Fig. 7.5. You may notice that the use of array name for subjects has simplified in code.

```
Program
          main()
          {
               struct  marks
               {
                    int  sub[3];
                    int  total;
          };
               struct marks student[3] =
               {45,67,81,0,75,53,69,0,57,36,71,0};

               struct marks total;
               int  i,j;
```

```
     for(i = 0; i <= 2; i++)
     {
        for(j = 0; j <= 2; j++)
     {
        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. 7.5 *Use of subscripted members arrays in structures*

STRUCTURES WITHIN STRUCTURES

LO 7.4
Illustrate nested structures and 'structures and functions'

Structures within a structure means *nesting* of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees:

```
struct salary
{
   char  name;
   char  department;
   int   basic_pay;
```

```
  int  dearness_allowance;
  int  house_rent_allowance;
  int  city_allowance;
}
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below:

```
struct salary
{
   char name;
   char department;
   struct
   {
      int dearness;
      int house_rent;
      int city;
   }
   allowance;
}
employee;
```

The salary structure contains a member named **allowance**, which itself is a structure with three members. The members contained in the inner structure namely **dearness, house_rent**, and **city** can be referred to as:

employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following are invalid:

employee.allowance (actual member is missing)

employee.house_rent (inner structure variable is missing)

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
   .....
   struct
   {
      int dearness;
      .....
   }
   allowance,
   arrears;
}
employee[100];
```

The inner structure has two variables, **allowance** and **arrears**. This implies that both of them have the same structure template. Note the comma after the name **allowance.** A base member can be accessed as follows:

employee[1].allowance.dearness
employee[1].arrears.dearness

We can also use tag names to define inner structures. Example:

```
struct pay
  {
     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 upto 15 levels. However, C99 allows 63 levels of nesting.*

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.

WORKED-OUT PROBLEM 7.6

M

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. 7.6. 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. 7.6 *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.

UNIONS

LO 7.5
Determine how structures and unions differ in terms of their storage technique

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 7.7 shows how all the three variables share the same address. This assumes that a float variable requires 4 bytes of storage.

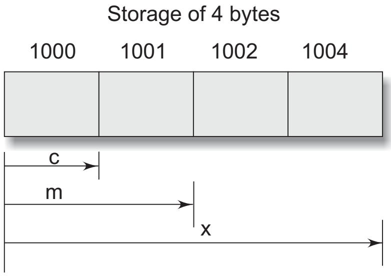


Fig. 7.7 *Sharing of a storage locating by union members*

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).

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.

SIZE OF STRUCTURES

We normally use structures, unions, and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The expression

```
sizeof(struct x)
```

will evaluate the number of bytes required to hold all the members of the structure **x.** If **y** is a simple structure variable of type **struct x**, then the expression

```
sizeof(y)
```

would also give the same answer. However, if **y** is an array variable of type **struct x,** then

```
sizeof(y)
```

would give the total number of bytes the array **y** requires.

This kind of information would be useful to determine the number of records in a database. For example, the expression

```
sizeof(y)/sizeof(x)
```

would give the number of elements in the array **y.**

BIT FIELDS

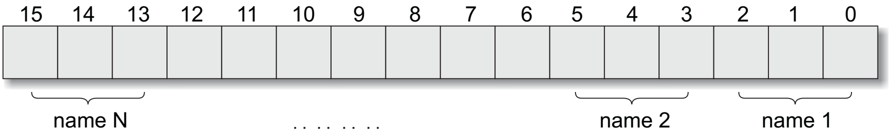
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a 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, behaviour would be unpredicted.

Suppose, we want to store and use personal information of employees in compressed form, this can be done as follows:

```
struct personal
{
unsigned sex                  :     1
unsigned age                  :     7
unsigned m_status             :     1
unsigned children             :     3
unsigned                      :     4
} emp;
```

This defines a variable name **emp** with four bit fields. The range of values each field could have is follows:

Bit field	*Bit length*	*Range of value*
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.

Note *Other related topics such as 'Structures with Pointers' and 'Structures and Linked Lists' are discussed in Chapter 6 and Chapter 9, respectively.*

KEY CONCEPTS

- **ARRAY:** It is a fixed-size sequenced collection of elements of the same data type. **[LO 7.1]**
- **DOT OPERATOR:** This links a structure variable with a structure member. It is used to read/write member values. **[LO 7.1]**
- **STRUCTURE:** This is a user-defined data type that allows different data types to be combined together to represent a data record. **[LO 7.1]**
- **UNION:** It is similar to a structure in syntax but differs in storage technique. Unlike structures, union members use the same memory location for storing all member values. **[LO 7.5]**
- **BIT FIELD:** This refers to a set of adjacent bits with size ranging from 1 to 16 bits. **[LO 7.5]**

ALWAYS REMEMBER

- Remember to place a semicolon at the end of definition of structures and unions. **[LO 7.1]**
- 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. **[LO 7.1]**
- 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.** **[LO 7.1]**
- When we use **typedef** definition, the *type_name* comes after the closing brace but before the semicolon. **[LO 7.1]**
- 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. **[LO 7.1]**
- It is an error to use a structure variable as a member of its own **struct** type structure. **[LO 7.1]**
- Declaring a variable using the tag name only (without the keyword struct) is an error. **[LO 7.1]**
- It is illegal to refer to a structure member using only the member name. **[LO 7.1]**
- When using **scanf** for reading values for members, we must use address operator & with non-string members. **[LO 7.1]**
- 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. **[LO 7.1]**
- Use short and meaningful structure tag names. **[LO 7.1]**
- Avoid using same names for members of different structures (although it is not illegal). **[LO 7.1]**
- It is an error to compare two structure variables. **[LO 7.2]**
- Assigning a structure of one type to a structure of another type is an error. **[LO 7.2]**
- When accessing a member with a pointer and dot notation, parentheses are required around the pointer, like (*ptr).number. **[LO 7.2]**
- The selection operator (–>) is a single token. Any space between the symbols – and > is an error. **[LO 7.2]**

- Forgetting to include the array subscript when referring to individual structures of an array of structures is an error. **[LO 7.3]**
- When structures are nested, a member must be qualified with all levels of structures nesting it. **[LO 7.4]**
- Passing structures to functions by pointers is more efficient than passing by value. (Passing by pointers are discussed in Chapter 6.) **[LO 7.4]**
- 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. **[LO 7.5]**
- It is an error to initialize a union with data that does not match the type of the first member. **[LO 7.5]**
- 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. **[LO 7.5]**
- Bit fields cannot be arrayed. **[LO 7.5]**

BRIEF CASES

1. Book Shop Inventory [LO 7.1, 7.2, 7.3, 7.4, 7.5 M]

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. 7.8. The program uses a template to define the structure of the book. Note that the date of publication, a member of **record** structure, is also defined as a structure.

When the title and author of a book are specified, the program searches for the book in the list using the function

look_up(table, s1, s2, m)

The parameter **table** which receives the structure variable **book** is declared as type **struct record.** The parameters **s1** and **s2** receive the string values of **title** and **author** while **m** receives the total number of books in the list. Total number of books is given by the expression

sizeof(book)/sizeof(struct record)

The search ends when the book is found in the list and the function returns the serial number of the book. The function returns –1 when the book is not found. Remember that the serial number of the first book in the list is zero. The program terminates when we respond “NO” to the question

Do you want any other book?

Note that we use the function

get(string)

to get title, author, etc. from the terminal. This enables us to input strings with spaces such as “C Language”. We cannot use **scanf** to read this string since it contains two words.

Since we are reading the quantity as a string using the **get(string)** function, we have to convert it to an integer before using it in any expressions. This is done using the **atoi()** function.

Programs

```
#include  <stdio.h>
#include  <string.h>
struct  record
{
     char    author[20];
     char    title[30];
     float   price;
     struct

     {
        char    month[10];
        int     year;
     }
     date;
     char    publisher[10];
     int     quantity;
};
  int look_up(struct record table[],char s1[],char s2[],int m);
  void get (char string [ ] );
  main()
{
     char title[30], author[20];
     int  index, no_of_records;
     char response[10], quantity[10];
     struct record book[] =  {
     {"Ritche","C Language",45.00,"May",1977,"PHI",10},
     {"Kochan","Programming in C",75.50,"July",1983,"Hayden",5},
     {"Balagurusamy","BASIC",30.00,"January",1984,"TMH",0},
     {"Balagurusamy","COBOL",60.00,"December",1988,"Macmillan",25}
                     };

  no_of_records = sizeof(book)/ sizeof(struct record);
     do
     {
        printf("Enter title and author name as per the list\n");
        printf("\nTitle:    ");
        get(title);
        printf("Author:   ");
        get(author);
        index = look_up(book, title, author, no_of_records);
     if(index != -1)     /*  Book found  */
     {
          printf("\n%s %s %.2f %s %d %s\n\n",
                     book[index].author,
                     book[index].title,
                     book[index].price,
                     book[index].date.month,
                     book[index].date.year,
                     book[index].publisher);

          printf("Enter number of copies:");
          get(quantity);
          if(atoi(quantity) < book[index].quantity)

             printf("Cost of %d copies = %.2f\n",atoi(quantity),
                   book[index].price * atoi(quantity));
          else
               printf("\nRequired copies not in stock\n\n");
          }
          else
               printf("\nBook not in list\n\n");

          printf("\nDo you want any other book? (YES / NO):");
          get(response);
   }
   while(response[0] == 'Y' || response[0] == 'y');
   printf("\n\nThank you.  Good bye!\n");
}
   void get(char string [] )
{
   char  c;
   int  i = 0;
   do
   {
        c = getchar();
        string[i++] = c;
}
while(c != '\n');
string[i-1] = '\0';
}
```

```
int look_up(struct record table[],char s1[],char s2[],int m)
{
   int  i;
   for(i = 0; i < m;  i++)
      if(strcmp(s1, table[i].title) == 0  &&
         strcmp(s2, table[i].author) == 0)
         return(i);            /* book found       */
      return(-1);              /* book not found   */
}
```

Output

```
Enter title and author name as per the list
Title:    BASIC
Author:   Balagurusamy
Balagurusamy BASIC 30.00 January 1984 TMH
Enter number of copies:5
Required copies not in stock
Do you want any other book? (YES / NO):y
Enter title and author name as per the list
Title:    COBOL
Author:   Balagurusamy
Balagurusamy COBOL 60.00 December 1988 Macmillan
Enter number of copies:7
Cost of 7 copies = 420.00

Do you want any other book? (YES / NO):y
Enter title and author name as per the list
Title:    C Programming
Author:   Ritche

Book not in list


Do you want any other book? (YES / NO):n


Thank you.  Good bye!
```

Fig. 7.8 *Program of bookshop inventory*

REVIEW QUESTIONS

7.1 State whether the following statements are *true* or *false*.

(a) A **struct** type in C is a built-in data type. **[LO 7.1 E]**

(b) The tag name of a structure is optional. **[LO 7.1 E]**

(c) Structures may contain members of only one data type. **[LO 7.1 E]**

(d) The keyword **typedef** is used to define a new data type. **[LO 7.1 E]**

(e) A structure variable is used to declare a data type containing multiple fields. **[LO 7.1 M]**

(f) It is legal to copy a content of a structure variable to another structure variable of the same type. **[LO 7.1 M]**

(g) Pointers can be used to access the members of structure variables. **[LO 7.1 H]**

(h) In accessing a member of a structure using a pointer p, the following two are equivalent: (*p)**.**member_name and p –> member_name **[LO 7.1 H]**

(i) We can perform mathematical operations on structure variables that contain only numeric type members. **[LO 7.2 M]**

(j) An array cannot be used as a member of a structure. **[LO 7.3 E]**

(k) A member in a structure can itself be a structure. **[LO 7.4 E]**

(l) Structures are always passed to functions by pointers. **[LO 7.4 H]**

(m) A union may be initialized in the same way a structure is initialized. **[LO 7.5 E]**

(n) A union can have another union as one of the members. **[LO 7.5 H]**

(o) A structure cannot have a union as one of its members. **[LO 7.5 H]**

7.2 Fill in the blanks in the following statements:

(a) The name of a structure is referred to as _______________. **[LO 7.1 E]**

(b) The variables declared in a structure definition are called its ______________. **[LO 7.1 E]**

(c) The _____________ can be used to create a synonym for a previously defined data type. **[LO 7.1 M]**

(d) The selection operator –> requires the use of a _________________ to access the members of a structure. **[LO 7.1 H]**

(e) A_______________ is a collection of data items under one name in which the items share the same storage. **[LO 7.5 E]**

7.3 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. **[LO 7.1, 7.2, 7.3 E]**

(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";`

7.4 Given the declaration **[LO 7.1, 7.2 M]**

```
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)`

```
    .....
```

7.5 Given the declaration **[LO 7.3 M]**

```
struct item_bank
{
        int number;
        double cost;
};
```

which of the following are correct statements for declaring one dimensional array of structures of type **struct item_bank?**

(a) `int item_bank items[10];`
(b) `struct items[10] item_bank;`
(c) `struct item_bank items (10);`
(d) `struct item_bank items [10];`
(e) `struct items item_bank [10];`

7.6 Given the following declaration **[LO 7.1, 7.3 M]**

```
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];`

7.7 How does a structure differ from an array? **[LO 7.3 E]**

7.8 Explain the meaning and purpose of the following: **[LO 7.1, 7.5 M]**

(a) Template
(b) **struct** keyword
(c) **typedef** keyword
(d) **sizeof** operator
(e) Tag name

7.9 Explain what is wrong in the following structure declaration: **[LO 7.1 E]**

```
struct
{
```

```
                int number;
                float price;
        }
        main( )
        {
                . . . . .
                . . . . .
        }
```

7.10 When do we use the following? **[LO 7.5 M]**

(a) Unions

(b) Bit fields

(c) The **sizeof** operator

7.11 What is meant by the following terms?

(a) Array of structures **[LO 7.3 E]**

(b) Nested structures **[LO 7.4 M]**

(c) Unions **[LO 7.5 M]**

Give a typical example of use of each of them.

7.12 Describe with examples, the different ways of assigning values to structure members. **[LO 7.1 M]**

7.13 State the rules for initializing structures. **[LO 7.1 E]**

7.14 What is a 'slack byte'? How does it affect the implementation of structures? **[LO 7.2 M]**

7.15 Describe three different approaches that can be used to pass structures as function arguments. **[LO 7.4 H]**

7.16 What are the important points to be considered when implementing bit-fields in structures? **[LO 7.5 M]**

7.17 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. **[LO 7.1 M]**

7.18 What will be the output of the following program? **[LO 7.5 H]**

```
        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

7.1 Given the structure definitions and declarations **[LO 7.1, 7.2 M]**

```
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) . . .`

7.2 What is the error in the following program? **[LO 7.1 M]**

```
typedef struct product
{
     char name [ 10 ];
     float price ;
} PRODUCT products [ 10 ];
```

PROGRAMMING EXERCISES

7.1 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: **[LO 7.1 M]**

16:40:51

7.2 Modify the above program such that a function is used to input values to the members and another function to display the time. **[LO 7.1, 7.4 M]**

7.3 Design a function **update** that would accept the data structure designed in Exercise 7.1 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.) **[LO 7.1, 7.2, 7.4 M]**

7.4 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: **[LO 7.1, 7.2, 7.4 M]**

- 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

7.5 Design a function **update** that accepts the **date** structure designed in Exercise 7.4 to increment the date by one day and return the new date. The following rules are applicable: **[LO 7.1, 7.2, 7.4 M]**

- 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

7.6 Modify the input function used in Exercise 7.4 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. [LO 7.1, 7.2, 7.4 M]**

Use suitable algorithm to convert the long integer 19450815 into year, month and day.

7.7 Add a function called **nextdate** to the program designed in Exercise 7.4 to perform the following task: **[LO 7.1, 7.2, 7.4 H]**

- Accepts two arguments, one of the structure **data** 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.

7.8 Use the **date** structure defined in Exercise 7.4 to store two dates. Develop a function that will take these two dates as input and compares them. **[LO 7.1, 7.2, 7.4 M]**

- It returns 1, if the **date1** is earlier than **date2**
- It returns 0, if **date1** is later date

7.9 Define a structure to represent a vector (a series of integer values) and write a modular program to perform the following tasks: **[LO 7.1, 7.2, 7.3, 7.4 M]**

- 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, ...)

7.10 Add a function to the program of Exercise 7.9 that accepts two vectors as input parameters and return the addition of two vectors. **[LO 7.1, 7.2, 7.3, 7.4 H]**

7.11 Create two structures named **metric** and **British** which store the values of distances. The **metric** structure stores the values in metres and centimetres 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. **[LO 7.1, 7.2 M]**

7.12 Define a structure named **census** with the following three members: **[LO 7.1, 7.3, 7.4 M]**

- 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

7.13 Define a structure that can describe an hotel. It should have members that include the name, address, grade, average room charge, and number of rooms. **[LO 7.1, 7.2 M]**

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

7.14 Define a structure called **cricket** that will describe the following information: **[LO 7.1, 7.2, 7.3 M]**

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.

7.15 Design a structure **student_record** to contain name, date of birth, and total marks obtained. Use the **date** structure designed in Exercise 7.4 to represent the date of birth. **[LO 7.1, 7.2, 7.4 M]**

Develop a program to read data for 10 students in a class and list them rank-wise.

Dynamic Memory Allocation

Chapter
8

LEARNING OBJECTIVES

LO 8.1	Describe dynamic memory allocation
LO 8.2	Differentiate between malloc and calloc
LO 8.3	Explain how allocated space is altered or released
LO 8.4	Discuss the concept of linked lists
LO 8.5	Determine how a linked list is implemented

INTRODUCTION

Most often we face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Such situations can be handled more easily and effectively by using what is known as *dynamic data structures* in conjunction with *dynamic memory management* techniques.

Dynamic data structures provide flexibility in adding, deleting or rearranging data items at run time. Dynamic memory management techniques permit us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space. This chapter discusses the concept of *linked lists*, one of the basic types of dynamic data structures. Before we take up linked lists, we shall briefly introduce the dynamic storage management functions that are available in C. These functions would be extensively used in processing linked lists.

DYNAMIC MEMORY ALLOCATION

LO 8.1
Describe dynamic memory allocation

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as *dynamic memory allocation*. Although C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution. They are listed in Table 8.1. These functions help us build complex application programs that use the available memory intelligently.

Table 8.1 *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.

Memory Allocation Process

Before we discuss these functions, let us look at the memory allocation process associated with a C program. Figure 8.1 shows the conceptual view of storage of a C program in memory.

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).

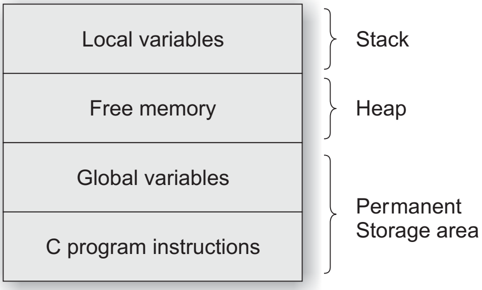


Fig. 8.1 *Storage of a C program*

ALLOCATING A BLOCK OF MEMORY: MALLOC

LO 8.2 Differentiate between malloc and calloc

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:

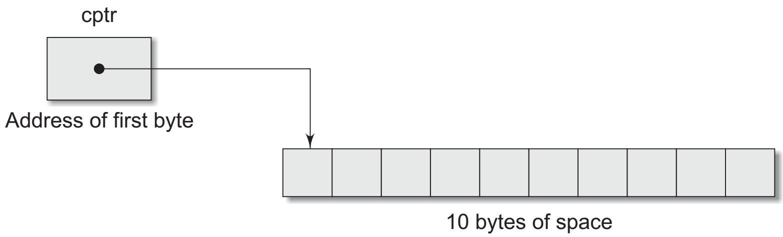
```
x = (int *) malloc (100 *sizeof(int));
```

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. 8.2.

WORKED-OUT PROBLEM 8.1 E

Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig. 8.2. 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)
```

E for Easy, **M** for Medium and **H** for High

```
                {
                   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 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. 8.2 *Memory allocation with* ***malloc***

ALLOCATING MULTIPLE BLOCKS OF MEMORY: CALLOC

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of **calloc** is:

```
ptr = (cast-type *) calloc (n, elem-size);
```

The above statement allocates contiguous space for *n* blocks, each of size *elem-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

The following segment of a program allocates space for a structure variable:

```
. . . .
. . . .
struct student
{
     char name[25];
     float age;
     long int id_num;
};
typedef struct student record;
record *st_ptr;
int class_size = 30;
st_ptr=(record *)calloc(class_size, sizeof(record));
. . . .
. . . .
```

record is of type **struct student** having three members: **name, age**, and **id_num**. The **calloc** allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the **st_ptr**. This may be done as follows:

```
if(st_ptr == NULL)
{
     printf("Available memory not sufficient");
     exit(1);
}
```

RELEASING THE USED SPACE: FREE

LO 8.3
Explain how allocated space is altered or released

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.

The use of **free** function has been illustrated in Worked-Out Problem 8.2.

ALTERING THE SIZE OF A BLOCK: REALLOC

It is likely that we discover later, the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it. In both the cases, we can change the memory size already allocated with the help of the function **realloc**. This process is called the *reallocation* of memory. For example, if the original allocation is done by the statement

```
ptr = malloc(size);
```

then reallocation of space may be done by the statement

```
ptr = realloc(ptr, newsize);
```

This function allocates a new memory space of size *newsize* to the pointer variable **ptr** and returns a pointer to the first byte of the new memory block. The *newsize* may be larger or smaller than the *size*. Remember, the new memory block may or may not begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.

If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed (lost). This implies that it is necessary to test the success of operation before proceeding further. This is illustrated in the program of Worked-Out Problem 8.2.

WORKED-OUT PROBLEM 8.2

M

Write a program to store a character string in a block of memory space created by **malloc** and then modify the same to store a larger string.

The program is shown in Fig. 8.3. The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remains same even after modification of the original size.

Program

```
#include <stdio.h>
#include<stdlib.h>
#define NULL 0
main()
{
     char *buffer;
     /* Allocating memory */
     if((buffer = (char *)malloc(10)) == NULL)
     {
        printf("malloc failed.\n");
        exit(1);
     }
     printf("Buffer of size %d created \n",_msize(buffer));
     strcpy(buffer, "HYDERABAD");
     printf("\nBuffer contains: %s \n ", buffer);
     /* Reallocation */
     if((buffer = (char *)realloc(buffer, 15)) == NULL)
     {
        printf("Reallocation failed. \n");
        exit(1);
     }
```

```
          printf("\nBuffer size modified. \n");
          printf("\nBuffer still contains: %s \n",buffer);
          strcpy(buffer, "SECUNDERABAD");
          printf("\nBuffer now contains: %s \n",buffer);
      /* Freeing memory */
      free(buffer);
      }
```

Output

```
      Buffer of size 10 created
      Buffer contains: HYDERABAD
      Buffer size modified
      Buffer still contains: HYDERABAD
      Buffer now contains: SECUNDERABAD
```

Fig. 8.3 *Reallocation and release of memory space*

CONCEPTS OF LINKED LISTS

LO 8.4
Discuss the concept of linked lists

We know that a list refers to a set of items organized sequentially. An array is an example of list. In an array, the sequential organization is provided implicitly by its index. We use the index for accessing and manipulation of array elements. One major problem with the arrays is that the size of an array must be specified precisely at the beginning. As pointed out earlier, this may be a difficult task in many real-life applications.

A completely different way to represent a list is to make each item in the list part of a structure that also contains a "link" to the structure containing the next item, as shown in Fig. 8.4. This type of list is called a *linked list* because it is a list whose order is given by links from one item to the next.

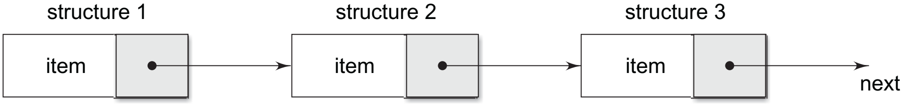
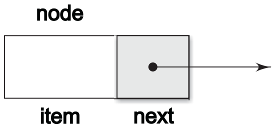


Fig. 8.4 *A linked list*

Each structure of the list is called a *node* and consists of two fields, one containing the item, and the other containing the address of the next item (a pointer to the next item) in the list. A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows:

```
struct node
{
   int item;
   struct node *next;
};
```

The first member is an integer item and the second a pointer to the next node in the list as shown below. Remember, the **item** is an integer here only for simplicity, and could be any complex data type.

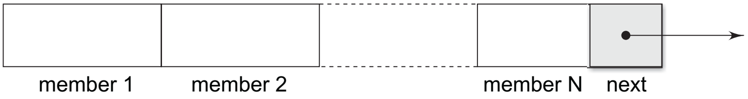


Such structures, which contain a member field that points to the same structure type are called *self-refrential* structure.

A node may be represented in general form as follows:

```
struct tag-name
{
     type member1;
     type member2;
     . . . .
     . . . .
     struct tag-name *next;
};
```

The structure may contain more than one item with different data types. However, one of the items must be a pointer of the type **tag-name**.



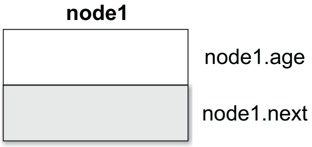
Let use consider a simple example to illustrate the concept of linking. Suppose we define a structure as follows:

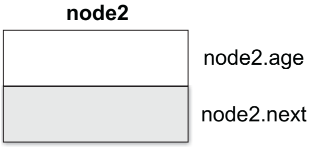
```
struct link_list
{
     float age:
     struct link_list *next;
};
```

For simplicity, let as assume that the list contains two nodes **node1** and **node2**. They are of type **struct link_list** and are defined as follows:

```
struct link_list node1, node2;
```

This statement creates space for two nodes each containing two empty fields as shown:

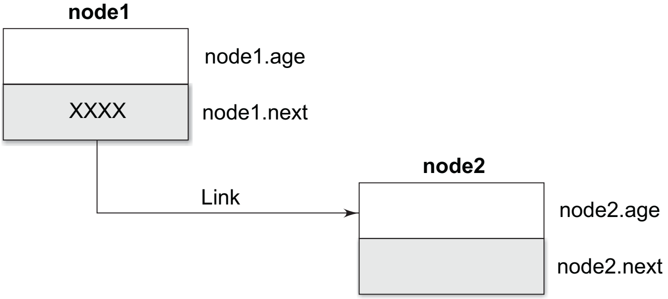




The **next** pointer of **node1** can be made to point to **node2** by the statement

```
node1.next = &node2;
```

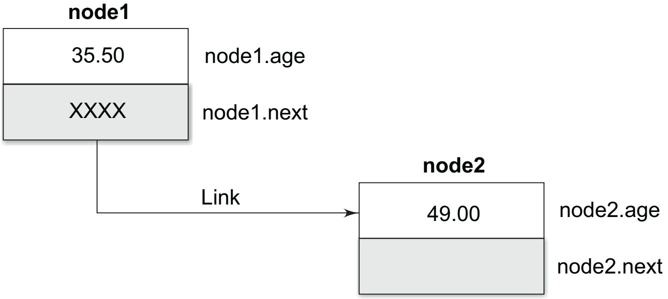
This statement stores the address of **node2** into the field **node1.next** and thus establishes a "link" between **node1** and **node2** as shown:



"xxxx" is the address of **node2** where the value of the variable **node2.age** will be stored. Now let us assign values to the field age.

```
node1.age = 35.50;
node2.age = 49.00;
```

The result is as follows:



We may continue this process to create a liked list of any number of values.

For example:

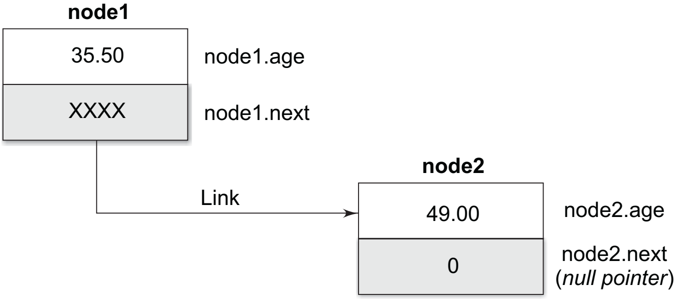
```
node2.next = &node3;
```

would add another link provided **node3** has been declared as a variable of type **struct link list.**

No list goes on forever. Every list must have an end. We must therefore indicate the end of a linked list. This is necessary for processing the list. C has a special pointer value called **null** that can be stored in the **next** field of the last node. In our two-node list, the end of the list is marked as follows:

```
node2.next = 0;
```

The final linked list containing two nodes is as shown:



The value of the age member of **node2** can be accessed using the **next** member of **node1** as follows:

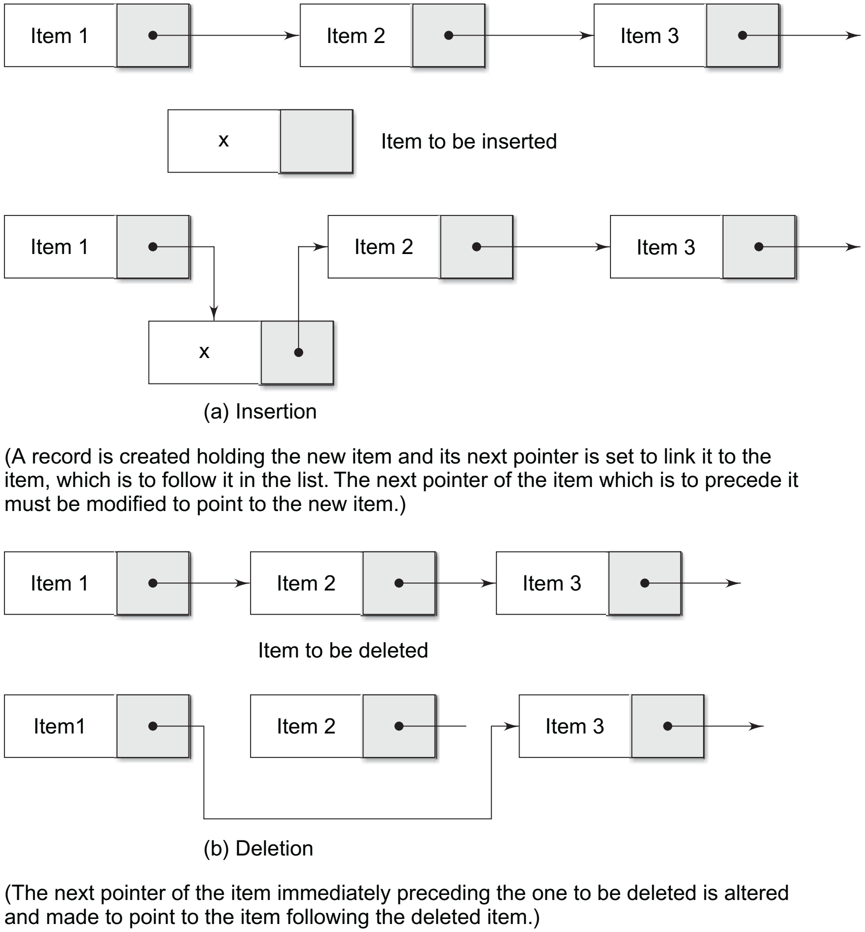
```
printf("%f\n", node1.next->age);
```

ADVANTAGES OF LINKED LISTS

A linked list is *dynamic data structure*. Therefore, the primary advantage of linked lists over arrays is that linked lists can grow or shrink in size during the execution of a program. A linked list can be made just as long as required.

Another advantage is that a linked list does not waste memory space. It uses the memory that is just needed for the list at any point of time. This is because it is not necessary to specify the number of nodes to be used in the list.

The third, and the most important advantage is that the linked lists provide flexibility is allowing the items to be rearranged efficiently. It is easier to insert or delete items by rearranging the links. This is shown in Fig. 8.5.



(a) Insertion

(A record is created holding the new item and its next pointer is set to link it to the item, which is to follow it in the list. The next pointer of the item which is to precede it must be modified to point to the new item.)

(b) Deletion

(The next pointer of the item immediately preceding the one to be deleted is altered and made to point to the item following the deleted item.)

Fig. 8.5 *Insertion into and deletion from a linked list*

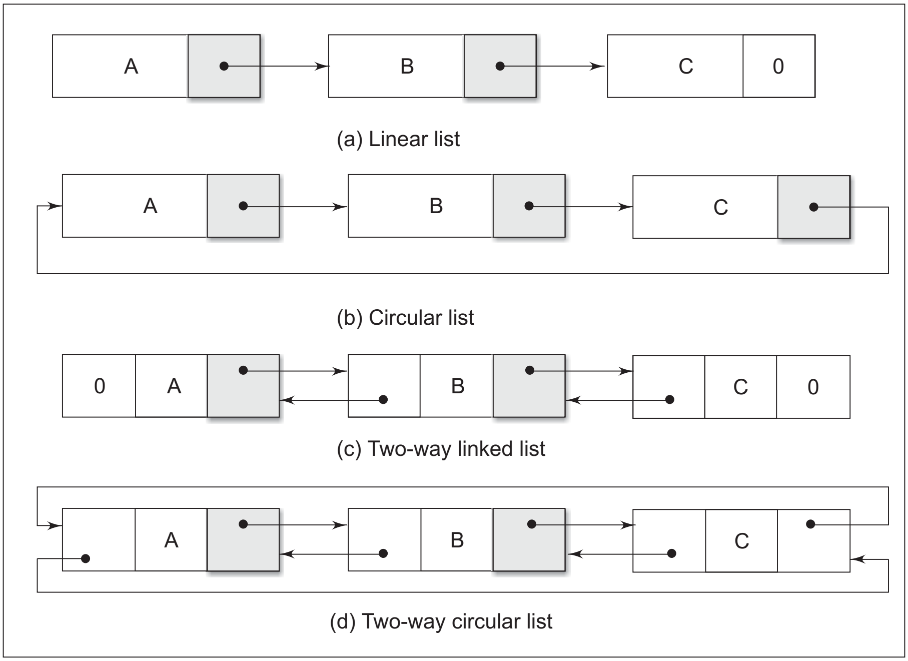
The major limitation of linked lists is that the access to any arbitrary item is little cumbersome and time consuming. Whenever we deal with a fixed length list, it would be better to use an array rather than a linked list. We must also note that a linked list will use more storage than an array with the same number of items. This is because each item has an additional link field.

TYPES OF LINKED LISTS

There are different types of lined lists. The one we discussed so far is known as *linear singly* linked list. The other *linked* lists are as follows:

- Circular linked lists.
- Two-way or doubly linked lists.
- Circular doubly linked lists.

The circular linked lists have no beginning and no end. The last item points back to the first item. The doubly linked list uses double set of pointers, one pointing to the next item and other pointing to the preceding item. This allows us to traverse the list in either direction. Circular doubly linked lists employs both the forward pointer and backward pointer in circular form. Figure 8.6 illustrates various kinds of linked lists.



(a) Linear list

(b) Circular list

(c) Two-way linked list

(d) Two-way circular list

Fig. 8.6 *Different types of linked lists*

POINTERS REVISITED

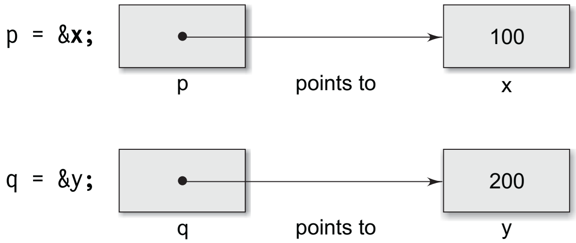
The concept of pointers was discussed in Chapter 6. Since pointers are used extensively in processing of the linked lists, we shall briefly review some of their properties that are directly relevant to the processing of lists.

LO 8.5 Determine how a linked list is implemented

We know that variables can be declared as pointers, specifying the type of data item they can point to. In effect, the pointer will hold the address of the data item and can be used to access its value. In processing linked lists, we mostly use pointers of type structures.

It is most important to remember the distinction between the pointer variable **ptr**, which contain the address of a variable, and the referenced variable ***ptr**, which denotes the value of variable to which **ptr**'s value points. The following examples illustrate this distinction. In these illustrations, we assume that the pointers **p** and **q** and the variables **x** and **y** are declared to be of same type.

(a) Initialization

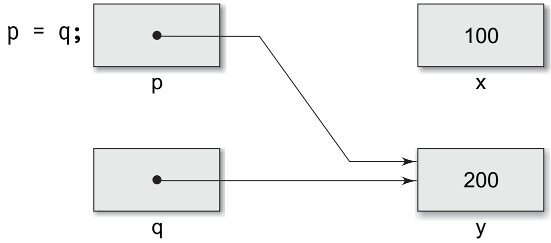


The pointer **p** contains the address of **x** and **q** contains the address of y.

***p =100 and *q = 200 and p< >q**

(b) Assignment p = q

The assignment **p = q** assigns the address of the variable **y** to the pointer variable **p** and therefore **p** now points to the variable y.

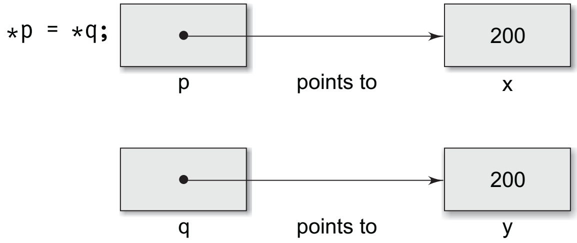


Both the pointer variables point to the same variable.

***p = *q = 200 but x <> y**

*(c) Assignment *p = *q*

This assignment statement puts the value of the variable pointed to by **q** in the location of the variable pointed to by **p**.



The pointer **p** still points to the same variable **x** but the old value of **x** is replaced by 200 (which is pointed to by **q**).

x = y = 200 but p <> q

(d) NULL pointers

A special constant known as NULL pointer (0) is available in C to initialize pointers that point to nothing. That is the statements

p = 0; (or p = NULL;) $p \rightarrow \boxed{0}$
q = 0; (q = NULL;) $q \rightarrow \boxed{0}$

make the pointers **p** and **q** point to nothing. They can be later used to point any values.

We know that a pointer must be initialized by assigning a memory address before using it. There are two ways of assigning memory address to a pointer.

1. Assigning an existing variable address (static assignment)

```
ptr = &count;
```

2. Using a memory allocation function (dynamic assignment)

```
ptr = (int*) malloc(sizeof(int));
```

CREATING A LINKED LIST

We can treat a linked list as an abstract data type and perform the following basic operations:

1. Creating a list.
2. Traversing the list.
3. Counting the items in the list.
4. Printing the list (or sub list).
5. Looking up an item for editing or printing.
6. Inserting an item.
7. Deleting an item.
8. Concatenating two lists.

In the previous sections, we created a two-element linked list using the structure variable names **node1** and **node2**. We also used the address operator **&** and member operators **.** and –> for creating and accessing individual items. The very idea of using a linked list is to avoid any reference to specific number of items in the list so that we can insert or delete items as and when necessary. This can be achieved by using "anonymous" locations to store nodes. Such locations are accessed not by name, but by means of pointers, which refer to them. (For example, we must avoid using references like **node1.age** and **node1.next** –> **age**.)

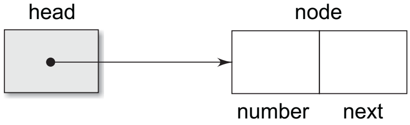
Anonymous locations are created using pointers and dynamic memory allocation functions such as **malloc**. We use a pointer **head** to create and access anonymous nodes. Consider the following:

```
struct linked_list
{
     int number;
     struct linked_list *next;
};
typedef struct linked_list node;
node *head;
head = (node *) malloc(sizeof(node));
```

The **struct** declaration merely describes the format of the nodes and does not allocate storage. Storage space for a node is created only when the function **malloc** is called in the statement

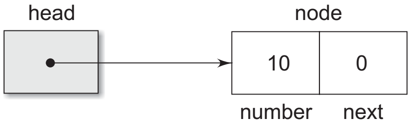
```
head = (node *) malloc(sizeof(node));
```

This statement obtains a piece of memory that is sufficient to store a node and assigns its address to the pointer variable **head**. This pointer indicates the beginning of the linked list.



The following statements store values in the member fields:

```
head –> number = 10;
head –> next = NULL;
```



The second node can be added as follows:

```
head –> next = (node *)malloc(sizeof(node));
head –> next –>number = 20;
head–>next–>next = NULL;
```

Although this process can be continued to create any number of nodes, it becomes cumbersome and clumsy if nodes are more than two. The above process may be easily implemented using both recursion and iteration techniques. The pointer can be moved from the current node to the next node by a self-replacement statement such as:

```
head = head –> next;
```

The Worked-Out Problem 8.3 shows creation of a complete linked list and printing of its contents using recursion.

WORKED-OUT PROBLEM 8.3

H

Write a program to create a linear linked list interactively and print out the list and the total number of items in the list.

The program shown in Fig. 8.7 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. 8.7 *Creating a linear linked list*

INSERTING AN ITEM

One of the advantages of linked lists is the comparative case with which new nodes can be inserted. It requires merely resetting of two pointers (rather than having to move around a list of data as would be the case with arrays).

Inserting a new item, say X, into the list has three situations:

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

The process of insertion precedes a search for the place of insertion. The search involves in locating a node after which the new item is to be inserted.

A general algorithm for insertion is as follows:

```
Begin
        if the list is empty or
        the new node comes before the head node then,
        insert the new node as the head node,
        else
        if the new node comes after the last node, then,
        insert the new node as the end node,
        else
        insert the new node in the body of the list.
End
```

Algorithm for placing the new item at the beginning of a linked list:

1. Obtain space for new node.
2. Assign data to the item field of new node.
3. Set the *next* field of the new node to point to the start of the list.
4. Change the head pointer to point to the new node.

Algorithm for inserting the new node X between two existing nodes, say, N1 and N2;

1. Set space for new node X.
2. Assign value to the item field of X.
3. Set the *next* field of X to point to node N2.
4. Set the *next* field of N1 to point to X.

Algorithm for inserting an item at the end of the list is similar to the one for inserting in the middle, except the *next* field of the new node is set to NULL (or set to point to a dummy or sentinel node, if it exists).

WORKED-OUT PROBLEM 8.4

H

Write a function to insert a given item before a specified node known as key node.

The function **insert** shown in Fig. 8.8 requests for the item to be inserted as well as the "key node". If the insertion happens to be at the beginning, then memory space is created for the new node, the value of new item is assigned to it and the pointer **head** is assigned to the next member. The pointer **new,** which indicates the beginning of the new node is assigned to **head.** Note the following statements:

```
                     new–>number = x;
                     new–>next = head;
                     head = new;
```

```
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 */
```

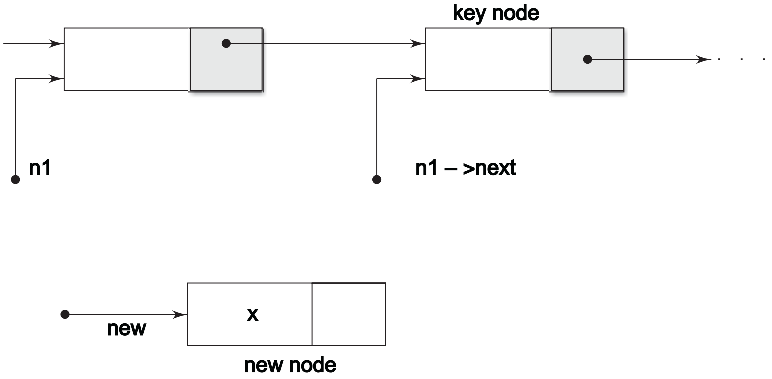
```
                  return(list);
                else

                if(list->next->next == NULL)  /* end */
                  return(NULL);
                else
                  find(list->next, key);
}
```

Fig. 8.8 *A function for inserting an item into a linked list*

However, if the new item is to be inserted after an existing node, then we use the function **find** recursively to locate the 'key node'. The new item is inserted before the key node using the algorithm discussed above. This is illustrated as:

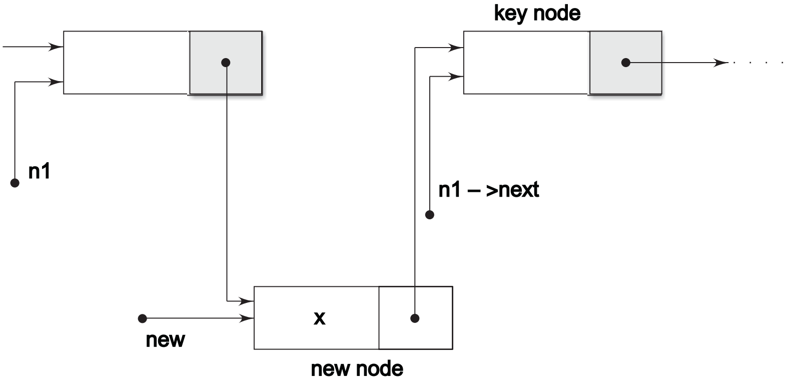
Before insertion

```
new = (node *)malloc(sizeof(node));
new->number = x;
```



After insertion

```
new->next = n1->next;
n1->next = new;
```



DELETING AN ITEM

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed. Here again we have three situations.

1. Deleting the first item.
2. Deleting the last item.
3. Deleting between two nodes in the middle of the list.

In the first case, the head pointer is altered to point to the second item in the list. In the other two cases, the pointer of the item immediately preceding the one to be deleted is altered to point to the item following the deleted item. The general algorithm for deletion is as follows:

```
Begin
        if the list is empty, then,
        node cannot be deleted
        else
        if node to be deleted is the first node, then,
        make the head to point to the second node,
        else
        delete the node from the body of the list.
End
```

The memory space of deleted node may be released for re-use. As in the case of insertion, the process of deletion also involves search for the item to be deleted.

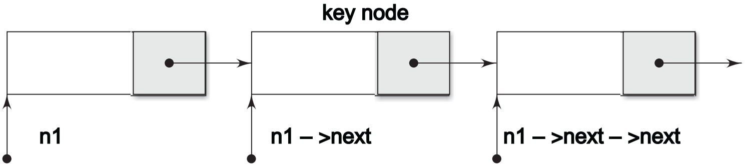
WORKED-OUT PROBLEM 8.5

H

Write a function to delete a specified node.

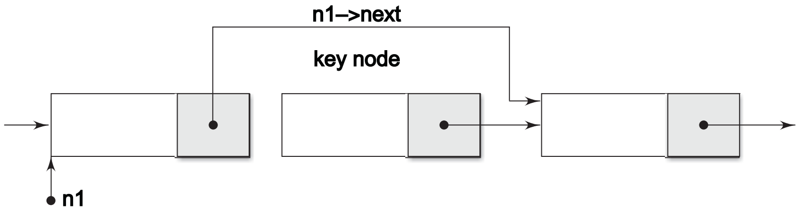
A function to delete a specified node is given in Fig. 8.9. The function first checks whether the specified item belongs to the first node. If yes, then the pointer to the second node is temporarily assigned the pointer variable **p**, the memory space occupied by the first node is freed and the location of the second node is assigned to **head**. Thus, the previous second node becomes the first node of the new list.

If the item to be deleted is not the first one, then we use the **find** function to locate the position of 'key node' containing the item to be deleted. The pointers are interchanged with the help of a temporary pointer variable making the pointer in the preceding node to point to the node following the key node. The memory space of key node that has been deleted if freed. The figure below shows the relative position of the key node.



The execution of the following code deletes the key node.

```
p = n1–>next–>next;
free (n1–>next);
n1–>next = p;
```



```
node *delete(node *head)
{
             node *find(node *p, int a);
             int  key;    /* item to be deleted */
             node *n1;    /* pointer to node preceding key node */
             node *p;     /* temporary pointer */
             printf("\n What is the item (number) to be deleted?");
             scanf("%d", &key);
             if(head–>number == key)    /* first node to be deleted) */
             {
                p = head–>next;       /* pointer to 2nd node  in list */
                free(head);           /* release space of key node */
                head = p;             /* make head to point to 1st node */
             }
             else
             {
                n1 = find(head, key);
                if(n1 == NULL)
                   printf("\n key not found \n");
             else                          /* delete key node */
             {
                   p = n1–>next–>next;        /*  pointer to the node
                                             following the keynode */

                   free(n1–>next);         /* free key node */
                   n1–>next = p;           /* establish link */
                }
             }
             return(head);
}
             /* USE FUNCTION find() HERE */
```

Fig. 8.9 *A function for deleting an item from linked list*

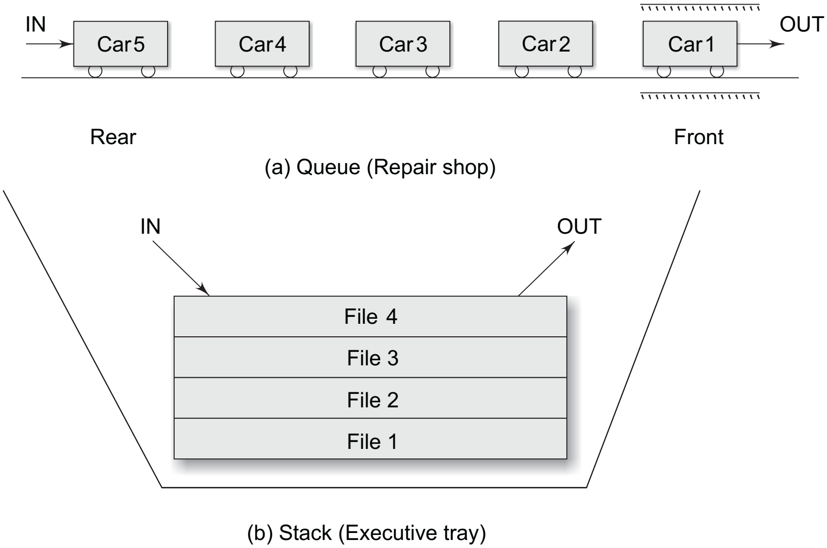
APPLICATION OF LINKED LISTS

Linked list concepts are useful to model many different abstract data types such as queues, stacks and trees.

If we restrict the process of insertion to one end of the list and deletions to the other end, then we have a model of a *queue*. That is, we can insert an item at the rear and remove an item at the front (see Fig. 8.10a). This obeys the discipline of "first in, first out" (FIFO). There are many examples of queues in real-life applications.

If we restrict insertions and deletions to occur only at the beginning of list, then we model another data structure known as *stack*. Stacks are also referred to as *push-down* lists. An example of a stack is the "in" tray of a busy executive. The files pile up in the tray, and whenever the executive has time to clear the files, he takes it off from the top. That is, files are added at the top and removed from the top (see Fig. 8.10b). Stacks are sometimes referred to as "last in, first out" (LIFO) structure.

Lists, queues and stacks are all inherently one-dimensional. A *tree* represents a two-dimensional linked list. Trees are frequently encountered in everyday life. One example is the organizational chart of a large company. Another example is the chart of sports tournaments.



(a) Queue (Repair shop)

(b) Stack (Executive tray)

Fig. 8.10 *Application of linked lists*

KEY CONCEPTS

- **DYNAMIC MEMORY ALLOCATION:** It is the process of allocating memory space at run time. **[LO 8.1]**
- **HEAP:** It is the free memory space that is available for dynamic allocation during program execution. **[LO 8.1]**
- **STACK:** It is a push-down list that stores the elements based on last in, first out (LIFO) principle. **[LO 8.1]**
- **CALLOC FUNCTION:** It is a function that allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory. **[LO 8.2]**

- **MALLOC FUNCTION:** It is a function that allocates requested size of bytes and returns a pointer to the first byte of the allocated space. **[LO 8.2]**
- **SIZE OF OPERATOR:** It is used to determine the size of a data type. **[LO 8.2]**
- **REALLOC FUNCTION:** It is a function that modifies the previously allocated memory space. **[LO 8.3]**
- **LINKED LIST:** It is a collection of structures ordered not by their physical placement in memory but by the logical links that are stored as a part of the data in the structure itself. **[LO 8.4]**
- **NULL POINTER:** It is a collection of structures ordered not by their physical placement in memory but by the logical links that are stored as a part of the data in the structure itself. **[LO 8.4]**

ALWAYS REMEMBER

- Release the dynamically allocated memory when it is no longer required to avoid any possible "memory leak". **[LO 8.1]**
- Use the **sizeof** operator to determine the size of a linked list. **[LO 8.2]**
- When using memory allocation functions **malloc** and **calloc,** test for a NULL pointer return value. Print appropriate message if the memory allocation fails. **[LO 8.2]**
- Never call memory allocation functions with a zero size. **[LO 8.2]**
- It is an error to assign the return value from **malloc** or **calloc** to anything other than a pointer. **[LO 8.2]**
- It is an error to release individually the elements of an array created with **calloc**. **[LO 8.2]**
- Using **free** function to release the memory not allocated dynamically with **malloc** or **calloc** is an error. **[LO 8.3]**
- Use of a invalid pointer with **free** may cause problems and, sometimes, system crash. **[LO 8.3]**
- Using a pointer after its memory has been released is an error. **[LO 8.3]**
- It is a logic error to set a pointer to NULL before the node has been released. The node is irretrievably lost. **[LO 8.3]**
- It is an error to declare a self-referential structure without a structure tag. **[LO 8.4]**
- It is a logic error to fail to set the link field in the last node to null. **[LO 8.5]**

BRIEF CASES

1. Insertion in a Sorted List [LO 8.2, 8.5 M]

The task of inserting a value into the current location in a sorted linked list involves two operations:

1. Finding the node before which the new node has to be inserted. We call this node as 'Key node'.
2. Creating a new node with the value to be inserted and inserting the new node by manipulating pointers appropriately.

In order to illustrate the process of insertion, we use a sorted linked list created by the create function discussed in Worked-Out Problem 8.3. Figure 8.11 shows a complete program that creates a list (using sorted input data) and then inserts a given value into the correct place using function insert.

Program

```
#include <stdio.h>
#include <stdio.h>
#define NULL 0

struct linked_list
{
     int number;
     struct linked-list *next;
};
typedef struct linked_lit node;

main()
{
     int n;
     node *head;
     void create(node *p);
     node *insert(node *p, int n);
     void print(node *p);
     head = (node *)malloc(sizeof(node));
     create(head);
     printf("\n");
     printf("Original list: ");
     print(head);
     printf("\n\n");
     printf("Input number to be inserted: ");
     scanf("%d", &n);
     head = inert(head,n);
     printf("\n");
     printf("New list:  ");
     print(head);
}
void create(node *list)
{
     printf("Input a number \n");
     printf("(type –999 at end): ");
     scanf("%d", &list–>number);

     if(list–>number == –999)
     {
        list–>next = NULL;
     }
     else /* create next node */
     {
          list–>next = (node *)malloc(sizeof(node));
          create(list–>next);
     }
     return:
}
```

```
void print(node *list)
{
        if(list->next != NULL)
        {
           printf("%d -->", list->number);

        if(list ->next->next = = NULL)
           printf("%d", list->next->number);

        print(list->next);
        }
        return:
}
node *insert(node *head, int x)
{
        node *p1, *p2, *p;
        p1 = NULL;
        p2 = head;  /* p2 points to first node */

        for( ; p2->number < x; p2 = p2->next)
        {
           p1 = p2;

           if(p2->next->next == NULL)
              {
                 p2 = p2->next;   /* insertion at end */
                 break;
        }
   }

   /*key node found and insert new node */

   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)
      head = p; /* new node becomes the first node */
   else
      p1->next = p; /* new node inserted in middle */

      return (head);
}
```

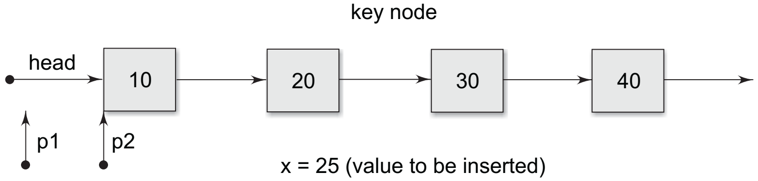
Output

```
            Input a number
            (type –999 at end ); 10

            Input a number
            (type –999 at end ); 20

            Input a number
            (type –999 at end ); 30

            Input a number
            (type –999 at end ); 40

            Input a number
            (type –999 at end ); -999

            Original list:  10 ––>20––>30––>40––>–999

            Input number to be inserted: 25
            New list: 10––>20––>25––>30––>40––>–999
```

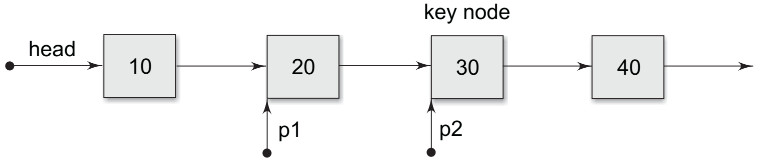
Fig. 8.11 *Inserting a number in a sorted linked list*

The function takes two arguments, one the value to be inserted and the other a pointer to the linked list. The function uses two pointers, **p1** and **p2** to search the list. Both the pointers are moved down the list with **p1** trailing **p2** by one node while the value **p2** points to is compared with the value to be inserted. The 'key node' is found when the number **p2** points to is greater (or equal) to the number to be inserted.

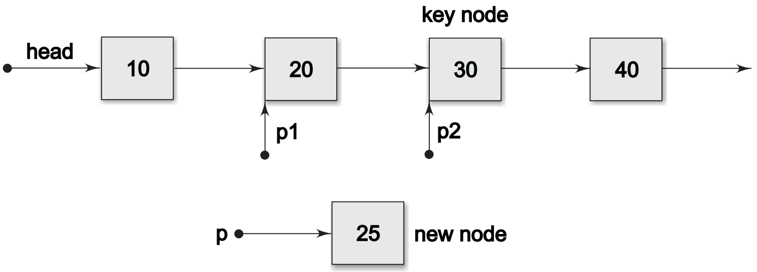
Once the key node is found, a new node containing the number is created and inserted between the nodes pointed to by **p1** and **p2**. The figures below illustrate the entire process.



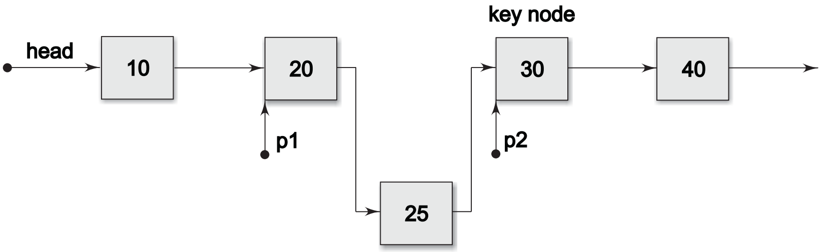
At the start of the search



When key node is found



When new node is created



When new node is inserted

2. Building a Sorted List [LO 8.2, 8.5 H]

The program in Fig. 8.11 can be used to create a sorted list. This is possible by creating 'one item' list using the create function and then inserting the remaining items one after another using insert function.

A new program that would build a sorted list from a given list of numbers is shown in Fig. 8.12. The **main** function creates a 'base node' using the first number in the list and then calls the function **insert_sort** repeatedly to build the entire sorted list. It uses the same sorting algorithm discussed above but does not use any dummy node. Note that the last item points to NULL.

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
```

Fig. 8.12 *Creation of sorted list from a given list of numbers*

REVIEW QUESTIONS

8.1 State whether the following statements are *true* or *false*

(a) Dynamically allocated memory can only be accessed using pointers. **[LO 8.1 M]**

(b) **calloc** is used to change the memory allocation previously allocated with **malloc**. **[LO 8.2 E]**

(c) Memory should be freed when it is no longer required. **[LO 8.3 E]**

(d) To ensure that it is released, allocated memory should be freed before the program ends. **[LO 8.3 E]**

(e) Only one call to free is necessary to release an entire array allocated with **calloc**. **[LO 8.3 M]**

(f) The link field in a linked list always points to successor. **[LO 8.4 E]**

(g) The first step in a adding a node to a linked list is to allocate memory for the next node. **[LO 8.5 M]**

8.2 Fill in the blanks in the following statements

(a) A_______________identifies the last logical node in a linked list. **[LO 8.4 M]**

(b) Function _______________ is used to dynamically allocate memory to arrays. **[LO 8.2 E]**

(c) A________________ is an ordered collection of data in which each element contains the location of the next element. **[LO 8.4 E]**

(d) Stacks are referred to as _______________. **[LO 8.1 M]**

(e) Data structures which contain a member field that points to the same structure type are called __________________ structures. **[LO 8.4 M]**

8.3 What is a linked list? How is it represented? **[LO 8.4 E]**

8.4 What is dynamic memory allocation? How does it help in building complex programs? **[LO 8.1 E]**

8.5 What is the principal difference between the functions **malloc** and **calloc**? **[LO 8.2 M]**

8.6 Why a linked list is called a dynamic data structure? What are the advantages of using linked lists over arrays? **[LO 8.4 M]**

8.7 Describe different types of linked lists. **[LO 8.4 E]**

8.8 The following code is defined in a header file *list.h* **[LO 8.5 H]**

```
typedef struct
{
     char name[15];
     int age;
     float weight;
}DATA;
struct linked_list
{
     DATA person;
     Struct linked_list *next;
};
     typedef struct linked_list NODE;
     typedef NODE *NDPTR;
```

Explain how could we use this header file for writing programs.

8.9 What does the following code achieve? **[LO 8.2 E]**

```
int * p ;
p = malloc (sizeof (int) ) ;
```

8.10 What does the following code do? **[LO 8.2 E]**

```
float *p;
p = calloc (10,sizeof(float) ) ;
```

8.11 What is the output of the following code? **[LO 8.2 M]**

```
int i, *ip ;
ip = calloc ( 4, sizeof(int) );
for (i = 0 ; i < 4 ; i++)
          * ip++ = i * i;
for (i = 0 ; i < 4 ; i++)
          printf("%d\n", *–ip );
```

8.12 What is printed by the following code? **[LO 8.2 M]**

```
int *p;
 p =  malloc (sizeof (int) );
*p =  100 ;
 p =  malloc (sizeof (int) );
*p =  111;
 printf("%d", *p);
```

8.13 What is the output of the following segment? **[LO 8.5 M]**

```
struct node
{
   int m ;
   struct node *next;
}  x, y, z, *p;
   x.m = 10 ;
   y.m = 20 ;
   z.m = 30 ;
   x.next = &y;
   y.next = &z;
   z.next = NULL;
   p = x.next;
   while (p != NULL)
{
   printf("%d\n", p -> m);
   p = p -> next;
}
```

DEBUGGING EXERCISES

8.1 Find errors, if any, in the following memory management statements:

(a) `*ptr = (int *)malloc(m, sizeof(int));` **[LO 8.2 M]**

(b) `table = (float *)calloc(100);` **[LO 8.2 M]**

(c) `node = free(ptr);` **[LO 8.3 M]**

8.2 Identify errors, if any, in the following structure definition statements: **[LO 8.5 E]**

```
struct
{
     char name[30]
     struct *next;
};
typedef struct node;
```

PROGRAMMING EXERCISES

8.1 In Worked-Out Problem 8.3, we have used print() in recursive mode. Rewrite this function using iterative technique in for loop. **[LO 8.5 M]**

8.2 Write a menu driven program to create a linked list of a class of students and perform the following operations: **[LO 8.5 H]**

(a) Write out the contents of the list.

(b) Edit the details of a specified student.

(c) Count the number of students above a specified age and weight.

Make use of the header file defined in Review Question 8.8.

8.3 Write recursive and non-recursive functions for reversing the elements in a linear list. Compare the relative efficiencies of them. **[LO 8.5 M]**

8.4 Write an interactive program to create linear linked lists of customer names and their telephone numbers. The program should be menu driven and include features for add ing a new customer and deleting an existing customer. **[LO 8.5 M]**

8.5 Modify the above program so that the list is always maintained in the alphabetical order of customer names. **[LO 8.5 H]**

8.6 Develop a program to combine two sorted lists to produce a third sorted lists which contains one occurrence of each of the elements in the original lists. **[LO 8.5 M]**

8.7 Write a program to create a circular linked list so that the input order of data item is maintained. Add function to carry out the following operations on circular linked list. **[LO 8.5 H]**

(a) Count the number of nodes

(b) Write out contents

(c) Locate and write the contents of a given node

8.8 Write a program to construct an ordered doubly linked list and write out the contents of a specified node. **[LO 8.5 M]**

8.9 Write a function that would traverse a linear singly linked list in reverse and write out the contents in reverse order. **[LO 8.5 M]**

8.10 Given two ordered singly linked lists, write a function that will merge them into a third ordered list. **[LO 8.5 M]**

8.11 Write a function that takes a pointer to the first node in a linked list as a parameter and returns a pointer to the last node. NULL should be returned if the list is empty. **[LO 8.5 M]**

8.12 Write a function that counts and returns the total number of nodes in a linked list. **[LO 8.5 E]**

8.13 Write a function that takes a specified node of a linked list and makes it as its last node. **[LO 8.5 M]**

8.14 Write a function that computers and returns the length of a circular list. **[LO 8.5 E]**

8.15 Write functions to implement the following tasks for a doubly linked list. **[LO 8.5 H]**

(a) To insert a node.

(b) To delete a node.

(c) To find a specified node.

File Management

Chapter 9

LEARNING OBJECTIVES

LO 9.1 Describe opening and closing of files

LO 9.2 Discuss input/output operations on files

LO 9.3 Determine how error handling is performed during I/O operations

LO 9.4 Explain random access to files

LO 9.5 Know the command line arguments

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.

DEFINING AND OPENING A FILE

LO 9.1
Describe opening and closing of files

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.

CLOSING A FILE

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form:

```
fclose(file_pointer);
```

This would close the file associated with the **FILE** pointer **file_pointer.** Look at the following segment of a program.

```
.....
.....
FILE *p1, *p2;
p1 = fopen("INPUT", "w");
p2 = fopen("OUTPUT", "r");
.....
.....
fclose(p1);
fclose(p2);
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

As a matter of fact all files are closed automatically whenever a program terminates. However, closing a file as soon as you are done with it is a good programming habit.

INPUT/OUTPUT OPERATIONS ON FILES

LO 9.2
Discuss 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.

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.

WORKED-OUT PROBLEM 9.1 E

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 *f1;
     char c;
     printf("Data Input\n\n");
     /* Open the file INPUT */
     f1 = fopen("INPUT", "w");
```

E for Easy, **M** for Medium and **H** for High

```
                    /* 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.

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.

WORKED-OUT PROBLEM 9.2

E

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. 9.2. It uses three files simultaneously and therefore, we need to define three-file pointers **f1, f2** and **f3.**

First, the file DATA containing integer values is created. The integer values are read from the terminal and are written to the file **DATA** with the help of the statement

putw(number, f1);

Notice that when we type –1, the reading is terminated and the file is closed. The next step is to open all the three files, **DATA** for reading, **ODD** and **EVEN** for writing. The contents of **DATA** file are read, integer by integer, by the function **getw(f1)** and written to **ODD** or **EVEN** file after an appropriate test. Note that the statement

(number = getw(f1)) != EOF

reads a value, assigns the same to **number**, and then tests for the end-of-file mark.

Finally, the program displays the contents of ODD and EVEN files. It is important to note that the files **ODD** and **EVEN** opened for writing are closed before they are reopened for reading.

Program

```
#include  <stdio.h>
main()
{
    FILE  *f1, *f2, *f3;
    int   number, i;

    printf("Contents of DATA file\n\n");
    f1 = fopen("DATA", "w");      /* Create DATA file    */
    for(i = 1; i <= 30; i++)
    {
        scanf("%d", &number);
        if(number == -1) break;
        putw(number,f1);
    }
    fclose(f1);

    f1 = fopen("DATA", "r");
    f2 = fopen("ODD", "w");
    f3 = fopen("EVEN", "w");

    /* Read from DATA file */
    while((number = getw(f1)) != EOF)
    {
        if(number %2 == 0)
          putw(number, f3);   /*  Write to EVEN file  */
        else
          putw(number, f2);   /*  Write to ODD file   */
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);

    f2 = fopen("ODD","r");
    f3 = fopen("EVEN", "r");

    printf("\n\nContents of ODD file\n\n");
    while((number = getw(f2)) != EOF)
      printf("%4d", number);

    printf("\n\nContents of EVEN file\n\n");
    while((number = getw(f3)) != EOF)
      printf("%4d", number);

    fclose(f2);
    fclose(f3);

}
```

Output

```
Contents of DATA file
111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 –1

Contents of ODD file
111 333 555 777 999 121 343 565

Contents of EVEN file
222 444 666 888   0 232 454
```

Fig. 9.2 *Operations on integer data*

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 familar **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

```
fprintf(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.**

WORKED-OUT PROBLEM 9.3

H

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. 9.3. The filename INVENTORY is supplied through the keyboard. Data is read using the function **fscanf** from the file **stdin,** which refers to the terminal and it is then written to the file that is being pointed to by the file pointer **fp.** Remember that the file pointer **fp** points to the file INVENTORY.

After closing the file INVENTORY, it is again reopened for reading. The data from the file, along with the item values are written to the file **stdout,** which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file.…

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. 9.3 *Operations on mixed data types*

ERROR HANDLING DURING I/O OPERATIONS

LO 9.3
Determine how error handling is performed 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");
```

WORKED-OUT PROBLEM 9.4

E

Write a program to illustrate error handling in file operations.

The program shown in Fig. 9.4 illustrates the use of the **NULL** pointer test and **feof** function. When we input filename as TETS, the function call

```
fopen("TETS", "r");
```

returns a **NULL** pointer because the file TETS does not exist and therefore the message “Cannot open the file” is printed out.

Similarly, the call **feof(fp2)** returns a non-zero integer when the entire data has been read, and hence the program prints the message “Ran out of data” and terminates further reading.

Program

```
#include  <stdio.h>

main()
```

```
        {
             char  *filename;
             FILE  *fp1, *fp2;
             int   i, number;

             fp1 = fopen("TEST", "w");
             for(i = 10; i <= 100; i += 10)
                  putw(i, fp1);

             fclose(fp1);

             printf("\nInput filename\n");

        open_file:
        scanf("%s", filename);

        if((fp2 = fopen(filename,"r")) == NULL)
        {
             printf("Cannot open the file.\n");
             printf("Type filename again.\n\n");
             goto open_file;
        }
        else

        for(i = 1; i <= 20; i++)
        {  number = getw(fp2);
           if(feof(fp2))
           {
             printf("\nRan out of data.\n");
             break;
           }
           else
             printf("%d\n", number);
        }

        fclose(fp2);
    }
```

Output

```
        Input filename
        TEST
```

```
Cannot open the file.
Type filename again.

TEST
10
20
30
40
50
60
70
80
90
100

Ran out of data.
```

Fig. 9.4 *Illustration of error handling in file operations*

RANDOM ACCESS TO FILES

LO 9.4
Explain 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*
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

When the operation is successful, **fseek** returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and **fseek** returns –1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

WORKED-OUT PROBLEM 9.5

E

Write a program that uses the functions **ftell** and **fseek**.

A program employing **ftell** and **fseek** functions is shown in Fig. 9.5. We have created a file **RANDOM** with the following contents:

Position ––––>	0	1	2	. . .	25
Character stored ––––>	A	B	C	. . .	Z

We are reading the file twice. First, we are reading the content of every fifth position and printing its value along with its position on the screen. The second time, we are reading the contents of the file from the end and printing the same on the screen.

During the first reading, the file pointer crosses the end-of-file mark when the parameter **n** of **fseek(fp,n,0)** becomes 30. Therefore, after printing the content of position 30, the loop is terminated.

For reading the file from the end, we use the statement

```
fseek(fp,–1L,2);
```

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This is achieved by the function

```
fseek(fp, –2L, 1);
```

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

Program

```
#include <stdio.h>
main()
{
    FILE  *fp;
    long  n;
    char c;
    fp = fopen("RANDOM", "w");
    while((c = getchar()) != EOF)
        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^Z
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. 9.5 *Illustration of **fseek** and **ftell** functions*

WORKED-OUT PROBLEM 9.6

M

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. 9.6. It uses a structure definition to describe each item and a function **append()** to add an item to the file.

On execution, the program requests for the filename to which data is to be appended. After appending the items, the position of the last character in the file is assigned to **n** and then the file is closed.

The file is reopened for reading and its contents are displayed. Note that reading and displaying are done under the control of a **while** loop. The loop tests the current file position against **n** and is terminated when they become equal.

Program

```
#include  <stdio.h>

struct invent_record
{
     char   name[10];
     int    number;
     float  price;
     int    quantity;
};
main()
{
     struct invent_record item;
     char  filename[10];
     int   response;
     FILE  *fp;
     long  n;
     void append (struct invent_record *x, file *y);
     printf("Type filename:");
     scanf("%s", filename);

     fp = fopen(filename, "a+");
     do
   {
        append(&item, fp);
        printf("\nItem %s appended.\n",item.name);
        printf("\nDo you want to add another item\
           (1 for YES /0 for NO)?");
        scanf("%d", &response);
   }    while (response == 1);

   n = ftell(fp);      /* Position of last character  */
   fclose(fp);

   fp = fopen(filename, "r");

   while(ftell(fp) < n)
   {
           fscanf(fp,"%s %d %f %d",
           item.name, &item.number, &item.price, &item.quantity);
           fprintf(stdout,"%-8s %7d %8.2f %8d\n",
           item.name, item.number, item.price, item.quantity);
   }
   fclose(fp);
}
void append(struct invent_record *product, File *ptr)
{
           printf("Item name:");
           scanf("%s", product->name);
           printf("Item number:");
           scanf("%d", &product->number);
           printf("Item price:");
           scanf("%f", &product->price);
           printf("Quantity:");
           scanf("%d", &product->quantity);
           fprintf(ptr, "%s %d %.2f %d",
                          product->name,
                          product->number,
                          product->price,
                          product->quantity);
}
```

Output

```
Type filename:INVENTORY
Item name:XXX
Item number:444
Item price:40.50
Quantity:34
Item XXX appended.
Do you want to add another item(1 for YES /0 for NO)?1
Item name:YYY
Item number:555
Item price:50.50
Quantity:45
Item YYY appended.
Do you want to add another item(1 for YES /0 for NO)?0
AAA-1        111      17.50      115
BBB-2        125      36.00       75
C-3          247      31.75      104
XXX          444      40.50       34
YYY          555      50.50       45
```

Fig. 9.6 *Adding items to an existing file*

WORKED-OUT PROBLEM 9.7

H

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. 9.7 *Program to reverse n characters in a file*

COMMAND LINE ARGUMENTS

LO 9.5
Know the command line arguments

What is a command line argument? It is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process. For example, if we want to execute a program to copy the contents of a file named **X_FILE** to another one named Y_FILE, then we may use a command line like

C > PROGRAM X_FILE Y_FILE

where **PROGRAM** is the filename where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the filenames during execution. How do these parameters get into the program?

We know that every C program should have one **main** function and that it marks the beginning of the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact **main** can take two arguments called **argc** and **argv** and the information contained in the command line is passed on to the program through these arguments, when **main** is called up by the system.

The variable **argc** is an argument counter that counts the number of arguments on the command line. The **argv** is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line given above, **argc** is three and **argv** is an array of three pointers to strings as shown below:

argv[0] –> PROGRAM
argv[1] –> X_FILE
argv[2] –> Y_FILE

In order to access the command line arguments, we must declare the main function and its parameters as follows:

```
main(int arge, char *argv[])
{
        .....
        .....
}
```

The first parameter in the command line is always the program name and therefore **argv[0]** always represents the program name.

WORKED-OUT PROBLEM 9.8

E

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 9.8 shows the use of command line arguments. The command line is

F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

Each word in the command line is an argument to the **main** and therefore the total number of arguments is 9.

The argument vector **argv[1]** points to the string TEXT and therefore the statement

```
fp = fopen(argv[1], "w");
```

opens a file with the name TEXT. The **for** loop that follows immediately writes the remaining 7 arguments to the file TEXT.

Program

```
  #include  <stdio.h>

  main(int arge, 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>F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGG

  No. of arguments in Command line = 9

  Contents of TEXT file
```

```
AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

C:\C\F12_7.EXE
 TEXT
    AAAAAA
         BBBBBB
              CCCCCC
                   DDDDDD
                        EEEEEE
                             FFFFFF
                                  GGGGGG
```

Fig. 9.8 *Use of command line arguments*

KEY CONCEPTS

- **FILENAME:** It is a string of characters that make up a valid filename for the operating system. **[LO 9.1]**
- **FSEEK:** It is a function that sets the position to a desired point in the file. **[LO 9.4]**
- **FTELL:** It is a function that returns the current position in the file. **[LO 9.4]**
- **REWIND:** It is a function that sets the position to the beginning of the file. **[LO 9.4]**
- **COMMAND LINE ARGUMENT:** It is a parameter supplied to a program from command prompt when the program is invoked. **[LO 9.5]**

ALWAYS REMEMBER

- Do not try to use a file before opening it. **[LO 9.1]**
- Remember, when an existing file is open using 'w' mode, the contents of file are deleted. **[LO 9.1]**
- When a file is used for both reading and writing, we must open it in 'w+' mode. **[LO 9.1]**
- It is an error to omit the file pointer when using a file function. **[LO 9.1]**
- It is an error to open a file for reading when it does not exist. **[LO 9.1]**
- It is an error to access a file with its name rather than its file pointer. **[LO 9.1]**
- It is a good practice to close all files before terminating a program. **[LO 9.1]**
- EOF is integer type with a value –1. Therefore, we must use an integer variable to test EOF. **[LO 9.2]**
- It is an error to try to read from a file that is in write mode and vice versa. **[LO 9.2]**
- 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. **[LO 9.3]**
- It is an error to attempt to place the file marker before the first byte of a file. **[LO 9.4]**
- It is a good practice to check the return value of fseek function every time it is invoked. A positive return value ensures that the file pointer is within the file boundaries. **[LO 9.5]**

REVIEW QUESTIONS

9.1 State whether the following statements are *true* or *false*.

(a) A file must be opened before it can be used. **[LO 9.1 E]**

(b) All files must be explicitly closed. **[LO 9.1 E]**

(c) Files are always referred to by name in C programs. **[LO 9.1 E]**

(d) Using **fseek** to position a file beyond the end of the file is an error. **[LO 9.1 M]**

(e) Function **fseek** may be used to seek from the beginning of the file only. **[LO 9.4 E]**

9.2 Fill in the blanks in the following statements.

(a) The mode _______ is used for opening a file for updating. **[LO 9.1 E]**

(b) The function _____ is used to write data to randomly accessed file. **[LO 9.2 M]**

(c) The function ____gives the current position in the file. **[LO 9.4 E]**

(d) The function _______ may be used to position a file at the beginning. **[LO 9.4 M]**

9.3 Describe the use and limitations of the functions **getc** and **putc. [LO 9.2 E]**

9.4 What is the significance of EOF? **[LO 9.2 M]**

9.5 When a program is terminated, all the files used by it are automatically closed. Why is it then necessary to close a file during execution of the program? **[LO 9.1 H]**

9.6 Distinguish between the following functions:

(a) getc and getchar **[LO 9.2 E]**

(b) printf and fprintf **[LO 9.2 E]**

(c) feof and ferror **[LO 9.3 M]**

9.7 How does an append mode differ from a write mode? **[LO 9.1 M]**

9.8 What are the common uses of **rewind** and **ftell** functions? **[LO 9.4 E]**

9.9 Explain the general format of **fseek** function? **[LO 9.4 E]**

9.10 What is the difference between the statements **rewind(fp);** and **fseek(fp,0L,0);? [LO 9.4 M]**

9.11 What does the following statement mean? **[LO 9.1 H]**

```
FILE(*p) (void)
```

9.12 What does the following statement do? **[LO 9.2 M]**

```
while ( (c = getchar( ) != EOF )
         putc(c, fl);
```

9.13 What does the following statement do? **[LO 9.2 M]**

```
while ( (m = getw(fl) ) != EOF)
        printf("%5d", m);
```

9.14 What does the following segment do? **[LO 9.2 H]**

```
. . . .
for (i = 1; i <= 5; i++ )
{
     fscanf(stdin, "%s", name);
     fprintf(fp, "%s", name);
}
. . . .
```

9.15 What is the purpose of the following functions? **[LO 9.3 E]**

(a) feof ()

(b) ferror ( )

9.16 Give examples of using **feof** and **ferror** in a program. **[LO 9.3 M]**

9.17 Can we read from a file and write to the same file without resetting the file pointer? If not, why? **[LO 9.2 H]**

9.18 When do we use the following functions? **[LO 9.4 M]**

(a) free ( )

(b) rewind ( )

9.19 Describe an algorithm that will append the contents of one file to the end of another file. **[LO 9.1 E]**

DEBUGGING EXERCISES

9.1 Find error, if any, in the following statements: **[LO 9.1 M]**

```
FILE fptr;
fptr = fopen ("data", "a+");
```

PROGRAMMING EXERCISES

9.1 Write a program to copy the contents of one file into another. **[LO 9.2 E]**

9.2 Two files DATA1 and DATA2 contain sorted lists of integers. Write a program to produce a third file DATA which holds a single sorted, merged list of these two lists. Use command line arguments to specify the file names. **[LO 9.2 M]**

9.3 Write a program that compares two files and returns 0 if they are equal and 1 is they are not. **[LO 9.2 E]**

9.4 Write a program that appends one file at the end of another. **[LO 9.4 E]**

9.5 Write a program that reads a file containing integers and appends at its end the sum of all the integers. **[LO 9.4 M]**

9.6 Write a program that prompts the user for two files, one containing a line of text known as source file and other, an empty file known as target file and then copies the contents of source file into target file. **[LO 9.1, 9.2, 9.5 H]**

Modify the program so that a specified character is deleted from the source file as it is copied to the target file.

9.7 Write a program that requests for a file name and an integer, known as offset value. The program then reads the file starting from the location specified by the offset value and prints the contents on the screen. **[LO 9.4 M]**

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.

9.8 Write a program to create a sequential file that could store details about five products. Details include product code, cost and number of items available and are provided through keyboard. **[LO 9.2 M]**

9.9 Write a program to read the file created in Exercise 9.8 and compute and print the total value of all the five products. **[LO 9.2 M]**

9.10 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. **[LO 9.2, 9.4 H]**