

R S Bichkar

# Programming with





R S Bichkar

# Programming with



 Universities Press

# Programming with C

*For our entire range of books please use search strings "**Orient BlackSwan**", "**Universities Press India**" and "**Permanent Black**" in store.*

# Programming with C

**R. S. Bichkar**

Professor (Department of E&TC Engg)  
G. H. Raisoni College of Engineering and Management  
Pune



**Universities Press**

## **PROGRAMMING WITH C**

---

### **Universities Press (India) Private Limited**

*Registered Office*

3-6-747/1/A & 3-6-754/1, Himayatnagar, Hyderabad 500 029 (A.P.), INDIA

e-mail: [info@universitiespress.com](mailto:info@universitiespress.com)

*Distributed by*

Orient Blackswan Private Limited

*Registered Office*

3-6-752 Himayatnagar, Hyderabad 500 029 (A.P.), INDIA

e-mail: [info@orientblackswan.com](mailto:info@orientblackswan.com)

*Other Offices*

Bengaluru, Bhopal, Bhubaneshwar, Chennai, Ernakulam, Guwahati, Hyderabad,  
Jaipur, Kolkata, Lucknow, Mumbai, New Delhi, Noida, Patna, Vijayawada

© Universities Press (India) Private Limited 2011

First published 2012

eISBN 978-81-7371-779-6

e-edition: First Published 2017

ePUB Conversion: [TEXTSOFT Solutions Pvt. Ltd.](#)

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests write to the publisher.

*To my beloved parents*

# Contents

## Preface

### **1 Introduction to Computers, Programming and the C Language**

1.1 Computer Architecture

1.2 Program Development Process

1.3 Structured and Modular Programming

1.4 Flowcharts

    1.4.1 Flowchart Symbols

    1.4.2 Flowcharts for Sequencing

    1.4.3 Flowcharts for Selection

    1.4.4 Flowcharts for Loops

1.5 The C Programming Language

    1.5.1 C Language Standards

    1.5.2 C Programs

1.6 Program Development Environments

    1.6.1 Turbo C/C++

    1.6.2 Dev-C++

    1.6.3 Code::Blocks

1.7 Advanced Concepts

    1.7.1 A Quick Tour of the C Language

    1.7.2 Main Memory Organization

    1.7.3 Binary Number System

Exercises

Exercises (Advanced Concepts)

### **2 Representing Data**

2.1 Character Set

[2.2 Keywords](#)

[2.3 Basic Data Types](#)

[2.4 Constants](#)

[2.4.1 Integer Constants](#)

[2.4.2 Floating Constants](#)

[2.4.3 Character Constants](#)

[2.5 String Literals](#)

[2.6 Identifiers](#)

[2.7 Variables](#)

[2.7.1 Variable Declaration](#)

[2.7.2 Variable Initialization](#)

[2.8 Symbolic Constants](#)

[2.9 Advanced Concepts](#)

[2.9.1 Type Suffixes](#)

[2.9.2 Type Modifiers](#)

[2.9.3 Type Qualifiers](#)

[2.9.4 Character Strings](#)

[2.9.5 Constants in Octal and Hexadecimal Form](#)

[Exercises](#)

[Exercises \(Advanced Concepts\)](#)

### [3 Arithmetic Operators and Expressions](#)

[3.1 Operators](#)

[3.1.1 Arithmetic Operators](#)

[3.1.2 Precedence and Associativity of Operators](#)

[3.2 Expressions](#)

[3.2.1 Simple Arithmetic Expressions](#)

[3.2.2 Parenthesized Expressions](#)

[3.2.3 Initializing Variables Using Expressions](#)

[3.2.4 Constant Expressions](#)

[3.2.5 Avoiding Common Mistakes While Writing C Expressions](#)

[3.3 Assignment](#)

- [3.3.1 Simple Assignment Expressions](#)
  - [3.3.2 Simple Assignment Statement](#)
  - [3.3.3 Compound Assignments](#)
  - [3.3.4 Nested Assignments](#)
  - [3.3.5 Explicit Type Conversion](#)
  - [3.3.6 Assignments Containing Increment and Decrement Operators](#)
  - [3.4 Advanced Concepts](#)
    - [3.4.1 Expressions](#)
    - [3.4.2 Assignments](#)
    - [3.4.3 The Comma Operator](#)
    - [3.4.4 Operations on Strings: Element Access and Assignment](#)
  - [Exercises](#)
  - [Exercises \(Advanced Concepts\)](#)
- ## [4 The C Standard Library](#)
- [4.1 About the C Standard Library](#)
    - [4.1.1 Header Files](#)
    - [4.1.2 Functions](#)
  - [4.2 Input and Output Facilities](#)
    - [4.2.1 Formatted Output - the printf Function](#)
    - [4.2.2 Formatted Input - the scanf Function](#)
    - [4.2.3 Character I/O](#)
    - [4.2.4 String I/O - the gets and puts Functions](#)
  - [4.3 Mathematical Library](#)
    - [4.3.1 Powers and Logarithms](#)
    - [4.3.2 Trigonometric and Hyperbolic Functions](#)
    - [4.3.3 Other Mathematical Functions](#)
  - [4.4 Advanced Concepts](#)
    - [4.4.1 Streams](#)
    - [4.4.2 Function Prototypes](#)
    - [4.4.3 Formatted Output using the printf Function](#)
    - [4.4.4 Character Classification and Conversion](#)

[4.4.5 Manipulating Strings](#)

[4.4.6 Utility Functions](#)

[4.4.7 Miscellaneous Functions](#)

[Exercises](#)

[Exercises \(Advanced Concepts\)](#)

## **5 Conditional Control**

[5.1 Relational and Equality Operators](#)

[5.1.1 Relational and Equality Expressions](#)

[5.1.2 Evaluation of Relational and Equality Expressions](#)

[5.2 The if Statement](#)

[5.2.1 Using Block Statements in the if Statement](#)

[5.3 Logical Operators](#)

[5.3.1 Logical AND Operator](#)

[5.3.2 Logical OR Operator](#)

[5.3.3 Logical NOT Operator](#)

[5.3.4 Evaluation of Boolean Expressions](#)

[5.4 Character Test, Classification and Conversion](#)

[5.5 The switch Statement](#)

[5.6 Conditional Expression Operator \(?:\)](#)

[5.7 Advanced Concepts](#)

[5.7.1 Common Mistakes in if Statements](#)

[5.7.2 Common Mistakes in Boolean Expressions](#)

[5.7.3 Alternative Forms of Boolean Expressions](#)

[Exercises](#)

[Exercises \(Advanced Concepts\)](#)

## **6 Looping Control**

[6.1 The for Loop](#)

[6.1.1 Using Compound Statements in the for Loop](#)

[6.2 The while Loop](#)

[6.2.1 The while Loop for Known Number of Iterations](#)

[6.2.2 Text Input Using a Loop and the getchar Function](#)

## [6.3 The do ... while Loop](#)

## [6.4 Advanced Concepts](#)

### [6.4.1 Variations in for Loops](#)

### [6.4.2 Variations in the while and do ... while Loops](#)

## [Exercises](#)

### [Exercises \(Advanced Concepts\)](#)

## [7 Nested Control Structures](#)

### [7.1 Nested if Statements](#)

#### [7.1.1 Two-Level Nested if Statements](#)

#### [7.1.2 Higher Level Nested if Statements](#)

#### [7.1.3 if-else-if Statement](#)

### [7.2 Using if Statements within Loops](#)

#### [7.2.1 Using if Statements within a for Loop](#)

#### [7.2.2 Using if Statements within while and do ... while Loops](#)

### [7.3 Nested Loops](#)

### [7.4 Nested Control Structures involving switch Statement](#)

### [7.5 Loop Interruption](#)

#### [7.5.1 The break Statement](#)

#### [7.5.2 The continue Statement](#)

### [7.6 Advanced Concepts](#)

#### [7.6.1 Logical Operators and Nested if Statements](#)

#### [7.6.2 Nested Conditional Expression Operators](#)

## [Exercises](#)

### [Exercises \(Advanced Concepts\)](#)

## [8 Functions](#)

### [8.1 About Functions](#)

#### [8.1.1 Function Call](#)

#### [8.1.2 Advantages of Functions](#)

### [8.2 User-defined Functions](#)

#### [8.2.1 Function Definition](#)

#### [8.2.2 The return Statement](#)

### [8.2.3 Function Declaration or Prototype](#)

### [8.3 Program Structure](#)

### [8.4 Methods of Parameter Passing](#)

### [8.5 Recursion](#)

### [8.6 Advanced Concepts](#)

#### [8.6.1 Function Parameter as a Loop Variable](#)

#### [8.6.2 const Parameters](#)

#### [8.6.3 Storage Classes](#)

### [Exercises](#)

### [Exercises \(Advanced Concepts\)](#)

## [9 Vectors or One-dimensional Arrays](#)

### [9.1 Introduction](#)

### [9.2 One-dimensional Arrays](#)

#### [9.2.1 Array Declaration](#)

#### [9.2.2 Accessing Array Elements](#)

#### [9.2.3 Operations on Array Elements](#)

#### [9.2.4 Operations on Entire Arrays](#)

#### [9.2.5 Array Initialization](#)

### [9.3 Arrays as Function Parameters](#)

### [9.4 Advanced Concepts](#)

#### [9.4.1 const Vectors](#)

#### [9.4.2 Static Arrays](#)

#### [9.4.3 External or Global Arrays](#)

### [Exercises](#)

### [Exercises \(Advanced Concepts\)](#)

## [10 Matrices and Multidimensional Arrays](#)

### [10.1 Two-dimensional Arrays or Matrices](#)

#### [10.1.1 Declaration](#)

#### [10.1.2 Accessing Matrix Elements](#)

#### [10.1.3 Operations on Matrix Elements](#)

#### [10.1.4 Operations on Entire Matrices](#)

[10.1.5 Initialization](#)

[10.1.6 Matrices as Function Parameters](#)

## [10.2 Multidimensional Arrays](#)

[10.2.1 Declaration](#)

[10.2.2 Element Access and Operations on Elements and Entire Arrays](#)

[10.2.3 Initialization](#)

[10.2.4 Multidimensional Arrays as Function Parameters](#)

## [10.3 Advanced Concepts](#)

[10.3.1 const, static and extern Arrays](#)

[10.3.2 Memory Allocation](#)

[10.3.3 Eliminate the Row and Column of a Square Matrix](#)

[10.3.4 Determinant of a Square Matrix](#)

[10.3.5 Cofactor of a Matrix](#)

[10.3.6 Inverse of a Matrix](#)

## [Exercises](#)

[Exercises \(Advanced Concepts\)](#)

# [11 Pointers](#)

## [11.1 Pointer Basics](#)

[11.1.1 What is a Pointer?](#)

[11.1.2 Declaring Pointer Variables](#)

[11.1.3 Address Operator \(&\) and Dereference Operator \(\\*\)](#)

[11.1.4 Pointer Assignment and Initialization](#)

[11.1.5 Simple Expressions Involving Pointers](#)

## [11.2 Call by Reference](#)

## [11.3 Vectors and Pointers](#)

[11.3.1 Operations with Pointers to Vector Elements](#)

[11.3.2 Accessing Vector Elements by Using Array Name as a Pointer](#)

[11.3.3 Accessing Vector Elements Using Another Pointer Variable](#)

[11.3.4 Passing a Vector to a Function Using a Pointer](#)

## [11.4 Advanced Concepts](#)

[11.4.1 The Typecast and sizeof Operators](#)

[11.4.2 Returning a Pointer from a Function](#)

[11.4.3 Matrices and Pointers](#)

[11.4.4 Multidimensional Arrays and Pointers](#)

[11.4.5 Pointer to a Pointer](#)

[11.4.6 Array of Pointers](#)

[11.4.7 Dynamic Memory Management](#)

[11.4.8 Dynamic Arrays](#)

[11.4.9 Pointer to a Function](#)

[11.4.10 Polymorphic Functions using void Pointer](#)

[11.4.11 Complex Declarations Involving Pointers](#)

[Exercises](#)

[Exercises \(Advanced Concepts\)](#)

## [12 Strings](#)

[12.1 Processing Strings](#)

[12.1.1 Processing Strings Using Loops](#)

[12.1.2 Writing Functions for String Processing](#)

[12.2 Library Functions for String Processing](#)

[12.2.1 ANSI C Standard Library Functions for String Processing](#)

[12.3 Advanced Concepts](#)

[12.3.1 Nesting String Manipulation Functions](#)

[12.3.2 Avoiding Pitfalls in String Processing](#)

[12.3.3 Working with Words in a String](#)

[12.3.4 ANSI Functions for Advanced String Processing](#)

[12.3.5 Non-ANSI Functions for String Processing](#)

[12.3.6 Dynamic Memory Allocation](#)

[12.3.7 Command-Line Arguments](#)

[Exercises](#)

[Exercises \(Advanced Concepts\)](#)

## [13 Structures](#)

[13.1 Structures](#)

[13.1.1 Declaring Structures](#)

- [13.1.2 Defining Structure Variables](#)
- [13.1.3 Accessing Structure Members - the Dot Operator](#)
- [13.1.4 Structure Initialization](#)
- [13.1.5 Structure Assignment](#)
- [13.1.6 Other Operations on Structures](#)
- [13.2 Structures and Functions](#)
  - [13.2.1 Passing a Structure to a Function](#)
  - [13.2.2 Structure as a Function Value](#)
- [13.3 Structure Containing Arrays](#)
  - [13.3.1 Declaring Structures Containing Arrays](#)
  - [13.3.2 Initializing Structures Containing Arrays](#)
  - [13.3.3 Accessing Member Arrays](#)
  - [13.3.4 Accessing Elements of Member Arrays](#)
  - [13.3.5 Using Structure Containing Arrays as Function Parameter and Return Value](#)
- [13.4 Nested Structures](#)
  - [13.4.1 Declaration of Nested Structures](#)
  - [13.4.2 Initializing Nested Structures](#)
  - [13.4.3 Processing Nested Structures](#)
- [13.5 Advanced Concepts](#)
  - [13.5.1 Memory Organization of Structures](#)
  - [13.5.2 Pointer to a Structure](#)
  - [13.5.3 Array of Structures](#)
  - [13.5.4 Representing Complex Data Using Arrays and Structures](#)
  - [13.5.5 Structure Containing Pointer Members](#)
- [Exercises](#)
  - [Exercises \(Advanced Concepts\)](#)

## 14 Files

- [14.1 File Basics](#)
  - [14.1.1 What is a File?](#)
  - [14.1.2 Streams](#)
  - [14.1.3 Standard Library Support for File Processing](#)

## [14.2 File Access Functions](#)

[14.2.1 The fopen Function](#)

[14.2.2 The fclose Function](#)

## [14.3 Character I/O](#)

[14.3.1 The fgetc Function and the getc Macro](#)

[14.3.2 The fputc Function and the putc Macro](#)

[14.3.3 The fgets and fputs Functions](#)

[14.3.4 The ungetc Function](#)

## [14.4 Formatted I/O](#)

### [14.5 Advanced Concepts](#)

[14.5.1 file Structure](#)

[14.5.2 Pagination Control in Display of Text Files](#)

[14.5.3 Direct Input/Output](#)

[14.5.4 File Positioning](#)

### [Exercises](#)

[Exercises \(Advanced Concepts\)](#)

## [15 Searching and Sorting](#)

### [15.1 Searching](#)

[15.1.1 Linear \(or Sequential\) Search](#)

[15.1.2 Binary Search](#)

[15.1.3 Recursive Binary Search](#)

### [15.2 Sorting](#)

[15.2.1 Bubble Sort](#)

[15.2.2 Selection Sort](#)

[15.2.3 Insertion Sort](#)

### [15.3 Advanced Concepts](#)

[15.3.1 Alternative/Efficient Implementations for Sorting Functions](#)

[15.3.2 Sorting Strings](#)

[15.3.3 Standard Library Functions for Searching and Sorting](#)

### [Exercises](#)

[Exercises \(Advanced Concepts\)](#)

## 16 Miscellaneous Concepts

[16.1 Bitwise Operators](#)

[16.2 Enumerated Types](#)

[16.2.1 Declaring Enumerated Types](#)

[16.2.2 Variables of Enumerated Types](#)

[16.2.3 Specifying Values for Enumerated Constants](#)

[16.2.4 Working with Enumerated Data](#)

[16.3 Renaming Types Using `typedef`](#)

[16.3.1 Using `typedef` with Structures](#)

[16.3.2 Using `typedef` with Enumerated Types](#)

[16.4 More on Library Functions and Facilities](#)

[16.4.1 The `printf` Function](#)

[16.4.2 `modf`, `frexp` and `ldexp` Functions](#)

[16.4.3 Ranges of Integral Data Types](#)

[16.4.4 Traditional Math Constants](#)

[16.5 Advanced Concepts](#)

[16.5.1 Bitwise Operators](#)

[16.5.2 Renaming Types with `typedef`](#)

[16.5.3 Unions](#)

[16.5.4 Writing Multi-File Programs](#)

[Exercises](#)

[Exercises \(Advanced Concepts\)](#)

## 17 Graphics in Turbo C and Turbo C++

[17.1 Preliminaries](#)

[17.1.1 Capabilities of the Graphics Mode](#)

[17.1.2 Graphics Support in TC/TC++](#)

[17.1.3 Pixels and Resolution](#)

[17.1.4 Colours and Palettes](#)

[17.1.5 Graphics View Ports and Pages](#)

[17.1.6 Graphics Adapters, Drivers and Modes](#)

[17.1.7 Graphics Coordinate System and CP](#)

[17.2 Invoking the Graphics System](#)

[17.3 Setting Colours and the Current Position \(CP\)](#)

[17.4 Drawing Graphics Entities](#)

[17.4.1 Drawing Points and Lines](#)

[17.4.2 Drawing Rectangles and Polygons](#)

[17.4.3 Drawing Circular and Elliptical Shapes](#)

[17.4.4 Setting Line Styles and Fill Styles](#)

[17.4.5 Drawing Filled Rectangles and Polygons](#)

[17.4.6 Drawing Filled Circular and Elliptical Objects](#)

[17.5 Displaying Text](#)

[17.5.1 Functions to Display Text](#)

[17.5.2 Setting Text Characteristics](#)

[17.5.3 Enquiring Text Attributes](#)

[17.6 Animation](#)

[17.6.1 Animation of Simple Objects](#)

[17.6.2 Animation of Complex Objects](#)

[Exercises](#)

[Appendix A. The ASCII Character Set](#)

[Appendix B. Summary of C Operators](#)

[Appendix C. Summary of C Statements](#)

[Appendix D. The C Standard Library](#)

[Appendix E. Turbo C](#)

# Preface

It has been my experience that students often find learning C an uphill task. This, more often than not, is because they get bogged down by details of the language that are not absolutely essential for a beginner to know in the learning stage, e. g., the large number of data types and their limits in one or more C implementations, format specifiers in scanf and printf functions, numerous operators and their precedence, numerous standard library functions, etc., that generally form a part of the essential learning material they are exposed to. Consequently, after studying C for several months, students may develop the feeling that programming with C is a very difficult task. On the contrary, programming in C can be made easy and interesting with proper guidance and an intelligent approach. In this book, I have adopted the approach which I believe is best suited to learning C—I invite you to use the book and discover for yourself that it works. I am confident that by the time you finish reading the book and have experimented with the further opportunities it offers, most of you will agree with me that learning C is indeed a rewarding experience, and that it need not be a frustrating one.

The Pareto's principle (the 80-20 rule), observed to be true for most situations, applies to programming languages as well, I feel, particularly with regard to the feature-rich aspect of the C language—a beginner need know only about 20 per cent of the language details, for all practical purposes, to make a comfortable beginning with programming. The remaining 80 per cent need be mastered only if one aims to become an expert programmer. Alternatively stated, 80 per cent of programs can be written using 20 per cent of the details of C. Kindly note that my use of the word '*details*' applies more to the kind of detail I referred to earlier in my opening paragraph, rather than the importance of the particular features of C, such as conditional control, arrays, pointers, functions and structures. Of course, even these topics have quite a few finer aspects that a beginner need not concern himself or herself with, initially.

The approach I have used in this book is to present the reader with only *the required details* at the *appropriate time*. This is essential, particularly for a beginner; otherwise the intricacy of the language may very well prove to be overwhelming and make progress difficult. Thus, I have placed emphasis on commonly used concepts in C to enable beginners to gain a thorough understanding of the essential features of the language. Concepts which are either complex or less frequently used, are presented in a separate section in each chapter under the heading '*Advanced Concepts*'. This, a beginner may easily skip on the first reading and revisit later after gaining basic proficiency over the language.

I have also taken care to see that the topics are presented logically, an understanding of which I have come to with my many years of teaching the subject. To facilitate learning, I have provided numerous short examples and complete programs throughout the book. In several cases, the solutions have been discussed in great detail with alternative ways of implementing them. There are many useful tips on

commonly made errors and how to avoid them, and guidelines for good programming practices which facilitate the learning process. I have also tried to include variety in the end-of-chapter exercises—working them out should give the student ample practice in developing good programming skills. The chapter on Turbo C/C++ graphics has been included to help the students who are interested in doing projects involving graphics.

It is my firm belief that the book will be of great help to beginners, and at the same time that it has plenty to offer intermediate and advanced learners of the language. It took me many years and several drafts to bring the book to this form—I am sure it can be improved further with feedback from readers. I welcome inputs in this regard and hope to use the same for improving future editions of the book.

I am grateful to my colleagues at SGGS Institute of Engineering & Technology, Nanded, particularly Dr P. Kar and R.P. Parvekar for their continuous inspiration. I am also grateful to Dr D. D. Shah, Principal, G. H. Raisoni College of Engineering & Management, Pune, for his valuable suggestions.

I thank Mr Madhu Reddy, Director, Universities Press for publishing this book, and the editors, Ms Madhavi Sethupathi and Ms Sreelatha Menon, for their editorial support and useful suggestions. Thanks are due to the referees for pointing out various ways of improving the manuscript.

And, finally, my most heartfelt thanks to my wife, Seema, and my children—Gandhar and Shalmali—for providing support, encouragement as well as valuable suggestions.

R. S. Bichkar  
June 2012

# 1 Introduction to Computers, Programming and the C Language

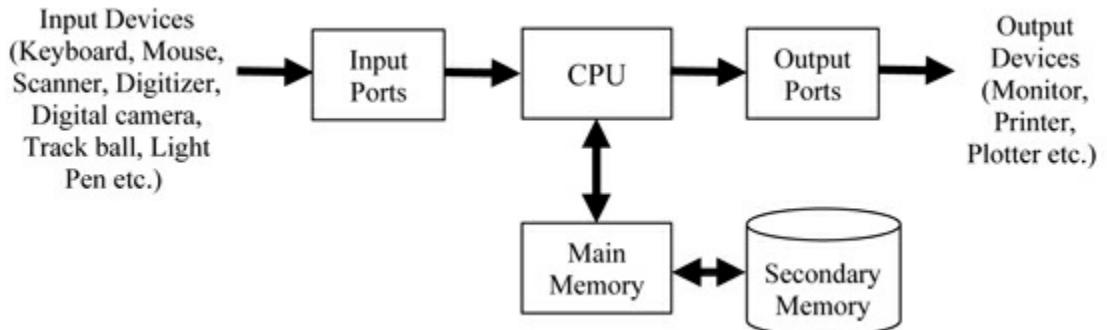
This chapter introduces computers, programming and the C language. The topics covered include the architecture of a digital computer, the program development process, flowcharts, introduction to C language and popular program development environments. The *Advanced Concepts* section provides a quick tour of C language, main memory organization and binary number system. A beginner may skip this section in first reading.

## 1.1 Computer Architecture

Computers (more specifically, *digital* computers) are based on the **stored program concept** given by Von Neumann. The block diagram of such a computer is shown in Fig. 1.1. It consists of a central processing unit (CPU), main memory, input and output ports, buses, secondary memory and the input and output devices.

CPU is the heart of a computer. The actions to be performed by the CPU are written as a program, which is a sequence of instructions. The CPU fetches and executes the program instructions one by one. It performs arithmetic and logical operations on data stored in the main memory and controls the activities in a computer. The CPU provides several internal registers that can be used to hold the operands of instructions as well as intermediate results. These registers are faster than the main memory.

The **main memory** is used to store programs that are being executed, the data and the results. This memory is also called **primary memory**. It is an electronic circuitry constructed using integrated circuits (ICs) and is very fast in operation. However, as the memory chips are expensive, a computer generally comes with limited main memory capacity.



**Fig. 1.1** The block diagram of a digital computer

There are mainly two types of memories: **random access memory** (RAM) and **read-only memory** (ROM). RAM is a read/write memory. To execute a program stored on the secondary storage (such as a hard disk), it is loaded in RAM before execution begins. RAM is a volatile memory and its contents are lost even if the power supply is momentarily interrupted. Hence, uninterrupted power supply (UPS) is often used to provide power to PCs. However, UPS is unnecessary for laptops as they have inbuilt battery.

As the main memory has limited capacity and is volatile and costly, the **secondary storage** is used for the long-term storage of programs and voluminous data. It is characterized by very large storage capacity, slower data transfer rate as compared to main memory and low costs. The storage capacities range from a few megabytes (MB) to several terabytes (TB). There are several secondary storage media such as magnetic disks and tapes, optical disks (CD, DVD), etc. As magnetic disks provide very good data transfer speeds at an affordable price, they are most commonly used. Optical disks are much cheaper than magnetic disks and are used for offline data storage. To read/write information from/to these secondary memories, various devices such as floppy disk drive, hard disk drive, CD/DVD drive, magnetic tape drive, etc., are provided in a computer.

The human-machine interface is achieved using the various devices interfaced through the input and output ports. The **input ports** are used to read the data from input devices, whereas an **output ports** are used to write/display the data on output devices. The common input devices are keyboard and mouse. The common output devices are monitor and printer. Besides these devices, we can interface several other devices like scanner, plotter, joystick, light pen, camera, modem, etc. As the devices are in the periphery of the CPU, they are often called as peripheral devices or simply **peripherals**.

A group of signal lines is called a **bus**. The various blocks/circuits of a computer are connected using buses. There are three types of buses: data bus, address bus and control bus. These names indicate the type of electrical signals they carry.

The constituents of a computer system can be broadly classified into two parts: hardware and software. The **hardware** comprises physical circuits, devices, equipment, etc. Thus, CPU, main memory, I/O ports and various input/output devices (such as keyboard, mouse, monitor, printer, etc.) are hardware components.

The **software** refers to the programs used to perform the desired activities. These include operating

systems (MS Windows, Linux, etc.), office suites (MS Office, LibreOffice, etc.), program development environments (Turbo C/C++, Dev C++, Code::Blocks, MS Visual Studio, NetBeans, Eclipse, etc.), browsers (Internet Explorer, Mozilla Firefox, etc.), media players, anti-virus software, utilities, games, design and simulation software and a lot more.

## 1.2 Program Development Process

The process of developing a computer program for the solution of a problem involves several phases or steps. These include problem definition, design, coding, compilation, testing and debugging, documentation and maintenance.

The **problem definition** step defines the problem and its scope. It also identifies the input data and the desired output.

In **design** step, the problem is analyzed to determine the data structures and operations required for its solution. The problem is often subdivided into smaller sub-problems to avoid repetitive coding and to simplify the design. The solution is represented using flowcharts and/or algorithms. The design step also involves the formulation of the test data set.

The solution provided in the design step is usually independent of any programming language. In the **coding** step, a suitable programming language, such as C, C++, Java, etc., is used to write the program. This program is entered into the computer using a program called **editor** and saved in a program file. We can either use a stand-alone editor such as *vi*, *emacs*, *notepad*, etc. or one provided as an integral part of the IDE (integrated development environment), as in Turbo C/C++, Dev-C++, Code::Blocks, etc.

Once a program is entered, it is compiled using a program called **compiler**. The compiler verifies whether the language rules have been followed strictly or not, and points out **syntax errors**, if any, contained in the program. These errors are corrected using the editor and the program is compiled again. This process is repeated till we obtain a program free of syntax errors. The compiler converts such a program into an **executable file**.

The executable file generated by the compiler is then executed. If the program requires any input data, it is provided and the program output is checked. The program is tested for each entry in the test data set. If the program gives incorrect output for any of the test data, it is said to contain a **bug**. The **debugging** step is used to identify and correct these bugs. This step is also repetitive and involves the use of editing, compilation and testing phases.

Each program must be properly documented to ease the process of program maintenance. **Documentation** is an integral part of the overall program development cycle. Comments are included in a program for the purpose of documentation. The flowchart and algorithm also form important parts of the program documentation.

The program is now ready for use and is distributed to users. As the program continues to be used, more subtle bugs are likely to be found. These bugs must be fixed. In addition, programs may need maintenance due to efficiency considerations that may arise while users attempt to solve larger

problems. Further, the program might be expected to do things beyond its original objectives. This may be because users demand new features or because of changes in the user requirements. Thus program maintenance may range from minor corrections to major rewriting. Program maintenance is one of the most important phases in the life of a program, and huge costs may be incurred if programs are not properly designed, developed and documented.

## 1.3 Structured and Modular Programming

In the early days, computer programs were written in a haphazard fashion without much consideration to program clarity or program maintenance likely to come up in the future. Shortcuts were taken where possible, patches were used to fix the problems and programs jumped from one place to another in what is now called **spaghetti code**. Such programs became so involved that they were often unreadable. The program maintenance costs were very high and at times the programs were simply discarded and rewritten afresh.

The **structured programming** philosophy was initiated by Edsger Dijkstra and others to solve these programming problems. In structured programming, programs are written using only three control structures: **sequencing**, **selection** and **repetition**. Each structure is entered at the top and is exited at the bottom. A branch instruction such as *goto* is unnecessary and disallowed. Thus, the problem of spaghetti code is eliminated. Such programs are easier to read, understand and maintain.

**Modular programming** is a technique which allows a program to be decomposed into a number of individual modules or subprograms, which can be developed easily and independently. Modular programming has several advantages:

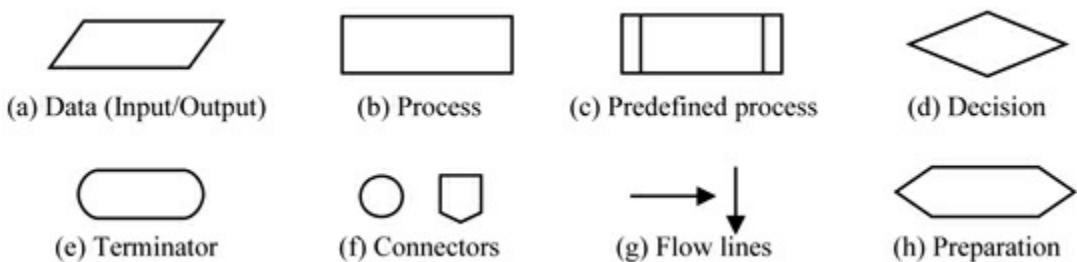
1. Programs have better logical clarity.
2. Programs are easy to develop, debug and maintain.
3. A module can be accessed several times, and each time a different information can be processed. This avoids repetitive coding and reduces program length considerably.
4. It permits *information hiding*, an important concept of structured programming. This means that the rest of the program need not know how the information is processed by a particular module. It just needs to know what information is processed by a particular module (or what action is performed).

## 1.4 Flowcharts

A **flowchart** is a diagram showing a sequence of activities to be performed for the solution of a problem. It provides a concise (symbolic) description of the solution in place of a lengthy narrative. A set of conventional symbols is used to draw a flowchart. The activities in a flowchart are connected by flow lines indicating the sequence of operations.

### 1.4.1 Flowchart Symbols

A flowchart is drawn using a set of conventional symbols. Fig. 1.2 shows the commonly used flowchart symbols. These symbols are connected using flow lines. Typical connections of the flowchart symbols and flow lines are shown in Fig. 1.3. The flowchart symbols are described below.



**Fig. 1.2 Commonly used flowchart symbols**

**Data symbol:** The data symbol, shown in Fig. 1.2a, is also called the input/output symbol. It is used to indicate data input or output operations. Commands like INPUT, READ, WRITE, PRINT, etc. are written in it along with the parameters. For example, to read two numbers from the keyboard, we can write INPUT *a, b*.

Two flow lines are associated with this symbol as shown in Fig. 1.3a. One enters it and the other leaves it. The data input or output is with respect to the computer. If we give the data to the computer, it is considered as input, whereas the results printed by computer on the monitor, printer or a disk file is output data.

**Process symbol:** The process symbol, shown in Fig. 1.2b, is generally used for data manipulation, assignment of values to variables, etc. The operations to be performed are written in the symbol. The backward arrow ( $\leftarrow$ ) is usually used to assign the value of expression to variable, as in *variable*  $\leftarrow$  *expression*. Two flow lines are associated with this symbol as shown in Fig. 1.3b. One enters the symbol and other leaves it.

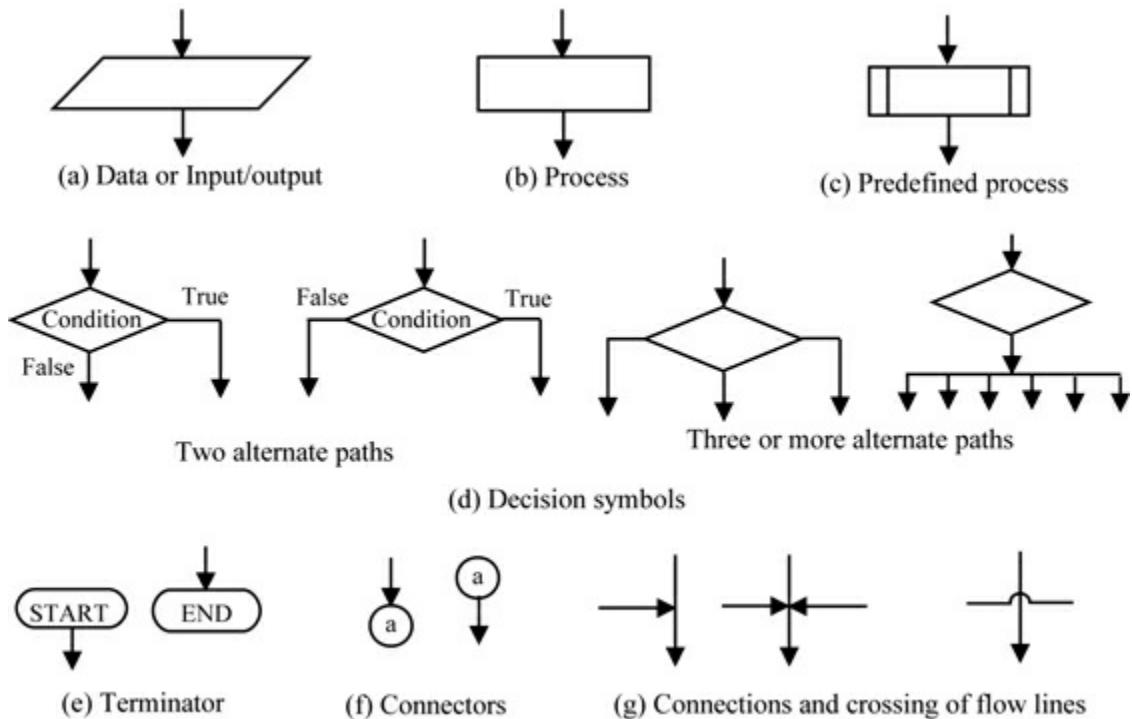
**Predefined process symbol:** The predefined process symbol is shown in Fig. 1.2c. It is also called subprogram symbol. It is used for a predefined subprogram <sup>t</sup> (a function or a subroutine). Two flow lines are associated with it, as shown in Fig. 1.3c. The incoming flow line indicates a jump to the subprogram and the outgoing line indicates the return to the calling program. The flowchart for a subprogram is drawn separately.

**Decision symbol:** The symbol used is a diamond, as shown in Fig. 1.2d. It is used where a decision is to be taken to select one of the two or more alternative paths. A condition (or a question) is written in this symbol. One flow line enters the symbol, and depending on the type of condition, there may be two or more flow lines coming out of this symbol. These flow lines go to different parts of the flowchart, constituting alternative paths.

**Terminator symbol:** The terminator (or terminal) symbol shown in Fig. 1.2e is used to denote the beginning and end of the flowchart. Words like START and END are written in it. Only one flow line is associated with it, as shown in Fig. 1.3e.

**Connector symbol:** Connector symbols shown in Fig. 1.2f are used to show the connection between two or more parts of flowcharts drawn at different places or on different pages. These symbols are useful when drawing lengthy flowcharts. A letter or a number is written inside the symbol. A small circle is used to show the connection if two parts are on the same page. The other connector symbol is used as an off-page connector. The connection between two parts of a flowchart is established by writing the same letter or number in two connector symbols. Only one flow line is usually associated with these symbols, as shown in Fig. 1.3f.

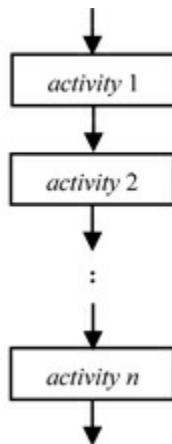
**Flow lines:** The flow lines are used to connect flowchart symbols, indicating the sequence of activities to be performed. An arrow on a flow line indicates the direction of flow. The general direction of flow is from top to bottom and left to right. The conventions used for connection and crossing of flow lines are shown in Fig. 1.3g.



**Fig. 1.3** Commonly used flowchart symbols with flow lines

#### 1.4.2 Flowcharts for Sequencing

In sequencing construct, the activities in a flowchart are written in a linear sequence as shown in Fig. 1.4. The activities are performed one after another in the sequence indicated by flow lines.



**Fig. 1.4** Flowchart for sequencing construct

**Example 1.1** Flowchart to calculate the volume and surface area of a sphere

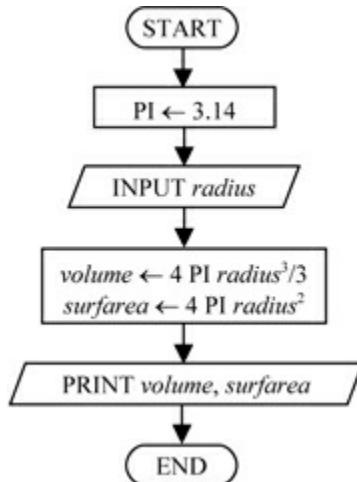
The volume and surface area of a sphere are given as  $v = 4\pi r^3/3$  and  $a = 4\pi r^2$ , respectively, where  $r$  is the radius of the sphere and  $\pi = 3.14$  is a mathematical constant. Let us use the variables *radius*, *volume* and *surfarea* to represent radius, volume and surface area of the sphere. Note that *radius* is an input variable, whereas *volume* and *surfarea* are output variables. Since  $\pi$  is not a valid name in most programming languages including C, we use PI for it. Note the use of uppercase letters to represent the name of a constant.

The flowchart is given in Fig. 1.5. We first assign the value 3.14 to PI and accept the value of *radius* from the keyboard. Then we compute *volume* and *surfarea* and print them.

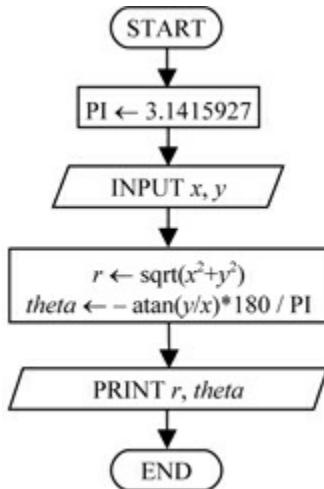
**Example 1.2** Flowchart to convert Cartesian coordinates to the polar form

For a point having Cartesian coordinates  $(x, y)$ , the polar coordinates are given as  $(r, \theta)$  where

$$r = \sqrt{x^2 + y^2} \text{ and } \theta = \tan^{-1} \frac{y}{x}.$$



**Fig. 1.5** Flowchart to calculate volume and surface area of a sphere



**Fig. 1.6** Flowchart to convert Cartesian coordinates to polar form

In these equations,  $x$  and  $y$  are input quantities and  $r$  and  $\theta$  are output quantities. To calculate square root and arc-tangent (i. e.,  $\tan^{-1}$ ), the C standard library provides the *sqrt* and *atan* functions, respectively. The *atan* function returns the angle in radians. Let us assume that we will display angle  $\theta$  in degrees. Thus, the modified equation to calculate the value of  $\theta$  in degrees is given as  $\theta = -\tan^{-1}(y/x) \times (180/\pi)$ .

Let us use the variables  $x$ ,  $y$ ,  $r$  and *theta* to represent the coordinate values and constant *PI* to represent the constant  $\pi$ . The flowchart given in Fig. 1.6 first assigns value 3.1415927 to *PI* (note that a more accurate value is used here) and then accepts Cartesian coordinates of a point (i. e.,  $x$  and  $y$ ) from the keyboard. The corresponding polar coordinate values are then calculated and printed.

### 1.4.3 Flowcharts for Selection

The selection constructs are used to select one of several alternative activities for execution. They make use of a decision symbol. A condition (or a question) is written within the decision symbol. Depending on the outcome of this condition, the appropriate activity is selected for execution.

Two constructs are available for selection: *if* and *switch* construct. The *if* construct tests the value of a logical condition such as  $a \geq b$ ,  $\text{marks} < 35$ , etc. and depending on the outcome, selects one of the two alternative activities for execution. The **switch construct** tests the value of an expression and depending on its value, selects one of several alternative paths. These constructs are described below.

#### The **if** Construct

The *if* construct uses a decision symbol with a logical condition. Since a logical condition can evaluate to either true or false, only two flow lines leave the decision symbol. We have two types of *if* constructs: *simple if* (also called the *if-then* construct) and *if-else* (also called the *if-then-else* construct).

The **simple if** construct is shown in Fig. 1.7a. In this construct, only one activity is specified. This *activity* is executed only if the *condition* evaluates as true; otherwise, nothing happens. On the other hand, the **if-else construct** shown in Fig. 1.7b specifies two activities, only one of which is executed depending on the outcome of the condition. If the *condition* evaluates as true, then *activity 1* is executed; otherwise, *activity 2* is executed.

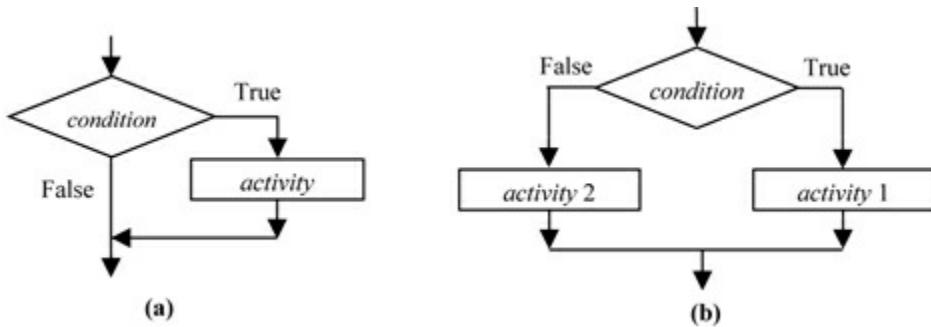


Fig. 1.7 *if* constructs: (a) simple *if* construct, (b) *if-else* construct

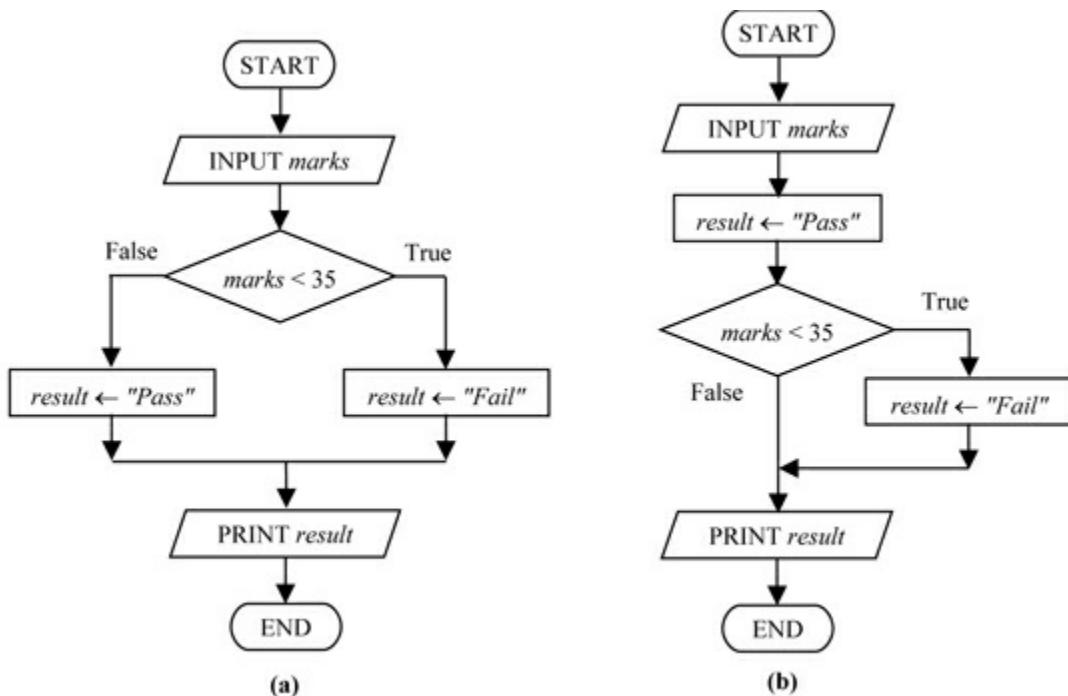
#### Example 1.3 Examination result in a single subject

Let us draw a flowchart to accept marks obtained by a student in a subject and determine the result, i. e., Pass/Fail. Assume that the maximum marks in this subject are 100 and a student fails if he/she scores less than 35 marks.

Let us use variable *marks* to store the marks in a subject and variable *result* to store the result *string*<sup>†</sup>. In this example, variable *result* should be assigned either string "Pass" or "Fail" depending on the value of *marks*. Thus, *if-else* construct is suitable. The flowchart given in Fig. 1.8a first reads the value of *marks* and then tests it in an *if-else* construct. If the condition  $\text{marks} < 35$  is satisfied, variable *result* is

assigned string "Fail"; otherwise, it is assigned string "Pass". Finally, the value of the variable *result* is printed.

Note that we can also use *simple if* construct in place of *if-else* to achieve the same result. The resulting flowchart is given in Fig. 1.8b. After reading the value of *marks*, the variable *result* is initialized with the string "Pass". Then the condition *marks < 35* is tested and if this condition is satisfied, the *result* is changed to "Fail". Finally, *result* is printed.



**Fig. 1.8** Flowcharts to determine and print the examination result in a single subject

### The Nested **if** Construct

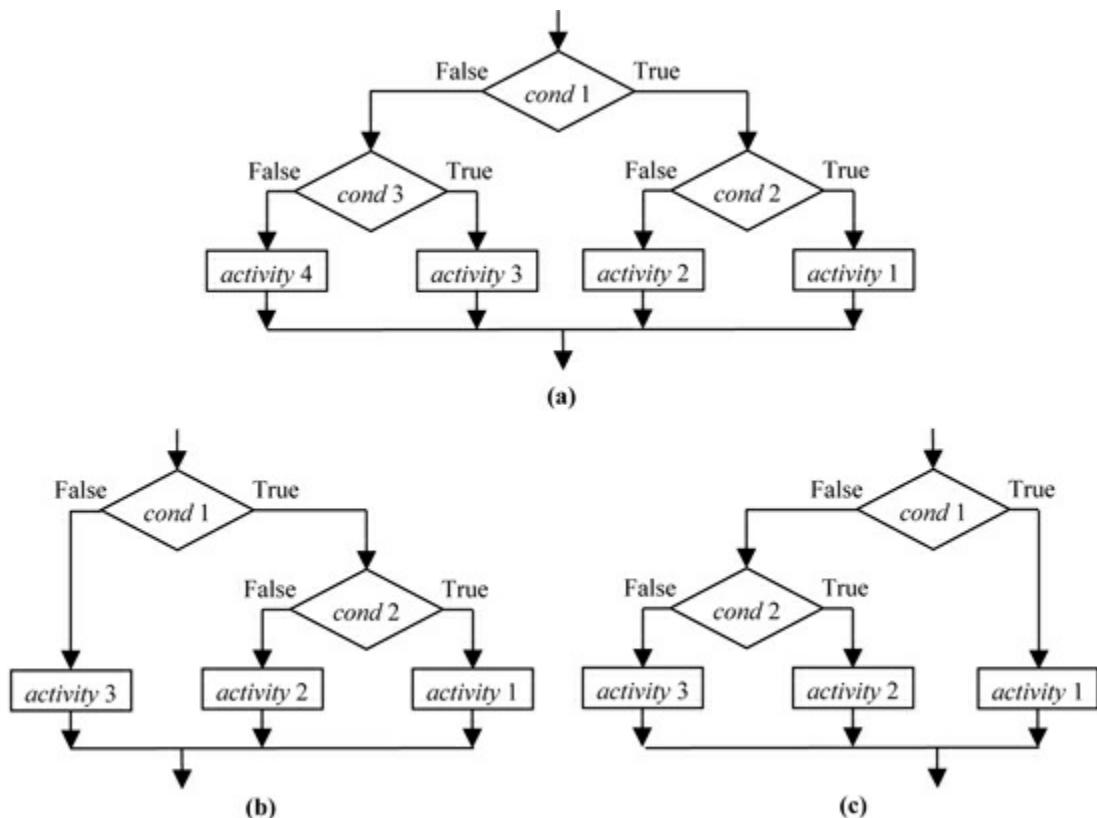
The activity blocks within an *if* construct may be replaced by *if* constructs themselves. Such constructs are called *nested if* constructs. Fig. 1.9 shows the flowcharts for various **two-level nested if** control structures obtained by replacing one or more activities in the *if-else* construct with *if-else* construct itself. In Fig. 1.9a, both the activities are replaced, whereas in other two figures, only one of the activities is replaced. In Fig. 1.9b, the activity in the *True* branch is replaced, whereas in Fig. 1.9c the activity in the *False* branch is replaced.

We can obtain more forms of two-level *nested if* flowcharts by replacing the activity with *simple if* construct rather than the *if-else* construct. We can also replace the activity in the *simple if* construct with either *simple if* or an *if-else* construct.

If we further replace the activities in the two-level *nested if* constructs with the *simple if* or *if-else* construct, we obtain a three-level *nested if* construct.

### Example 1.4 Examination result in a single subject with data validity check

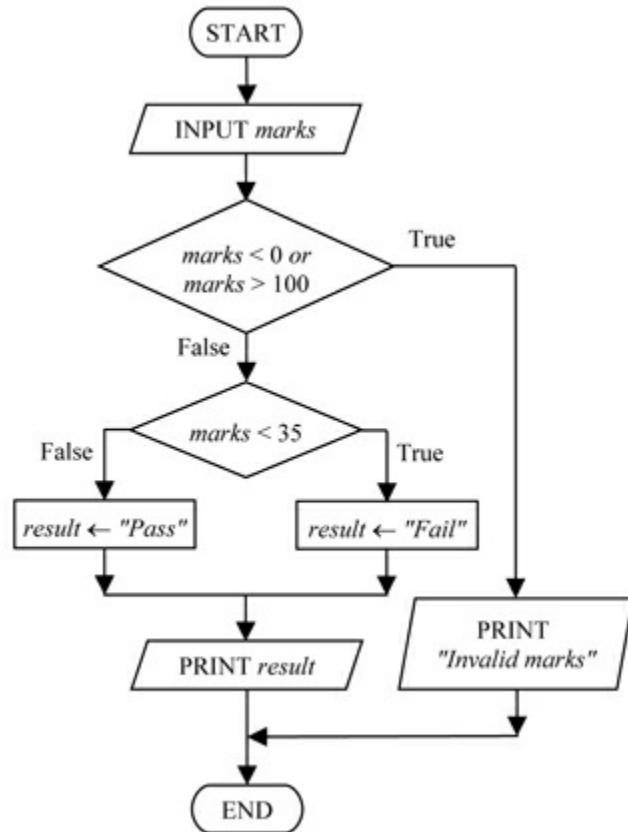
Flowcharts to accept the marks of a student in a single subject and determine the result were given in Fig. 1.8. Now consider that the user may enter invalid values of *marks*, e. g., for 140, the flowchart determines result as "Pass" and for value of marks as -50, the result is "Fail". To avoid such situations of processing invalid user data, it is generally a good idea to test the validity of data before performing any operations on it. If the data is invalid, usually an error message is printed and the program ends. Otherwise, the entered data is valid and it is processed.



**Fig. 1.9** Two-level nested-if control structures

Fig. 1.10 gives a modified flowchart. It uses a *nested if* construct to test whether the value of *marks* entered is valid or not and determines the result only if it is valid. In this flowchart, an *if-else* construct to determine the result is included in the *False* branch of outer *if-else* construct that tests the validity of *marks*. If the value of *marks* is invalid (less than 0 or greater than 100), an error message "Invalid marks" is printed and program ends; otherwise, program execution continues with the determination of the result. Note that we can alternatively redraw this flowchart by including the *if-else* construct to determine the result in the *True* branch of the *if-else* construct that tests the validity of the marks. This is left as an exercise.

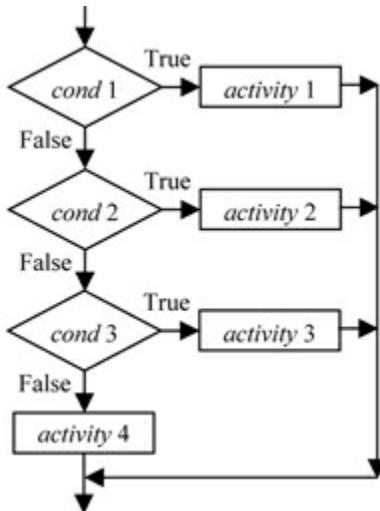
Most users will usually expect that the program should discard invalid data and allow the user to enter the data again. It is likely that a user will again make some error during data entry. Thus, the data should be accepted repeatedly until the user enters valid data. Such functionality can be added using loops, which are covered shortly.



**Fig. 1.10** Flowchart to determine examination result in a single subject with marks validity check

### The **if-else-if** Construct

The **if-else-if construct** is a special case of the *nested if* control structure in which only the activities in the *False* branches are replaced by *if-else* constructs as shown in Fig. 1.11. Observe that the *True* branch in each condition block contains a simple activity (not an *if* construct) and the *False* branch contains an *if-else-if* construct.



**Fig. 1.11 Flowcharts for if-else-if construct**

If condition *cond 1* is true, *activity 1* is performed; otherwise, condition *cond 2* is evaluated. If this condition evaluates as true, *activity 2* is performed; otherwise, condition *cond 3* is evaluated. Finally, if this condition is true, *activity 3* is performed. Otherwise, *activity 4* is performed.

#### Example 1.5 Flowchart to calculate the salary of an employee

The salary of an employee of ABC Inc. consists of basic salary, dearness allowance and house rent allowance. The dearness allowance is calculated as some percentage of basic salary (e. g., 65%) and the house rent allowance depends on the basic salary as shown below.

Basic salary (bs)	House rent allowance
$bs \geq 16400$	3000
$12000 \leq bs < 16400$	2000
$8000 \leq bs < 12000$	1500
$bs < 8000$	1200

A flowchart to calculate the salary of an employee is given in Fig. 1.12. It first accepts the basic salary (*bs*) and dearness allowance rate (*da\_rate*). It then uses an *if-else-if* construct to determine the house rent allowance (*hra*). Observe that the tests for basic salary in the *if-else-if* construct are performed in the order of the lower value of basic salary to higher values. Finally, it calculates dearness allowance (*da*) and total salary (*tot\_sal*) and prints the result.

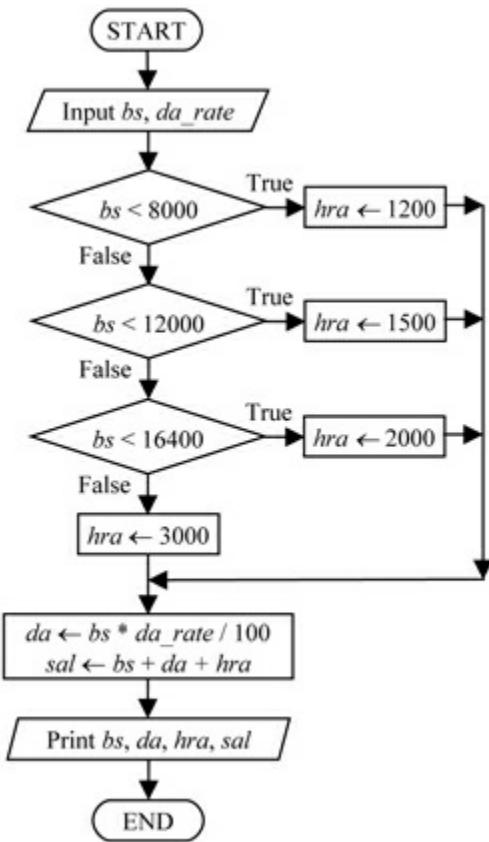
#### Example 1.6 Flowchart to determine interest on fixed deposit (varying interest rates)

Banks usually give interest on fixed deposits, with an interest rate that increases with the duration of the deposit. Consider a bank that gives simple interest on fixed deposits using the interest rate as shown below.

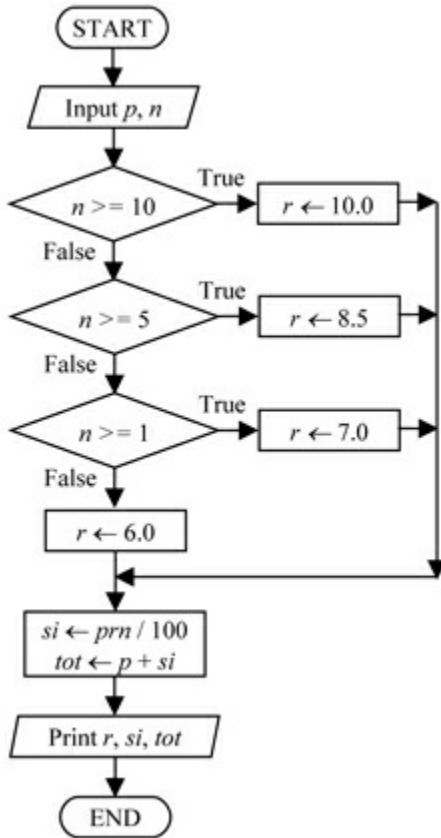
Duration $n$ (years)	Interest rate
$n \geq 10$ years	10.0%
$5 \leq n < 10$	8.5%
$1 \leq n < 5$	7.0%
$n < 1$	6.0%

Fig. 1.13 shows a flowchart to determine the interest and the total amount for an amount deposited with this bank. The flowchart first accepts the values of principal ( $p$ ), i. e., the amount deposited and the duration ( $n$ ) in years. It then uses an *if-else-if* construct to determine the interest rate ( $r$ ) and then calculates and prints the simple interest ( $si$ ) and the total amount ( $tot$ ).

Observe how the conditions are written in the *if-else-if* construct. The condition block tests if the value of  $n$  is greater than or equal to 10. If this condition is true, the interest rate  $r$  is assigned a value of 10.0; otherwise, the second condition tests if the value of  $n$  is greater than or equal to 5. Note that since this condition is tested only if the first condition ( $n > 10$ ) evaluates as false, the test for  $n$  less than 10 is not required. Similarly, the test in the next condition is written concisely as  $n \geq 1$  instead of  $n \geq 1$  and  $n < 5$ . Finally, note that the last condition in the table given above is not required as the duration will be obviously less than 1 year if the above three conditions evaluate as false.



**Fig. 1.12** Flowchart to determine salary of an employee



**Fig. 1.13 Flowcharts to calculate simple interest on fixed deposit with variable interest rate**

### The **switch** Construct

The **switch** construct is used to select one of several alternative activities for execution. Fig. 1.14 shows a *switch* construct. A decision symbol tests the value of an expression, which is usually an integer expression. Several flow lines leave the decision symbol, each leading to one of the alternative activities (also called *cases*) to be performed. Each flow line is labelled with a list containing one or more values of expression written in the decision symbol. Also, one of the flow lines, usually the last (i. e., rightmost), is labelled with words such as *Default* or *Otherwise*. However, such a flow line is optional. Note that each value in the lists *List 1*, *List 2*, ... *List N* must be unique.

When control enters a *switch* construct, the expression written in the decision symbol is evaluated. If the value of this expression is included in *List 1*, then *Activity 1* is executed; otherwise, if the value is included in *List 2*, then *Activity 2* is executed and so on. If the value is not included in any of the lists from *List 1* to *List N*, the control follows the flow line labelled *Default*, if it exists, to execute the default activity. After execution of any one of the alternative activities, the next activity in the flowchart (not shown in Fig. 1.14) is executed. Note that if the value of the expression is not contained in any label list and if the default activity is also not specified, none of the alternative activities will be executed.

### Example 1.7 Flowchart to evaluate expression of the form $a op b$ (simple calculator)

Fig. 1.15 shows a flowchart to evaluate an expression  $a op b$  entered from the keyboard, where  $a$  and  $b$  are numbers and  $op$  is an operator that may be one of the following:  $+$ ,  $-$ ,  $*$  and  $/$ . The *switch* construct is suitable in this case as we have to perform a multi-way decision comprising addition, subtraction, multiplication and division depending on the value of operator  $op$ .

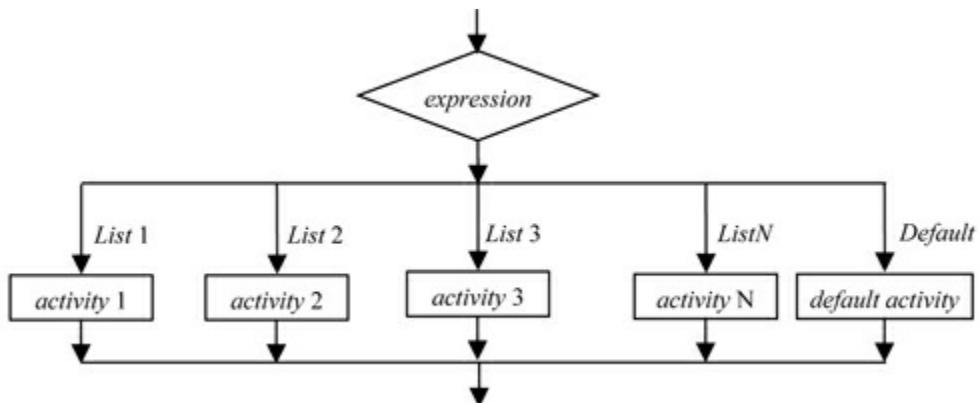


Fig. 1.14 The switch construct

The flowchart first accepts values of variables  $a$ ,  $op$  and  $b$  and then uses a *switch* construct to print the result depending on the value of  $op$ . Note that the *Default* alternative of the *switch* construct prints an error message "Invalid operator".

#### Nesting of **case** and **if** Constructs

As in the case of *if* constructs, the *switch* construct can also be nested wherein one or more activities in a *switch* construct can be replaced by other *switch* constructs.

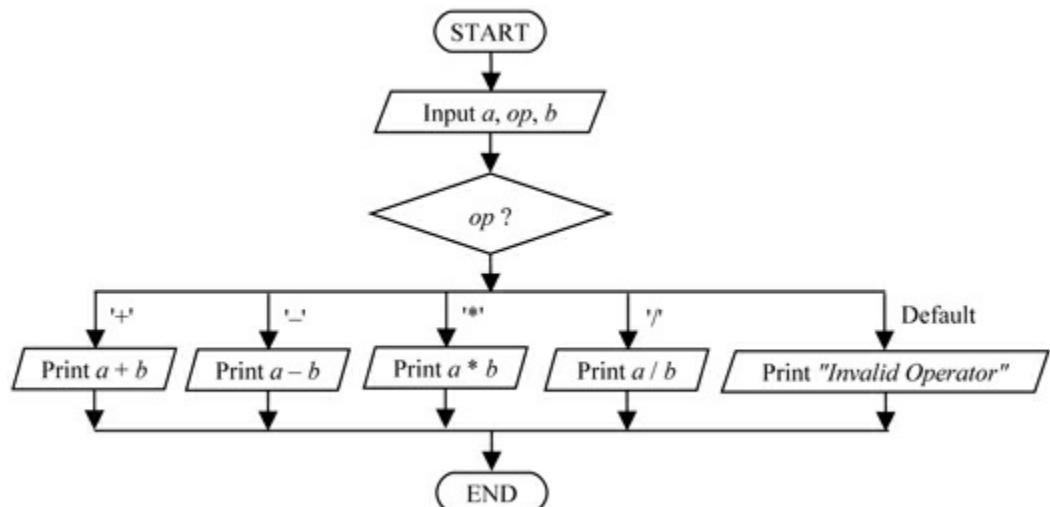
In addition, the *if* and *switch* constructs can be nested within each other. Thus, one or more activities in a *switch* construct can be replaced by *if* constructs and one or more activities in an *if* construct can be replaced by *switch* constructs.

#### 1.4.4 Flowcharts for Loops

When a particular activity is to be repeatedly executed, a **loop** or **repetition** construct is used. In this construct, after the activity is executed once, the control is again transferred to the beginning of that activity. One execution of the activities in a loop is called an **iteration**. There are two types of loops: **controlled** and **endless** or **infinite loop**.

The execution of the controlled loop stops when a specified condition is satisfied or a specified number of iterations are performed. Depending on the number of iterations performed, controlled loops can be classified into two broad categories:

- Loops with unknown number of iterations, e. g., we have to add the numbers from a given list till a zero number is encountered. The number of iterations required in this case can't be determined in advance as it will vary with different sets of data.



**Fig. 1.15 Flowchart for a simple calculator**

- Loops with known iterations. For example, to process the marks of 50 students in a class, we require 50 iterations.

We have three constructs for the implementation of controlled loops: *while*, *do ... while* and *for*. The *while* and *do ... while* constructs are more suited for the implementation of loops with unknown number of iterations. Although we can use them to implement the loops with known number of iterations as well, the *for* construct is more suitable and is usually preferred over the others. Note that some programming languages, such as Pascal, provide a *repeat until* construct which is a variant of the *do ... while* construct.

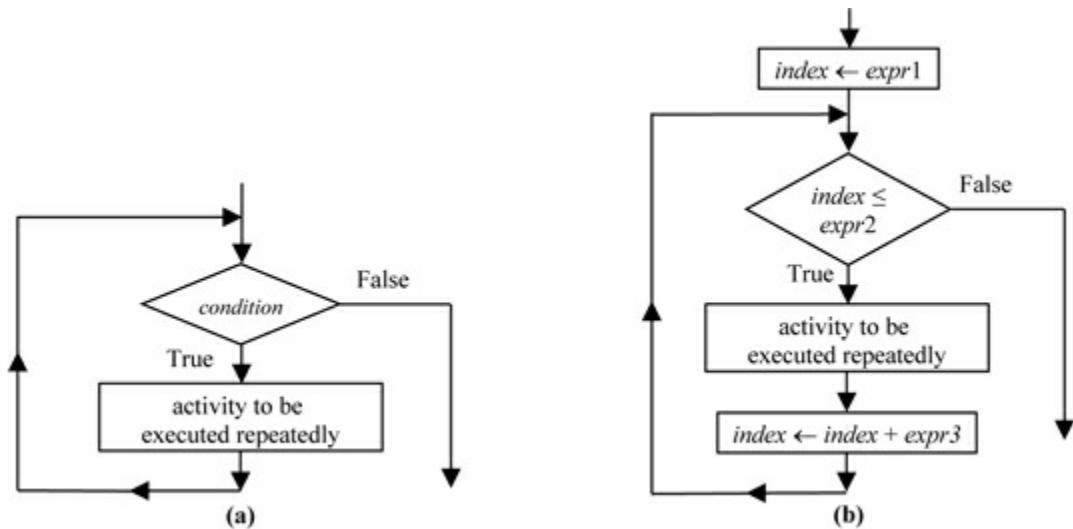
An endless loop performs the activities included in the loop continuously and there is no provision to exit the loop. To stop the execution of such a loop, we can either interrupt the program from the operating system (e. g., close the window in which the program is running) or shut down the computer.

## The **while** Loop

The **while loop** is suitable for situations where the number of iterations are not known in advance. Fig. 1.16a shows the *while* loop construct. In this loop, a condition is tested at the beginning of the loop and activities in the loop are executed *while* (i. e., as long as) the condition evaluates as true. The control leaves the loop when the condition evaluates as false. Observe that as the loop condition is tested at the beginning, the activities in the loop may not be executed at all. The *while* loop construct is also called **entry-controlled** or **top-tested loop**.

Although the *while* loop is more suited for unknown number of iterations, we can also use it to

implement loops with known number of iterations, as illustrated in Fig. 1.16b. For this, we use an index or **loop variable** to keep track of the number of iterations performed.



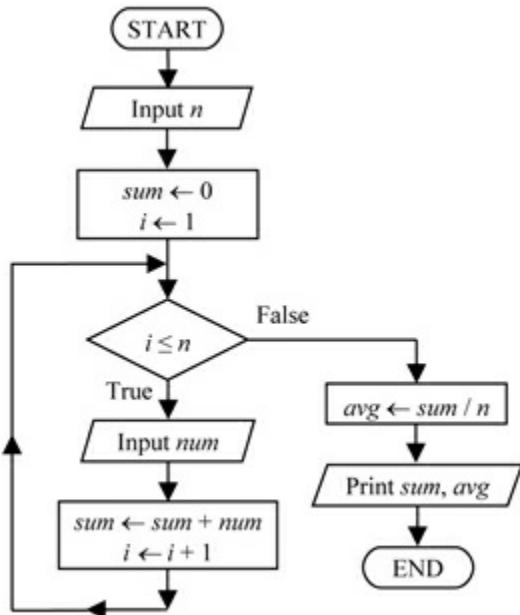
**Fig. 1.16** The while loop: (a) unknown number of iterations, (b) known number of iterations

The loop variable is assigned an initial value (*expr1*) before the loop and its value is incremented by stepping up the value (*expr3*) in the loop, usually after the desired activity is executed. The activities in the loop are executed repeatedly as long as the value of the index variable is less than or equal to desired final value (*expr2*). For example, to perform 10 iterations, we can set the initial, final and step values of the index variable as 1, 10 and 1, respectively.

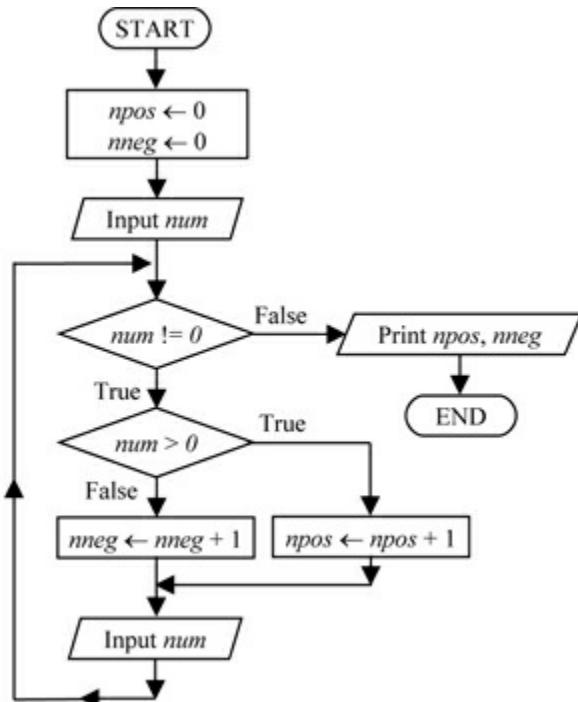
#### Example 1.8 Determine the sum and average of $n$ numbers entered from the keyboard

A flowchart to determine the sum and average of  $n$  numbers entered from the keyboard is given in Fig. 1.17. In this example, we first accept the value of  $n$  and then set up a *while* loop to determine the sum of  $n$  given numbers.

The variable *sum*, which is used to hold the sum of given numbers, is initialized to zero before the loop. Observe that variable *i* is used as the loop variable with the initial, final and increment values of 1,  $n$  and 1, respectively. Thus, the loop variable assumes consecutive values from 1 to  $n$ . In each iteration of the loop, one number is accepted from the keyboard and is added to variable *sum*. The control leaves the loop when the condition  $i \leq n$  evaluates as false. Now, the variable *sum* contains the sum of  $n$  given numbers. We then determine the average of the given numbers in variable *avg*, and print the values of *sum* and *avg*.



**Fig. 1.17** Flowchart to determine sum and average of  $n$  numbers entered from keyboard



**Fig. 1.18** Flowchart to count positive and negative numbers in a given list terminated with zero

**Example 1.9** Count positive and negative numbers in a given list until zero is entered

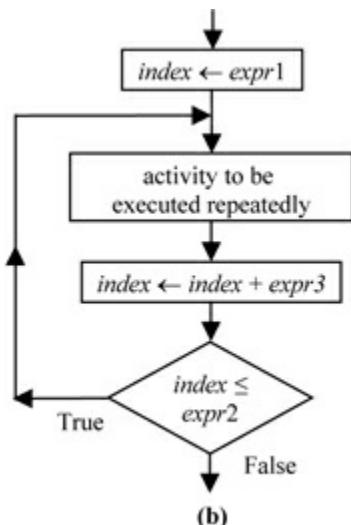
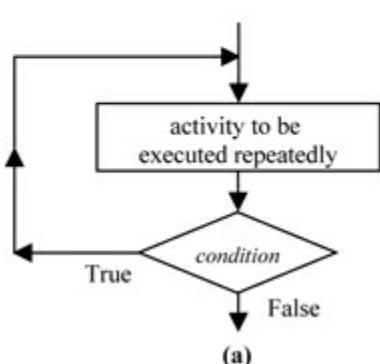
Fig. 1.18 shows a flowchart to count positive and negative numbers in a list entered from the keyboard until zero is entered. Initially, the variables `npos` and `nneg`, which are used to store the counts of positive and negative numbers in the given list, are initialized to zero and the first number in the given list is accepted in variable `num`. Then a *while* loop is set up, the activities in which are executed as long as the value of number `num` is non-zero. In each iteration of this loop, we increment either `npos` or `nneg` depending on the current value of variable `num` and then accept a new value in variable `num`. The values of `npos` and `nneg` are printed when control leaves the loop.

### The **do ... while** Loop

A flowchart for the ***do ... while* loop** is shown in Fig. 1.19a. The activities within the loop are performed repeatedly as long the specified condition evaluates as true, and control leaves the loop when the condition evaluates as false. Since the condition is tested at the end of the loop, such a loop is called **bottom-tested or exit-controlled loop**.

Both the *while* and *do ... while* loops are better suited to situations where the number of iterations is not known in advance. However, the basic difference between a *while* and a *do ... while* loop is that the activities within a *while* loop may not be executed at all as it is an entry-controlled loop, whereas the activities within a *do ... while* loop will always be executed at least once as it is an exit-controlled loop.

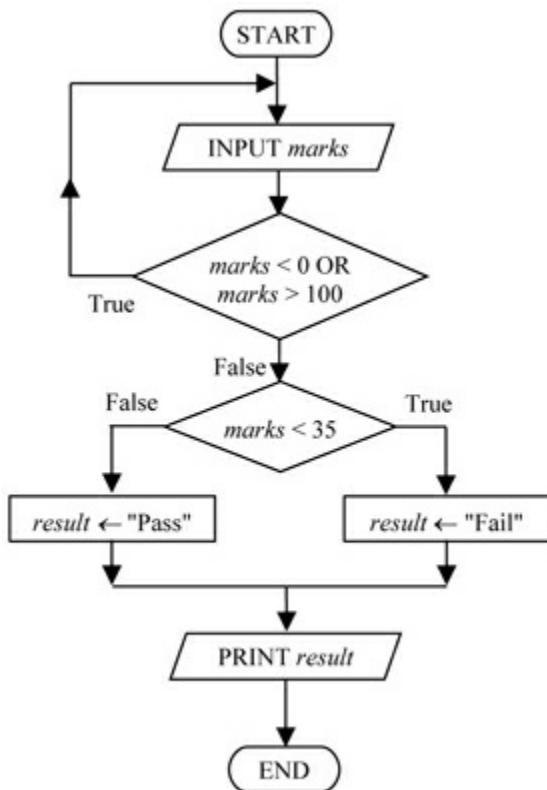
As in the case of a *while* loop, we can also use a *do ... while* loop to implement loops having known number of iterations, as shown in Fig. 1.19b. For this, we use an index variable to keep track of the number of iterations performed. The value of the index variable is initialized to *expr1* before the loop and is incremented by the stepping value (*expr3*) in the loop, usually after the desired activities are performed. The activities in the loop are executed repeatedly as long as the value of the loop variable is less than or equal to desired final value (*expr2*). For example, to perform 10 iterations, we can set the initial, final and step values of the index variable as 1, 10 and 1, respectively.



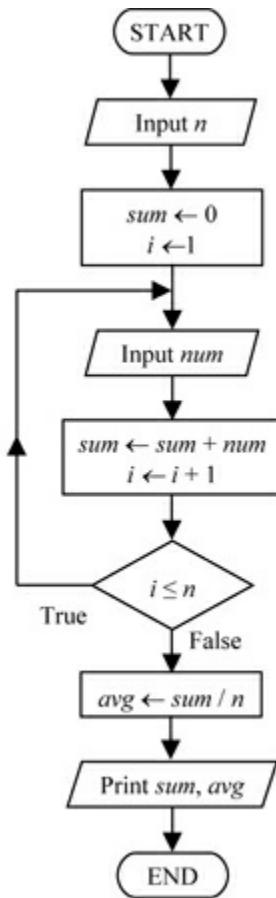
**Fig. 1.19** The do ... while loop: (a) unknown number of iterations, (b) known number of iterations

**Example 1.10** Accept valid marks and print examination result in a single subject

A flowchart to print the examination result in a single subject with marks validity test was given in Fig. 1.10, in which an error message is displayed if the user enters invalid marks. Now, let us modify that flowchart to accept valid marks (i. e., allow the user to re-enter the marks as long as he/she enters an invalid value) and then display the result. Clearly, we require a loop to accept the value of **marks**. As we have to accept at least one value of **marks**, the *do ... while* loop is better suited here compared to the *while* loop. The modified flowchart is shown in Fig. 1.20.



**Fig. 1.20** Flowcharts to accept valid marks and determine examination result in a single subject



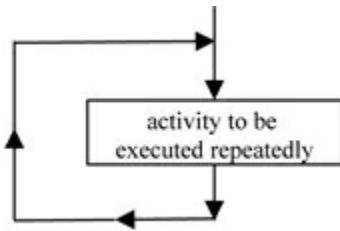
**Fig. 1.21** Flowchart to determine sum and average of  $n$  numbers entered from the keyboard

**Example 1.11** Determine the sum and average of  $n$  numbers entered from the keyboard

A flowchart to determine the sum and average of  $n$  numbers entered from the keyboard using the *while* loop was given in Fig. 1.17. A modified flowchart of the same that uses the *do ... while* loop is given in Fig. 1.21.

### Endless Loop

Fig. 1.22 shows the flowchart for an **endless loop**. After the execution of the activities within the loop, the control is transferred unconditionally to the beginning of those activities. Although the activity to be repeated has been shown as a single block, it will usually consist of more complex operations involving sequence, selection and loops.



**Fig. 1.22 An endless loop**

## 1.5 The C Programming Language

C is one of the most popular programming languages of all time. It was developed in 1972 by **Dennis Ritchie** at Bell Telephone Laboratories for use with the Unix operating system.

C is a general purpose programming language. It has several useful characteristics that include efficiency, portability, ability to access machine hardware and low demand on system resources. Hence, it is often used for the development of system software such as operating systems, compilers, interpreters, libraries and embedded system applications. C allows efficient implementation of algorithms and data structures, making it useful for computation-intensive applications. C has also been widely used to implement application software.

### 1.5.1 C Language Standards

The C language was described by Brian Kernighan and Dennis Ritchie in the first edition of their book, *The C Programming Language*, which is popularly known as **K&R**. For many years, this book served as an informal specification of the C language, which is often referred to as **K&R C**.

Initially, the C language was primarily used in academic environments. In the early 80s, it was implemented on a large number of computers. As the popularity of C began to increase significantly, the developers of C compilers provided several non-compatible extensions to the language. Hence, the American National Standards Institute (ANSI) standardized the C language in 1989 by incorporating many of these extensions and by introducing several new features to obtain a superset of K&R C. This version of the language is referred to as **ANSI C**, **Standard C** or **C89**. The second edition of *K&R* describes this standard. Programmers restricted themselves to K&R C for several years even after the publication of ANSI C, mainly due to the continued use of older compilers and because carefully written K&R C code was accepted in ANSI C.

The International Organization for Standardization (ISO) adopted ANSI C as a standard in 1990. This standard is referred to as **C90**. Thus, C89 and C90 refer to the same programming language. The *Normative Amendment 1* to the 1990 C standard was published in 1995 to improve support for international character sets and to correct some details.

In 1999, a new standard for the C language was published jointly by ISO and IEC (International Electrotechnical Commission). This standard is referred to as **C99**. Like its predecessor, C99 also

introduced several new features that were already implemented as extensions in several C compilers. The C99 standard has since been amended three times by *Technical Corrigenda*. C99 is for the most part backward-compatible with C90, but is stricter in some ways. GCC, Sun Studio and other C compilers now support many or all of the new features of C99. However, in spite of the availability of compilers supporting the C99 standard, most of the code written today in C is still based on ANSI C. This is mainly for portability reasons as ANSI C is widely available on almost every platform. In 2007, work began for the next standard of C language, informally called **C1X**.

### 1.5.2 C Programs

The general structure of a simple C program is given below:

*preprocessor directives (#include, #define, etc.)*  
**int main()**  
{  
  *declarations*  
  *executable statements*  
  **return 0;**  
}

Most C programs will usually perform some input/output operations and thus require the following **preprocessor directive**:

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

This line includes the standard input/output **header file stdio.h**. Additional **#include** directives may be used to include other header files such as **math.h**, **ctype.h**, etc. if we use the functionality given in those header files. Chapter 4 covers this topic in more detail. The program may also include **#define** preprocessor directives to define symbolic constants or macros.

C programs will usually contain the **main function**, whose first line is generally written as

```
int main()
```

This line is followed by a block which is the body of the **main** function. It may contain declarations of variables, constants, etc. in the beginning, followed by executable statements. The executable statements include assignment statements, calls to functions (such as **scanf** and **printf**) and control statements such as **if**, **for**, **while**, **do ... while**, etc. Note that all declaration statements must precede the executable statements.

If the **main** function returns an **int** value as indicated by the **int** keyword before **main**, the last line of the **main** function will typically be a **return** statement that returns zero value as shown above. This value indicates to the caller of this program that program execution was successful. If we omit this **return** statement, the compiler may show a warning message.

C programmers often declare the `main` function as

```
void main()
```

The `void` keyword indicates that the `main` function does not return any value. Thus, the `return` statement need not be included in the `main` function.

The program given below prints a message `Hello, world` on the display screen using the `printf` statement. Note that the `main` function does not have any declaration statements.

```
#include <stdio.h>

int main()
{
    printf("Hello, world");
    return 0;
}
```

## Statements and Blocks

The C language supports two types of statements: *declarative* and *executable*. The **declarative statements** provide declarations for various program elements such as variables, constants, etc. The **executable statements** can be executed to produce some action. These include assignment, control and function call statements.

A **block** is a sequence of statements enclosed within curly braces, i. e., `{ }`. Each block usually contains one or more executable statements, which may be preceded by declarations. The general format of a block is shown below.

```
{
    declarations;
    executable_statements ;
}
```

A block is considered as a single statement. The body of a function is always written as a block irrespective of the number of statements within it, i. e., the statements within a function are always included within curly braces. Blocks are also used extensively in control structures. This will be discussed in detail in the subsequent chapters.

The blocks may be nested, i. e., one block may be included inside another. Note that the entities declared within a block (such as variables, constants, etc.) can be accessed only within that block including any blocks nested within it. The part of the program in which an entity can be accessed is called its **scope**. The scope of an entity starts immediately after its declaration and ends with the closing brace of the block in which it is declared.

The entities declared within a block must have unique names, i. e., no two entities can have the same

name. However, a variable declared in a block can be declared in another block including the block contained within it.

### Example 1.12 Nested blocks

Consider an example containing nested blocks given below.

```
{ /* outer block */
    int a = 10, b = 20;
    printf("In outer block: a=%d, b=%d\n", a, b);
    { /* inner block */
        int a = 5, c = 30;
        printf("In inner block: a=%d, b=%d, c=%d\n", a, b);
    }
    printf("In outer block again: a=%d, b=%d\n", a, b);
}
{ /* another block */
    int a = 1, b = 2;
    printf("In another block: a=%d, b=%d\n", a, b);
}
```

The outer block declares variables **a** and **b**, initializes them with values 10 and 20 and prints them using the **printf** statement, the details of which are given in the next chapter. The inner block declares variables **a** and **c** and initializes them with values 5 and 30. It then prints the values of variables **a**, **b** and **c**. The values of variables **a** and **b** are printed again after the inner block is over, i. e., in the outer block. The variables **a** and **b** are declared again and initialized with values 1 and 2, respectively, in another block that follows the first block. The output obtained for this program segment (after including it in a complete program) is shown below.

```
In outer block: a=10, b=20
In inner block: a=5, b=20, c = 30
In outer block again: a=10, b=20
In another block: a=1, b=2
```

Observe that within the inner block, the variable **a** declared within it (local variable) hides the variable **a** declared in the outer block. Thus, we cannot access variable **a** declared in the outer block inside the inner block. Also note that the compiler will report an error if we try to access any of these variables outside the block in which they are declared, unless of course these variables are declared in the block in which we access them.

### Comments

**Comments** are used in a program to provide additional information to the reader. A comment is

delimited with /\* and \*/, i. e., a comment starts with /\* and ends with \*/. A comment is usually written on a line by itself or at the end of a program line. In addition, a comment may span several lines.

Comments are non-executable parts of a program. The C compiler simply skips them. Thus, the compiler will not point out mistakes in comments such as use of misspelled words, incorrect grammar and wrong comments.

Comments are very useful to a program reader. Well-written comments greatly improve the readability of a program. Generally, comments should be short and meaningful. They should contain the information that is not otherwise available in the program itself. Thus, the comment given below is unnecessary:

```
int a, b; /* declare variables a and b of type int */
```

Note that the character sequences /\* and \*/ in a string literal do not start or terminate a comment. Thus, the `printf` statement given below has no comment in it.

```
printf("/* This is not a comment */");
```

### Example 1.13 Comments in a C program

The program given below illustrates the use of comments.

```
/* Program to illustrate the use of comments.  
Program accepts two integer numbers and prints their sum.  
Written by Dr. R. S. Bichkar. Date: 18/1/03. */  
#include <stdio.h>  
  
int main()  
{  
    int a, b; /* input numbers */  
    int c; /*sum of a and b */  
  
    /* read numbers a and b */  
    scanf("%d %d", &a, &b);  
  
    /* determine sum and print it */  
    sum = a + b;  
    printf("Sum: ", sum);  
  
    return 0;  
}
```

Note that the text on the first three lines is a multiline comment used to illustrate the purpose of the program and information about the author and date on which the program was written. The next two

comments explain the purpose of variables used, whereas the last two comments explain the purpose of one or more statements in the program.

ANSI C does not allow the comments to be nested, i. e., one or more comments to be written within another outer comment as shown below.

```
/* Example of nested comments
   /* inner comment */
Again in outer comment */
```

In this example, the comment starts with the `/*` character sequence on the first line and ends with the `*/` character sequence on the second line. Thus, the ANSI C compiler reports an error when it encounters the text on the third line. However, some compilers such as Turbo C/C++ allow the comments to be nested. This feature is very useful to quickly comment a portion of the code that includes one or more comments. To enable the nested comments feature in Turbo C, set *Options → Compiler → Source → Nested comments to on*.

## 1.6 Program Development Environments

As C is a high level language (HLL), we require a translator to convert a C program to the machine language. This is typically done using a C **compiler**. The popular C compilers include Turbo C and GNU C. Note that the C++ programming language is a superset of C. Thus, C++ compilers, such as Turbo C++, GNU C++ and MS Visual C++ can also be used to compile C programs. The GNU C/C++ compilers are available on a large number of operating systems including MS Windows and Linux. The Turbo C/C++ compilers were developed mainly for MS DOS. However, they work in MS Windows environment under the DOS shell. Visual C++ is available in the MS Windows platform only.

Besides compilation, we need to perform several other tasks during the development of a program. These include program editing (entry and modification), testing, debugging, profiling, printing, etc. Although we can use separate tools for these activities, it is very convenient to use an **integrated development environment (IDE)** that provides most of these tools under a single umbrella. Turbo C/C++ and Microsoft Visual C++ are the two popular development environments. In addition, several free development environments are available. These include Dev-C++, Code::Blocks, Eclipse, NetBeans, etc. Dev-C++ and Code::Blocks are mainly used for the development of C/C++ programs. Eclipse and NetBeans are the more powerful industry standard IDEs for the development of state-of-the-art projects in Java, PHP, C/C++ and other languages. We strongly recommend that you use one of these advanced IDEs. In this section, we introduce Turbo C/C++, Dev-C++ and Code::Blocks as the representative development environments.

### 1.6.1 Turbo C/C++

Turbo C, developed by Borland International Inc., is an IDE that can be used for the development of C programs. It provides a compiler, editor, debugger, help system, and standard as well as additional

libraries. Version 2.01 is available for free download from the Borland web site. Fig. 1.23 shows the Turbo C IDE.

The Turbo C editor window allows us to work with two program files at a time. We can switch between them using *Alt+F6*. Mouse support is usually not available. Moreover, the editor provides *Wordstar-like* commands for several operations such as block operations, find and replace, etc. Hence, a person familiar with recent Windows applications such as Microsoft Word may find it somewhat awkward to edit programs in Turbo C.



**Fig. 1.23** Turbo C IDE

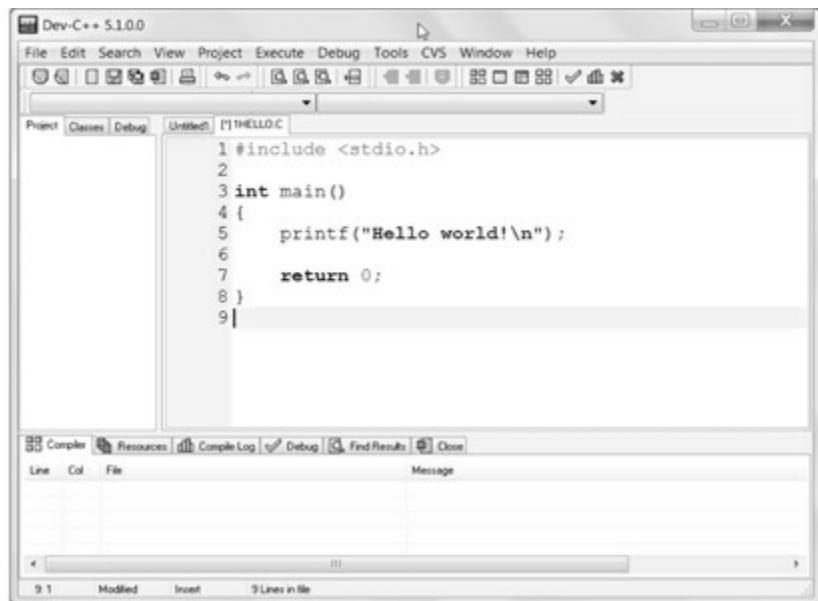
Turbo C++ IDE is very similar to Turbo C IDE. Version 1.01 is available for free download from the Borland's website. Turbo C++ provides several improved features that facilitate program development. First, it allows us to edit several files, each in its own window. We can switch between these files using the *Window* menu or the hot key *Alt+n*, where *n* is the window number. We can perform various operations on these windows such as zoom, size/move, tile, cascade, close, etc.

The Turbo C++ editor provides syntax highlighting that displays various program elements in different colors, improving program readability. It also provides mouse support and hot keys for block operations. We can use the Shift+arrow keys to select a block of text. The block operations that can be performed using hot keys are cut (*Shift+Del*), copy (*Ctrl+Ins*), paste (*Shift+Ins*), clear (*Ctrl+Del*), etc. The user interface for search and replace capability has also been greatly improved.

The help system of Turbo C++ is much better than Turbo C. It provides an index to quickly locate help on a specific topic. It also include several examples which we can copy to the editor window. However, note that the help topics and examples may include some C++ features.

The Turbo C++ files have been grouped in various sub-directories. These include BIN for executable

files, **INCLUDE** for library header files, **LIB** for libraries, **BGI** for graphics files, **DOC** for documentation files, etc. Thus, if Turbo C++ is installed in **C:\TCP**, the directory settings for the include and library directories would be **C:\TCP\INCLUDE** and **C:\TCP\LIB**, respectively.



**Fig. 1.24 Dev-C++ + IDE**

### 1.6.2 Dev-C++

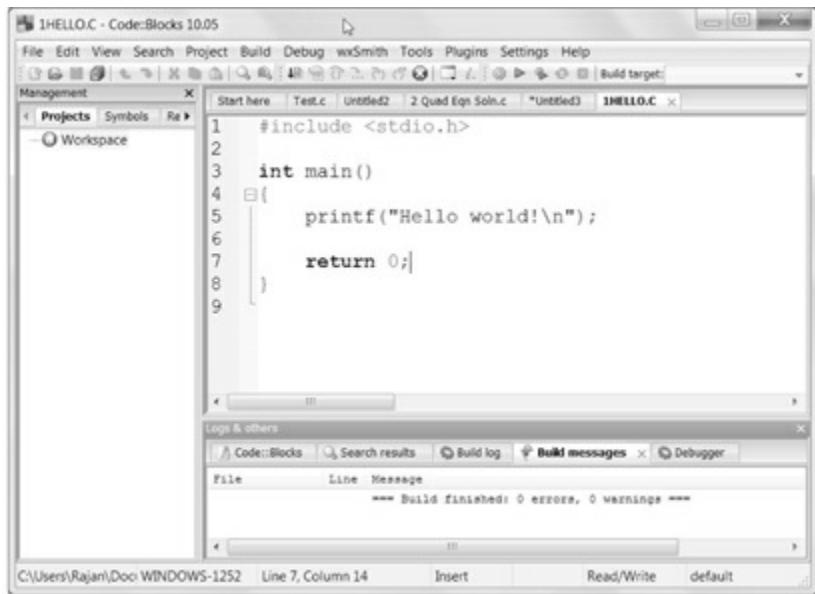
Bloodshed Dev-C++ is a free IDE for the development of C and C++ programs on MS Windows. It is bundled with MinGW (Minimalist GNU for Windows) compiler tools. It is a full-featured IDE that provides menus, toolbars, project workspace, multiple editing windows, configuration dialog boxes, syntax highlighting, code completion, selection of editor font and font size, etc. Dev-C++ allows us to execute standalone program files and create projects that consist of several source files and header files. Fig. 1.24 shows a screenshot of Dev-C++ with the *Hello World* program loaded in the editor window.

The last version of Dev-C++ was 4.9.9.2 released in 2005. It can be downloaded from [www.bloodshed.net](http://www.bloodshed.net). Subsequently, the project was inactive for several years till 2011. The project has now been revived as Orwell Dev-C++ and the latest version available, as of January 2012 is 5.1. It is available for download from [sourceforge.net/projects/orwelldevcpp](http://sourceforge.net/projects/orwelldevcpp) for both 32-bit and 64-bit Windows. The capabilities of Dev-C++ can be extended by using libraries or packages called *devpaks*. Several devpaks are available for download from [devpaks.org](http://devpaks.org) for graphics, compression, animation, etc.

### 1.6.3 Code::Blocks

Code::Blocks is a free, open source and cross-platform IDE that is available on MS Windows, Linux and

Mac OS X. Currently it is oriented towards C and C++. However, it can be used to build applications in several other languages. It supports multiple compilers including GCC and MSVC. Moreover, as it uses plugin architecture, its capabilities and features can be extended. The latest stable version is 10.05, released in May 2010. To install Code::Blocks, download the file `codeblocks-10.05mingw-setup.exe` from the [www.codeblocks.org](http://www.codeblocks.org) website.



**Fig. 1.25** Code::Blocks IDE

The IDE allows several files to be opened simultaneously. It provides several features that include syntax highlighting, code folding, selection of editor font and font size and an integrated to-do list. In addition, it provides code completion and class browser for C++ programs. Code::Blocks gives us the convenience of executing standalone program files or creating a project consisting of several source files and header files. It also supports a powerful debugging facility that includes full breakpoint support, user-defined watches, call stack, CPU registers and local function symbol and argument display.

## 1.7 Advanced Concepts

### 1.7.1 A Quick Tour of the C Language

This section presents four programs to illustrate the various facilities provided by the C programming language. These include preliminary concepts such as variables, data types, operators and expressions as well as more powerful and advanced facilities such as control structures, arrays, functions and pointers. The facilities have been introduced here just to get a feel of what we are going to come across in the rest of the book. Do not panic if you do not understand these concepts in this very quick tour. They will be clear when we study them later. Also, note that this is not a complete tour of C. There are several other

features which have not been introduced here.

### Program 1.1 Welcome to C

Consider a very simple C program given below used to display a text message on the screen. Note that the line numbers on the extreme left are not part of the program and are used in this section to facilitate explanation of the program.

```
1| /* Welcome program */
2| #include <stdio.h>
3|
4| void main()
5| {
6|     printf("Hi friends!\n");
7|     printf("Welcome to the world of C programming.\n");
8| }
```

Now let us understand this program. The first line in the program

```
/* Welcome program */
```

is a **comment**. A comment starts with the character sequence `/*` and ends with the character sequence `*/`. The comments can be included almost anywhere in a program and they are used to explain the program or its parts.

A program line that starts with a hash (#) is a directive for the C **preprocessor** to perform some action. The line

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

instructs the C preprocessor to include the contents of a standard **header file** `stdio.h` in the program. This file contains the information (more formally called declarations) of the input/output (I/O) facilities provided in the C language. This file is required to be included, as it contains the declaration of the `printf` function used in this program. Note that as every program will usually perform some console (i. e., keyboard and display) I/O, this line is usually required in each program. The next line is left blank to improve the readability of the program.

C programs consist of one or more functions. Every C program must contain at least one function, called **main**. In the above example, the code on lines 4 to 8 define the **main** function. The first line of this function

```
void main()
```

specifies what data goes inside the **main function** and what data is returned by it. The keyword `void` before `main` indicates that the function does not return any value and the empty pair of parenthesis, i.

e., ( ) after main indicates that it does not accept any input. Note that the pair of parenthesis after a name identifies it as a function.

The body of the **main** function includes two statements (calls to the **printf** standard library functions) enclosed within the opening brace '{' and the closing brace '}'. These function calls are used to print included text on the screen. Note that unlike some other programming languages such as BASIC and Pascal, C does not provide any built-in functionality for input/output operations. Instead, it is provided in the standard library supplied with the C compiler. The **printf** (print formatted) function is provided in the standard library to display information (numeric as well as text) on the screen. Note that the pair of parenthesis after **printf** identifies it as a function name.

A sequence of characters enclosed within double quotes is called a **character string** or simply a string. It is used to store textual information, e. g., "Apple", "January", "Monday", etc. Each **printf** function in this example accepts a string as the input and displays it on the screen. Note that the double quotes used to delimit the string are not printed and that the character sequence \n in these strings, which is a special character called *newline*, causes the cursor to move to the next line.

Also note that each **printf statement** is terminated with a semicolon. In fact, each statement in C language (but not each line) must be terminated with a semicolon. Remember that a semicolon is not required at the end of a preprocessor directive (line 2) and the first line of a function (line 4).

Remember that one or more statements enclosed within the curly braces is called a **block**. Observe how the statements within the block are usually indented (shifted right) to improve the program readability.

Finally, note that C is a case-sensitive language and that C programs are usually written in lowercase. Thus, if we write the word **include** as **Include**, **main** as **MAIN** or **printf** as **Printf**, the program will not work. However, as in the case of the above program, a program is likely to contain uppercase letters in comments, character strings and identifiers used as names of constants, variables and functions.

When a C program is compiled and executed, the statements within the main function are executed in the order in which they appear. Thus, the execution of the above program begins with the first **printf** statement. This statement prints the string "Hi friends!" on the screen and moves the cursor to the beginning of the next line, where the output of the second **printf** statement appears. Thus, the program displays the following text on the screen:

```
Hi friends!
Welcome to the world of C programming.
```

### Program 1.2 Conversion of Cartesian coordinates to polar form

The Cartesian coordinates  $(x, y)$  of a point can be converted to polar coordinates  $(r, \theta)$  where

$$r = \sqrt{x^2 + y^2} \text{ and } \theta = \tan^{-1}(y/x).$$

Now consider the program given below to convert the Cartesian coordinates of a point to the polar form.

```
1| /* conversion of Cartesian coordinates to Polar form */
2| #include <stdio.h>
3| #include <math.h>
4|
5| int main()
6| {
7|     int x, y;          /* Cartesian coordinates */
8|     double r, theta;   /* Polar coordinates */
9|
10|    /* read coordinates in Cartesian form */
11|    printf("Enter Cartesian coordinates: ");
12|    scanf("%d %d", &x, &y);
13|
14|    /* convert given coordinates to polar form */
15|    r = sqrt(x*x + y*y);
16|    theta = atan((double) y/x);
17|
18|    /* print Polar coordinates */
19|    printf("Polar coordinates are\n");
20|    printf(" r = %5.2f theta = %5.2f\n", r, theta);
21|
22|    return 0;
23| }
```

This program includes two standard header files—`stdio.h` and `math.h` on line 2 and line 3, respectively. The **math.h header file** contains the declarations of mathematical functionality provided in the C standard library such as evaluation of powers, logarithms, trigonometric and hyperbolic functions, etc. It is required in this program, as we have used the `sqrt` and `atan` functions declared in this header file.

This program contains a single user-defined function—`main`. The body of the function comprises of two parts—the declarations (lines 7 and 8) and the executable statements.

C language provides a rich set of **data types** to represent data of different types. A data type decides the range of values and possible operations that can be performed on data of that type. The C data types include four **basic data types** namely, `char`, `int`, `float` and `double` and several variants of these basic data types. The `char` and `int` data types are used to store character and integer data, respectively, whereas `float` and `double` are used to store real number data.

A program uses variables to store the quantities that can change during its execution. Each **variable** is given a name and is associated with a data type using a declaration statement. Each variable must be

declared before it is used. The declaration section of the `main` function in the above program (line 7 and 8) declares four variables: `x` and `y` of type `int` to represent the Cartesian coordinates of a point; and `r` and `theta` of type `double` to represent polar coordinates. Note that as the variable name must start with a letter or underscore, we cannot use  $\theta$  as a variable name.

The `main` function contains four types of statements—`printf` (lines 11, 19 and 20) to print a message or result on the screen, `scanf` (line 12) to read the data from the keyboard, **assignment statements** (lines 15 and 16) to assign a value of an expression to a variable and **return statement** (line 22) to return a value to the calling program.

The execution of this program begins with the first executable statement (line 11). It prints a message requesting the user to enter Cartesian coordinates of a point. The `scanf` statement then reads the values entered from the keyboard in variables `x` and `y`. The assignment statements in lines 15 and 16 calculate the values of polar coordinates `r` and `theta` from the given Cartesian coordinates. Note that the function `sqrt` calculates the square root of its argument, whereas the `atan` function calculates the arctangent. Also note the typecast (`double`) in line 16 used to convert the integer value of variable `x` to type `double` and avoid integer division of `x` and `y`. Observe that symbols `+`, `*` and `/` are used as operators for addition, multiplication and division, respectively. As the multiplication operator has a higher precedence (priority) of evaluation than the addition operator, the expression `x * x + y * y` is correctly interpreted as  $x^2 + y^2$  and not as  $(x^2 + y) y$ . Finally, the `printf` statements on lines 19 and 20 display the polar coordinates.

Observe that the program contains several comments that help us understand it. Also, observe how blank lines are used to separate parts of the program and improve its readability.

### Program 1.3 Determine the maximum and minimum of several numbers

Consider the program given below to accept several integer numbers from the keyboard and print the maximum and minimum of them. This program illustrates several control structures and arrays.

```
1| /* Determine maximum and minimum of several numbers */
2| #include <stdio.h>
3| #define MAX_NUM 50
4|
5| int main()
6| {
7|     float num[MAX_NUM], max, min;
8|     int n, i;
9|
10|    printf("-- Determine maximum and minimum of given numbers
11|
12|    /* accept valid value as array size */
13|    do {
```

```

14|         printf("How many numbers? ");
15|         scanf("%d", &n);
16|         if (n < 1 || n > MAX_NUM)
17|             printf("Error: Invalid array size\n");
18|         } while (n < 1 || n > MAX_NUM);
19|
20|         /* accept the numbers */
21|         printf("Enter %d numbers:\n", n);
22|         for(i = 0; i < n; i++)
23|             scanf("%f", &num[i]);
24|         /* determine maximum and minimum number and print them */
25|         max = min = num[0];
26|         i = 0;
27|         while (i < n) {
28|             if (num[i] > max)
29|                 max = num[i];
30|             else if (num[i] < min)
31|                 min = num[i];
32|             i++;
33|         }
34|
35|         printf("Maximum = %f  Minimum = %f\n", max, min);
36|
37|         return 0;
38|     }

```

The `#define` preprocessor directive on line 3 is used to define `MAX_NUM` as a **symbolic constant**, i. e., a constant having a name, with value 50. The use of such symbolic constants in place of literal constants (such as 50) makes the program easy to read and modify.

An **array** is a group of data items of the same type. The arrays in C are similar to the subscripted variables in mathematics, such as  $a_i$  and  $b_{ij}$ . C allows us to use one-dimensional as well as multidimensional arrays in our programs. Note that in C, each array subscript is written within square brackets after the array name. Thus, the mathematical variables  $a_i$  and  $b_{ij}$  are represented in C as `a[i]` and `b[i][j]`.

To store several numbers entered from the keyboard, we declare in line 7 a **one-dimensional array** called `num` of type `float`. Note that the declaration of an array specifies the maximum array size, i. e., the maximum number of elements it can hold. For array `num`, the maximum size is specified as `MAX_NUM`, with a value 50. The array `num` can thus store a maximum of 50 numbers, each of type `float`. Line 7 also declares two more variables of type `float`—`max` to store the maximum of the given numbers and `min` to store the minimum. The program also declares two more integer variables on line 8—`n` to represent the actual number of elements in array `num` and `i` as a subscript variable to access

array elements. Then the `printf` statement in line 10 prints the purpose of the program.

C language provides two types of **control statements**—conditional statements and loops. **Conditional statements** are used to execute parts of a program depending on the outcome of some condition(s). C language provides two conditional control statements, namely, `if` and `switch`. The **loop statements** allow one or more statements to be executed repeatedly. C language provides three loops: `for`, `while` and `do ... while`.

Lines 13 to 18 constitute a `do ... while` loop. It is used to accept the value of `n`, the actual number of elements in the array, repeatedly until the user enters a valid value, i. e., within the range 1 to  $n - 1$ . This loop contains three statements: `printf`, `scanf` and `if`. The `printf` statement on line 14 displays a message to the user and the `scanf` statement on line 15 accepts the value of `n` from the keyboard. The program then tests this value of `n` using an `if` statement (lines 16 and 17) and displays an error message if it is invalid, i. e., less than 1 or greater than `MAX_NUM`.

Note that the symbol `||` is the logical OR operator. C language also provides other **logical operators** such as `&&` (logical AND) and `!` (logical NOT). Also note that the statements within the `do ... while` loop (lines 14 to 17) are enclosed within the curly braces. As mentioned earlier, such a construct is called a block and is used to group together two or more statements.

The loop control condition for this `do ... while` is specified on line 18 after the keyword `while`. The statements within the `do ... while` loop are executed repeatedly while (as long as) the specified condition evaluates as *true*. Thus, if the user enters an invalid value for `n`, the statements within this loop are executed again. The control leaves the loop when the user enters a valid value. Thus, we ensure that the subsequent part of the program will not accept more numbers than we can fit in the array `num`.

The `printf` statement on line 21 then displays a message to enter the actual numbers. Then a `for` loop is used to accept `n` values and store them in array elements `num[0]` to `num[n-1]`. Note that the array elements are numbered from 0 onwards and not from 1.

The **for loop** is used when the number of repetitions are known in advance. Here, we wish to accept `n` elements of an array. This is considered as `n` repetitions of activity that accepts one number at a time. The `for` loop is set up such that index variable `i` assumes values from 0 to  $n - 1$ . For each value of `i`, the `scanf` statement which accepts the array element `num[i]` or the *i*th array element is executed. Note that code within the parentheses after the `for` keyword has three parts:

1. the initial expression, `i = 0`, which initializes variable `i` to zero when loop is entered,
2. the update expression, `i++`, which causes the value of `i` to be incremented by 1 after each execution of the loop body, the `scanf` statement in this case. Note that `++` is an increment operator and `--` is the decrement operator.
3. the final expression, `i < n`, which causes the statement within the loop (the `scanf` statement) to be executed as long as the value of `i` is less than `n`.

The program then determines the maximum and minimum of n elements in array `num` using a **while loop**, although a **for** loop would have been more suitable in this case. Initially the value of `num[0]` is assigned to variables `max` and `min` in line 25. Then a while loop is set up to process the remaining elements in the array. The loop variable `i` is initialized to zero before the loop and is incremented within the loop. Thus, the variable `i` takes the values from 1 to `n-1`.

For each value of `i`, the **if-else-if statement** is executed in which `num[i]` is first compared with the current value of `max`. If `num[i]` is greater than `max`, the value of `num[i]` is assigned to variable `max` as the new maximum. Otherwise, it is compared (in the `else` clause of the `if` statement) with the variable `min` and if it is less than `min`, it is assigned as the new minimum. After the `while` loop is over, the variables `max` and `min` will contain the maximum and minimum of the first `n` elements in array `num`. Finally, the values of variable `max` and `min` are printed on line 35 using `printf` statement.

#### Program 1.4 Addition of complex numbers

A complex number is represented in the form  $x + iy$ , where  $x$  is the real part and  $y$  is the imaginary part. Now consider the addition of two complex numbers  $x_1 + iy_1$  and  $x_2 + iy_2$ . The result is a complex number whose real part and imaginary part are obtained by adding the corresponding parts of the numbers being added. Thus, the result of the addition is given as  $(x_1 + x_2) + i(y_1 + y_2)$ .

The program to perform the addition of two complex numbers is given below. It introduces advanced concepts in C language such as functions, structures and pointers.

```
1|     /* addition of complex numbers */
2|     #include <stdio.h>
3|
4|     /* structure to represent a complex number */
5|     struct complex {
6|         float re, im;
7|     };
8|
9|     /* function prototypes */
10|    void read_complex(struct complex *a);
11|    void print_complex(struct complex a);
12|    struct complex add_complex(struct complex a, struct complex a
13|
14|    int main()
15|    {
16|        struct complex a, b, c;
17|
18|        printf("Enter complex number a: ");
19|        read_complex(&a);
```

```

20|
21|     printf("Enter complex number b: ");
22|     read_complex(&b);
23|
24|     c = add_complex(a, b);
25|
26|     printf("Addition of given complex numbers: ");
27|     print_complex(c);
28|
29|     return 0;
30|
31|
32| /* read a complex number */
33| void read_complex(struct complex *a)
34|
35| {
36|     scanf("%f %f", &a->re, &a->im);
37|
38| /* print a complex number */
39| void print_complex(struct complex a)
40|
41| {
42|     printf("(%.3f, %.3f)", a.re, a.im);
43|
44| /* add two complex numbers */
45| struct complex add_complex(struct complex a, struct complex b)
46|
47| {
48|     struct complex z;
49|
50|     z.re = a.re + b.re;
51|     z.im = a.im + b.im;
52|
53|     return z;

```

This program can be divided into three main parts: the **global declarations** (lines 2–12), the **main** function (lines 14–30) and the other **user-defined functions** (lines 32–53).

The declarations within a function are called ***local declarations***, whereas the declarations outside any function are called ***external*** or ***global declarations***. The difference is that the local declarations are accessible only within the function (or block) in which they are defined, whereas the global declarations are accessible from the point of declaration to the end of the program file. Thus, the global variables allow the information to be shared across several functions in a program.

Like an array, a **structure** is also used to represent a group of related data items as a single entity. However, the difference is that all the elements in an array must be of the same type, whereas the elements in a structure may be of the same or of different types. The program given above declares a structure named **complex** to represent a complex number. Note that this structure has two members of type **float**—**re** and **im**, used to represent the real and imaginary parts of a complex number.

A C program usually consists of several **functions**. The program given above defines three functions besides the function **main** namely, **read\_complex**, **print\_complex** and **add\_complex**.

As the name suggests, the **print\_complex** function (lines 39–42) prints the complex number specified as its argument. The function accepts the complex number in parameter **a**. The real and imaginary parts of the complex number represented by parameter **a** are accessed using the dot operator (.), as in **a.re** and **a.im**. They are printed using the **printf** statement in line 41.

The **read\_complex** function (lines 33–36) is similar to the **print\_complex** function. However, the parameter is a pointer to **struct complex** as indicated by '\*' in the declaration of parameter **a** (i. e., in **struct complex \*a**). The use of **pointer** allows us to modify the value of the **actual argument** passed to this function. Thus the call to this function from line 22, i. e., **read\_complex(&b)**, actually causes the values to be read in the real and imaginary components of the complex number **b**. Observe that the function **read\_complex** uses a **scanf** statement to read the values of the real and imaginary parts from the keyboard. Also observe that while using a pointer to a structure, the structure members are accessed using the -> operator, as in **a->re** and **a->im**.

The function **add\_complex** (lines 45–53) is used to add two complex numbers and return the result of the addition as a complex number. This function has two parameters, **a** and **b** (both of type **struct complex**) representing the numbers to be added. The function defines a local variable **c** (also of type **struct complex**) to represent the addition of the given complex numbers. The real part of this complex number is calculated in line 49 by adding the real parts of the given complex numbers **a** and **b**. Similarly, the imaginary part is determined in line 50. Finally, the result of the addition is returned using a **return statement**.

If a function is used before its definition, we should declare it before its use. As all functions in this program other than **main** are used before their definition, the declarations of these functions are provided before the **main** function (lines 10–12). Note that these are global declarations.

The **main** function declares three complex numbers **a**, **b** and **c** on line 16. It then reads the complex numbers **a** and **b** in lines 19 and 22 using the **read\_complex** function. Observe that we pass the address of the complex number using the *address of* operator (&), as in **&a**. This is essential as the function expects a pointer to a variable of type **struct complex**. The addition of complex numbers **a** and **b** is then performed to obtain the complex number **c** using the **add\_complex** function. Finally, the complex number **c** is printed using the **print\_complex** function.

Observe that the declaration of structure **complex** must be provided as a global declaration, as is the case in this program, since it is required in more than one function. Finally, note that although the

functions in a program can be defined in any sequence, it is good programming style to define the `main` function first followed by the other functions.

### 1.7.2 Main Memory Organization

The **main memory** in a computer is organized as a sequence of memory locations (Fig. 1.26). Each location has a unique address and it can store a single value. This address is used by the CPU to read its contents or to write new contents in it. The main memory has a random access property, i. e., the CPU can directly access the contents of any memory location and the time required to access a memory location is independent of its address.

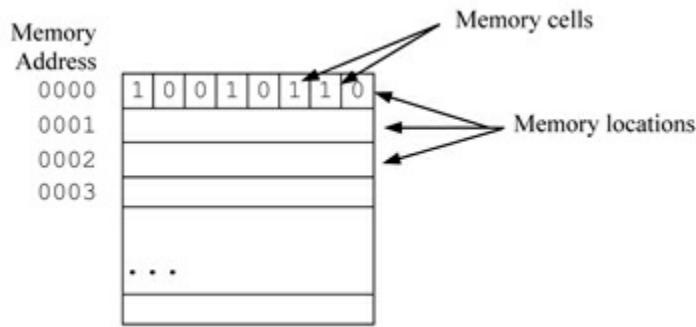
Each memory location further consists of miniature cells (shown in Fig. 1.26 for the first memory location only). Each cell can store two distinct voltage levels (e. g., 0 V and +5V). These voltage levels are assigned binary digits (**bits**) 0 and 1. The memory is usually byte-organized (A **byte** is a group of eight bits.). Each location can store one byte.

The memory capacities are specified in terms of **kilobyte** (KB), **megabyte** (MB) and **gigabyte** (GB). Note that 1 KB equals  $2^{10}$  bytes or 1024 bytes (and not 1000 as we might expect). Similarly, 1 MB is equal to  $2^{20}$  bytes or 1024 KB, 1 GB equals  $2^{30}$  bytes or 1024 MB. An even larger unit is **terabyte** (TB). 1 TB equals  $2^{40}$  bytes or 1024 GB.

### 1.7.3 Binary Number System

A comparison with the decimal number system, which has 10 digits (0, 1, ..., 9), will be useful here. As we already know, the subsequent decimal numbers are formed by repeating these digits in a systematic manner as 10, 11, ..., 19, 20, 21, ..., 99, 100, ...

The **binary number system** has only two digits, 0 and 1, called *bits*, short for binary digits. Subsequent numbers are formed by repeating these two bits. Thus, decimal numbers 2 and 3 are represented as 10 and 11, respectively. Next, the decimal numbers 4 through 7 are represented as 100, 101, 110 and 111. Observe that these numbers are formed by following bit 1 with bit patterns of decimal numbers 0 to 3, written using 2 bits (i. e., 00, 01, 10 and 11). Similarly, the bit patterns for numbers 8 to 15 are formed by following bit 1 with bit patterns of decimal numbers 0 to 7, written using three bits and so on. Table 1.1 shows the binary representations for decimal numbers from 0 to 15 using four bits for each number.



**Fig. 1.26** The byte-organized memory

Note that we can use  $n$  bits to represent a total of  $2^n$  numbers with values from 0 to  $2^n - 1$ . Thus, an 8-bit binary number can represent 256 values from 0 to 255 and a 16-bit binary number can represent 65536 values from 0 to 65535.

**Table 1.1** Representation of decimal numbers 0 to 15 in binary number system

Decimal	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

## Conversion between Binary and Decimal Number System

First, consider the conversion from binary to decimal number system. The rightmost bit in a given binary number is the least significant bit (LS Bit) and the leftmost bit is the most significant bit (MS Bit). The weights of bit positions starting from the LS Bit are given as 1, 2, 4, 8, 16, .... The decimal value of a given binary number is obtained by adding the weights of bit positions in which given binary number has 1 bit as illustrated below.

Bit position :	7	6	5	4	3	2	1	0
Weight ( $2^k$ ) :	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
Weight (Decimal) :	128	64	32	16	8	4	2	1
Binary number :	0	1	0	0	1	0	1	1
Contribution from each bit :	0	64	0	0	8	0	2	1
Decimal value:	75							

Now consider the conversion from decimal to binary number system. Let us convert number 75 to binary number system. This number can be represented using 8 bits as it is less than  $2^8 - 1$ , i. e., 255. We assign values to bit positions starting from MS Bit. The bit at position 7 will be 0 as the given number is smaller than the weight of this bit (128). Next, the bit at position 6, which has a weight of 64, will be 1. After subtracting the weight of this bit position from given number, the remaining value is  $75 - 64 = 11$ . Thus, the bits at positions 5 and 4, which have weights 32 and 16, respectively, will be 0. The bit at position 3, which has a weight 8, will be 1. Now the remaining value is  $11 - 8 = 3$ . Thus, the remaining bits will be 011. Hence, number 75 is represented in binary number system as 0100 1011.

A more systematic method for conversion from decimal to binary number system is illustrated below. In this method, the given number is divided by 2 (integer division) and the divisor is written to the right side of the number and the remainder below the number. The divisor is divided again by 2 and the process is repeated until the divisor becomes 0. The binary representation of given number is obtained by reversing the remainder row as 100 1011, i. e., 0100 1011 using eight bits.

$n$ or $n/2$ :	75	37	18	9	4	2	1	0
$n \% 2$ :	1	1	0	1	0	0	1	
Binary:	100 1011							

Note that the binary representation of the same number using 16 bits is obtained by writing 0s on the left side as 0000 0000 0100 1001.

## Representation of Negative Integer Numbers

Negative numbers are represented using 2's complement which is obtained by adding 1 to the 1's

complement of the binary representation of the given number. Consider the representation of decimal number -5 using 2's complement. The decimal number 5 is represented using 8 bits as 0000 0101. We first take 1's complement as 1111 1010 and then add 1 to it to obtain 1111 1011. Similarly, the representation of -5 using 16 bits can be obtained easily as 1111 1111 1111 1011. Observe that the MS Bit of these numbers is 1, indicating a negative value.

To determine the value of a negative number represented using 2's complement, we first subtract 1 from it and then take 1's complement. The number thus obtained is converted to decimal form and its negative value is taken. Consider, for example, the conversion of a 16-bit representation of -5, i. e., 1111 1111 1111 1011. Subtracting 1 from this number gives 1111 1111 1111 1010, whose 1's complement is obtained as 0000 0000 0000 0101. The decimal value of this number is 5. Thus, the given number is -5.

## Exercises

1. Explain in brief the main functions of the following parts of a computer:  
CPU, Memory, Secondary Storage, Printer
2. Categorize the following as either software or hardware:  
Display, Mouse, Linux, Code::Blocks, Modem, Network card, MS Windows, Joy stick, Printer, Turbo C++, Internet Explorer, Eclipse.
3. Categorize the following into input device, output device and secondary storage:  
Keyboard, CD, Printer, Mouse, Pen drive, DVD, LCD display, Hard disk, Magnetic tape, Joystick, Light pen, Plotter.
4. Explain the following terms in brief:  
Compiler, Editor, Executable file, Syntax error, Bug, Documentation
5. Define the following terms:  
Hardware, Software, Peripheral, Input device, Secondary storage, Main memory
6. What do you understand by the following terms:  
Sequence, Selection, Loop, Structured programming, Modular programming
7. Draw flowchart symbols for the following along with the associated flow lines:  
Input/output, Process, Decision, Terminator
8. Define the role of the following terms in a program code:  
Block, Header file, `main` function, Declaration statement, Comment
9. Select the correct option for the following questions:
  - a. Which of the following is not an input device?
    - i. Printer
    - ii. Scanner
    - iii. Keyboard

- iv. Mouse
- b. Which of the following is a secondary storage?
  - i. RAM
  - ii. Magnetic Tape
  - iii. ROM
  - iv. UPS
- c. Which of the following is not a C/C++ IDE?
  - i. Visual Studio
  - ii. Code::Blocks
  - iii. MS Office
  - iv. Eclipse
- d. Which of the following is called the ANSI C or Standard C?
  - i. K&R C
  - ii. C99
  - iii. C89
  - iv. C1X
- e. Which of the following is the correct sequence for development of a C program?
  - i. Design, Coding, Algorithm, Compilation, Testing
  - ii. Design, Algorithm, Compilation, Testing, Coding
  - iii. Algorithm, Coding, Compilation, Testing, Design
  - iv. Design, Algorithm, Coding, Compilation, Testing
- ). Draw a flowchart for each of the following problems:
  - a. Read the temperature in Centigrade scale and convert it to Fahrenheit scale.
  - b. Accept the radius of a circle and calculate its area and circumference.
  - c. Accept the name and age of a person from the keyboard and display whether he/she is eligible to vote or not.
  - d. Accept the marks of a student in a single subject (out of 100) and determine whether the student passed or failed. Further, determine the class obtained if he/she has passed (Assume: the usual rules for assigning the class).
  - e. Read a number and determine the sum of its digits.
  - f. Accept several numbers from the keyboard and count the even and odd numbers.
  - g. Determine whether a given number is prime or not.

- l. State whether the following statements are correct or not. Also, rewrite the false statements to make them true.
  - a. It is desirable to have a computer with fast and powerful processor and large main memory.
  - b. As comments increase program size and also the time required for its entry, it is a good programming practice to keep the comments to bare minimum.
  - c. A decision construct is used to specify repetitive calculations in a flowchart.
  - d. A block may contain declaration statements written between executable statements.

## Exercises (Advanced Concepts)

- l. Select the correct option for the following questions:
  - a. How many bits are present in 1 KB memory?
    - i. 1000
    - ii. 1024
    - iii. 4096
    - iv. 8192
  - b. Which of the following expression correctly represents 1 MB memory?
    - i. 1000000 bytes
    - ii. 1000 KB
    - iii.  $2^{20}$  bytes
    - iv.  $10^6$  bytes
  - c. Which of the following is equivalent to 1 GB memory?
    - i. 1024 KB
    - ii.  $2^{20}$  KB
    - iii.  $2^{20}$  bytes
    - iv.  $10^6$  bytes
  - d. How many bits will be required to represent a number in the range 0 to 10000?
    - i. 12
    - ii. 13
    - iii. 14
    - iv. 15

2. Convert the following decimal numbers to binary system.  
19, 33, 47, 125, 151, 321
3. Convert the following binary numbers to decimal system.  
1011, 10111, 11010, 101010, 110011, 1000110
4. State the functions used for the following operations:
  - a. Determine square root of a number
  - b. Read data from keyboard
  - c. Print output on display.
5. State the purpose of the following features of C language:
  - a. Standard library function
  - b. **for** loop
  - c. **if** statement
  - d. Array
  - e. Structure
  - f. User-defined function.

---

<sup>†</sup> Subprogram is a smaller program within the main program that performs a specific task independent of the remaining program.

<sup>‡</sup> A sequence of characters enclosed within double quotes is called a **character string** or simply a **string**. It is used to store textual information, e. g., "Apple", "January", "Monday", etc.

# 2 Representing Data

This chapter introduces the features of C language used for representing data in programs. These include character set, keywords, data types, constants, string literals, identifiers, variables and symbolic constants. The *Advanced Concepts* section includes type suffixes, type modifiers and qualifiers, character strings, and constants in octal and hexadecimal form. A beginner may skip this section in the first reading.

## 2.1 Character Set

The **character set** of C language is a set of characters used in C programs. It is summarized in Table 2.1.

**Table 2.1** *The character set of the C language*

Uppercase letters	A ... Z
Lowercase letters	a ... z
Decimal digits	0 ... 9
Space	
Special Symbols	Operator symbols (+ - * /% <> = &   !^~ ?: ,), brackets (( ) [] {}), separators (., ;), delimiters (" ' ) and other symbols (#_\\)
Control Characters	alert (\a), backspace (\b), carriage return (\r), new line (\n), horizontal tab (\t), vertical tab (\v), form feed (\f)
Null character	\0
Extended characters	Locale-specific with implementation dependent values

The **character set** includes uppercase letters ,(A ... Z), lowercase letters (a ... z), digits (0 ... 9),

space, special symbols, control characters and extended characters.

Depending on whether a character can be printed (displayed) or not, it can be categorized as **graphic** and **non-graphic**. Graphic characters are printable characters such as letters, digits and special symbols. A non-graphic character cannot be printed; rather its effect can be seen on the screen or a printer. Control characters such as backspace (\b), tab (\t) and newline (\n) are examples of non-graphic characters.

The special symbols include operator symbols (+ - \* / % < > = & | ! ~ ? : , ), bracket symbols (( ) [ ] { } ), separators ( . ; , ), delimiters ( " ' ) and other symbols (# \ ). They are explained in Table 2.2.

**Table 2.2 Special characters in the character set of the C language**

Character name	Symbol	Description
Plus	+	Addition and unary plus
Minus	-	Subtraction or negation
Asterisk	*	Multiplication and pointer dereference operators
Forward slash	/	Division
Percent	%	Modulo arithmetic (remainder after integer division)
Less than	<	Less than operator
Greater than	>	Greater than operator
Equal	=	Used in Assignment operators
Ampersand	&	Bit-wise and logical AND and address-of operator
Pipe		Bit-wise and logical OR
Exclamation	!	Logical NOT
Carat	^	Logical exclusive OR
Tilde	~	Bit-wise complement
Question mark	?	Used in conditional operator ( ?: )
Colon	:	Used in conditional operator ( ?: )
Comma	,	Comma operator (also used as a separator)
Left and right parentheses	( )	Used in expressions and function calls
Left and right square brackets	[ ]	Used in array subscripts
Left and right curly braces	{ }	Used in a block of code and initialization list
Period	.	Structure member
Semicolon	;	Statement terminator
Double quote	"	String Delimiters
Single quote	'	Character delimiter
Hash	#	Other Symbols
Underscore	_	Used in identifier names
Backslash	\	Used to form Escape sequences

Note that the operator symbols include plus (+) for addition, minus (-) for subtraction and negation, asterisk (\*) for multiplication and pointer dereference operators, forward slash (/) for division, percent (%) for modulo arithmetic, i. e., remainder after integer division, less than (<) and greater than (>) for comparison, equal to (=) for assignment, ampersand (&) for *and* operators and address-of operator, pipe (|) for *or* operators, exclamation (!) for not operator, caret (^) for exclusive or operator, tilde (~) for bit-wise complement, question-colon pair (? :) for conditional operator and comma (,) operator. Note that some symbols are used for more than one purpose.

The control characters are non-graphic characters. They are usually included in character constants and string literals using **escape sequences**. An escape sequence consists of a backslash (\) followed by one or more graphic character, as in \n, \t, etc. The control characters include alert (\a), backspace (\b), carriage return (\r), new line (\n), horizontal tab (\t), vertical tab (\v) and form feed (\f).

Amongst these, newline (\n) and horizontal tab (\t) are the most commonly used. When a **newline** character (\n) is printed on the screen, the cursor moves to the initial or leftmost position on next line. The **tab** character (\t) causes the cursor to move to next tab position, which usually occurs every eight character positions. The **alert** character (\a) causes the computer to produce an audible (beeping sound) or visual alert. Note that the blanks (spaces), newlines, horizontal and vertical tabs, and form feeds are collectively called **whitespace**. The character set also contains a **null character** (\0), which is used as a string terminator. It is a byte having all its bits set to 0.

The extended characters are locale-specific and their values are implementation dependent, i. e., they depend on your machine's configuration. Note that except some control and extended characters, all other characters are represented in memory in a single byte.

Most computers including personal computers (PCs) support the **ASCII character set**. Hence, unless otherwise mentioned, the ASCII code is assumed throughout this book. The ASCII character set has one good property that the digits (0 ... 9) are assigned sequential ASCII values. Similarly, the uppercase letters (A ... Z) and the lowercase letters (a ... z) are also assigned sequential ASCII values. This simplifies several operations on characters and character strings, e. g., comparison for the purpose of sorting, case conversion, etc.

## 2.2 Keywords

A **keyword** is a reserved word having predefined meaning that cannot be changed. The ANSI C standard (C89) defines 32 keywords. These keywords are listed in Table 2.3 according to their categories. This categorization helps us to remember most of the keywords.

The keywords can be grouped into two broad categories: **data types** and **control flow**. A rich set of basic data types and type modifiers provide a programmer with a large number of data types. In addition, a programmer can define new data types.

**Table 2.3** *The ANSI C keywords*

Category		Keywords			
Data types	Basic data types	char	int	float	double
	Type modifiers	short	long	signed	unsigned
	Type qualifiers	const	volatile		
	Storage classes	auto	static	register	extern
	User-defined data types	enum	struct	union	typedef
	Other	void	sizeof		
Control flow	Selection	if	else		
		switch	case	default	
	Loops	for	while	do	
	Transfer of control	break	continue	return	goto

The keywords related to data types include four **basic data types** (`char`, `int`, `float` and `double`), four **type modifiers** (`short`, `long`, `signed` and `unsigned`), two **type qualifiers** (`const` and `volatile`) and four **storage classes** (`auto`, `static`, `register` and `extern`). In addition, the keywords `enum`, `struct`, `union` and `typedef` are provided for defining **user-defined data types** and `void` is a generic type. Also, the `size of` keyword is used to determine memory size of data types, variables, etc.

The control flow group has keywords for selection, looping and transfer of control. These include five keywords for selection (`if`, `else`, `switch`, `case` and `default`), three keywords for loops (`for`, `while` and `do`) and four keywords for transfer of control (`break`, `continue`, `return` and `goto`).

## 2.3 Basic Data Types

During the execution of a program, the computer processes different types of data elements such as numbers, characters, text, etc. By associating a data type with a data element, a programming language restricts the values of that data element and the operations that can be performed on it. The type of a data element is specified using **type specifiers**, which include **basic data types**, **type modifiers** and **type qualifiers**. In this section we will discuss the basic data types. Type modifiers and qualifiers are discussed in Sections 2.9.2 and 2.9.3, respectively.

The C language provides four basic data types: **char**, **int**, **float** and **double**. The `char` data type stores a single character in the machine's character set (ASCII assumed). The `int` data type is used for integer numbers, whereas the `float` and `double` data types are used for floating-point, i. e., real numbers. In addition, C provides a `void` type that is used to indicate absence of type. It is generally used in function declarations.

Every data element requires some memory (RAM) for its storage, which depends upon various factors such as the type of data, the range of possible values and the **precision** (i. e., number of

significant digits in a floating-point value). In C language, the characteristics of data types (i. e., memory size, range and precision) are machine dependent and may vary from one machine to another.

Table 2.4 shows typical characteristics of basic data types in C language. The `char` type requires one byte and has a range of -128 to 127 (assuming it to be `signed` by default). The `int` type may require either 2 or 4 bytes depending on the machine and the compiler used. The `int` type is always signed by default, i. e., it can store both negative and positive values. While using Turbo C/C++, which were developed for 16-bit Disk Operating System (DOS), the `int` type has a size of 2 bytes giving a range of -32768 to 32767, whereas while using Code::Blocks, Dev-C++, Visual C++, etc., which were developed for 32-bit platforms such as MS Windows and Linux, the `int` type requires 4 bytes and provides a much larger range from -2147483648 to 2147483647 (i. e., approximately from -2.1 billion to 2.1 billion).

**Table 2.4** The characteristics of basic data types in the C language

Data type	Meaning	Size (bytes)	Range	Precision
<code>char</code>	a character in C character set	1	-128 ... 127	
<code>int</code>	an integer number	2 <sup>†</sup>	-32768 ... 32767	
		4 <sup>‡</sup>	-2147483648 ... 2147483647	
<code>float</code>	a single-precision (less accurate) floating-point number	4	$3.4 \times 10^{-38}$ ... $3.4 \times 10^{38}$	6
<code>double</code>	a double-precision (more accurate) floating-point number	8	$1.7 \times 10^{-308}$ ... $1.7 \times 10^{308}$	15

<sup>†</sup> on 16-bit platforms such as DOS, e. g., Turbo C/C++ (old versions)

<sup>‡</sup> on 32-bit platforms such as Windows, Linux, etc. (Code::Blocks, Dev-C++, Visual C++, etc.)

The C language implements the IEEE 754 standard for floating-point data. The floating-point numbers are always signed and store both negative and positive values. `float` is a single-precision (i. e., less accurate) type that requires 4 bytes of memory and provides a range of  $3.4 \times 10^{-38}$  to  $3.4 \times 10^{38}$  with a precision of 6 digits. The `double` data type, on the other hand, requires 8 bytes of memory and provides a much larger range from  $1.7 \times 10^{-308}$  to  $1.7 \times 10^{308}$  with a precision of 15 digits. We have mostly used the `float` data type in this book since it is sufficient for most common situations. However, for scientific applications involving very large or very small numbers, we should use `double` data type.

## 2.4 Constants

A **constant** (also called as **literal** or **literal constant**) is an entity whose value cannot be changed during

program execution. C supports four types of constants: *integer constants*, *floating constants*, *character constants* and *enumeration constants*. They are described in Table 2.5. In this section, we will study constants of the first three types. The enumeration constants, which are user-defined constants of type `int`, are discussed in Section 15.1.

**Table 2.5 Types of constants in C language**

Type of constant	Description	Examples
Integer constant	Number having an integer value	10, 1234
Floating constant	Number having a real value	0.15, 2.0, 3.14
Character constant	Single character (represented as a sequence of one or more characters enclosed within single quotes)	'A', '+', '0'
Enumeration constant	User-defined constants of type <code>int</code>	Jan, Feb, Red, Sunday, Monday

## 2.4.1 Integer Constants

An **integer constant** is a number that has an integer value. Integer constants can be specified in three forms: *decimal*, *octal* and *hexadecimal*. The octal and hexadecimal integer constants are discussed in Section 2.9.5.

A **decimal integer constant** consists of a non-zero digit (1 ... 9) optionally followed by one or more digits (0 ... 9). Other characters such as sign, comma, decimal point, space and special symbols are not allowed. Thus, the following are valid decimal integer constants:

5    123    12345    987654    543212345

However, the following are invalid decimal integer constants.

-1    12 34    \$125    0400    10+50    1,50,748

The data type of an integer constant depends on its value. The type is `int` if the value can be represented in it; otherwise, the type is `long int`. Thus, while using Turbo C/C++ which provides a 2-byte `int` type with a range of -32768 to 32767, the type of first three valid integer constants given above (5, 123 and 12345) is `int`, whereas the type of the last two valid constants (987654 and 543212345) is `long int` (which is a 4-byte integer). However, while using Dev-C++, Code::Blocks, Visual C++, GNU C++, etc., which provide a 4-byte `int` type, the type of each valid integer constant given above is `int`.

## 2.4.2 Floating Constants

A floating constant has a **significand part** (also called **mantissa**) that may be optionally followed by an

*exponent part* and a *type suffix*. The significand part may contain a digit sequence representing a *whole number part*, followed by a period (.), followed by a digit sequence representing a *fraction part*, as in *ddd.dddd* where *d* is a decimal digit (0 ... 9). We can omit either the whole number part or the fraction part, but not both and the period must be present if the exponent part is not specified. Several floating constants containing only significand part are given below.

12.345    12. .345    12345678.123456

The exponent part includes the letter **e** or **E**, followed by an optional sign (+ or -), followed by a digit sequence, as in **e±dd** or **E±dd** where ± is the optional + or - sign and *d* is a decimal digit. Thus, the general formats for floating number (excluding the type suffix) are as follows:

*ddd.ddde±dd*              *ddd.ddddE±dd*

The exponent indicates the power of 10 by which the significand is to be scaled. Thus, the value of a floating constant **ddd.ddddE±dd** is given as *ddd.dddd*  $\times 10^{\pm dd}$ . Note that either the period or the exponent part must be present in a floating constant. Several valid floating constants containing exponent part are given below.

1.23e10    1.e5    .1e90    1e20    1e-10

Note that a floating constant cannot contain any other characters such as comma, space, special symbols, etc. Several invalid floating-point constants are given below.

21,345.50    ±12.0    \$12.5    75.2%    500  
3-e10        e-20      1.23E1.5    1.1×10<sup>5</sup>    -12.+34

### 2.4.3 Character Constants

The **character constants** in C language can be grouped into two broad categories: *integer character* and *wide character*. This section discusses integer character constants. The wide character constants are beyond the scope of this book.

An integer character constant is a sequence of one or more characters enclosed in single quotes, as in '**A**'. The characters within the single quote may be any characters (except single quote ', back slash \ or newline character) or an escape sequence. An escape sequence is used to specify a non-graphic character, arbitrary integer value or a special symbol (' " \ ?). Three types of escape sequences can be used: *simple*, *octal* and *hexadecimal*.

A simple **escape sequence** is used to represent either a non-graphic character (such as \n for new line, \t for tab, etc.) or a special symbol (\' for single quote, \\ for backslash, \" for double quote and \? for question mark). Note that double quote and question mark can also be written directly (without using escape sequence). The escape sequence \0 is used to specify a null character. The octal and hexadecimal escape sequences are used less often. They are discussed in Section 2.9.5. Note that

since a newline is not permitted in a character constant, we cannot continue a character constant on the next line.

Several examples of valid character constants are given below.

'0'    'G'    'r'    '+'    ' " '    '\''    '\n'    '\0'    '\\'

However, the following are invalid character constants.

A    \n    ''    ""    '\'    ''    'D'    P'    'AB'

An integer character constant has type **int**. The value of a character constant is equal to its value in the machine's character set. Thus, for a machine using ASCII character set, the values of constants containing letters and digits are as follows: '0' ... '9' (48 ... 57), 'A' ... 'Z' (65 ... 80), 'a' ... 'z' (97 ... 112). However, we need not remember these values as we can directly use the character constants as if they were integer values. Finally, note that the character constant '**0**', which has a value 48 (ASCII), is not the same as integer constant **0**.

## 2.5 String Literals

A **string literal** or a **string constant** (often simply called **string**) in the C language can be grouped into two broad categories: *character string literals* and *wide string literals*. The wide string literals contain wide characters and their discussion is beyond the scope of this book.

A character string literal is a sequence of zero or more characters enclosed in double quotes, as in "**Hello**". The characters within double quote may be any member of the C character set (except double quote ", back slash \ or new line character) or an escape sequence, as discussed in character constants. The only difference is that, in a string literal, a single quote can be included directly (without using escape sequence) but a double quote should be represented using the \" escape sequence.

Each string constant is terminated with a **null character** (\0). Thus, the string "**Hello**" contains six characters: H, e, l, l, o and null (\0). Note that the double quotes are not the part of the string value. A string literal is stored in a character array, i. e., in consecutive character positions in the memory. Thus, the string "**Hello**" is stored in an array as shown below:

H	e	l	l	o	\0
0	1	2	3	4	5

Note that the first array element starts at array index 0. The value of each array element is equal to the value of the corresponding character in the string.

The length of a string is defined as the number of characters in it excluding the **null terminator**, e. g., length of string "**Hello**" is 5 and length of string "**Hello world!**" is 12. Finally, note that adjacent string literals are concatenated to form a single string, e. g., adjacent string literals "**Hello**"

"world!" are equivalent to string "HelloWorld!"

### Example 2.1 String Literals

Several valid character string literals are given below.

""	"Hello"	"\nEnter radius:\a"
" "	"12345"	"He said, \"I am fine\""
"ABC "	"A12+-3.45"	"Arithmetric operators:\t+-*/%"
"\n"	"ABC" "DEF"	"Unexpected\0 termination"

The following points may be noted about these string literals:

1. The string "" contains no character within double quotes. It is called a **null string**. It actually contains only one character – the null terminator.
2. The string " " contains a single space character (and a null terminator). It is not same as the character constant ' '.
3. The string "12345" is not the same as integer constant 12345.
4. The string "ABC " is not the same as the string "ABC". The former has an additional space character at the end. Also, strings "ABC", "Abc", "AbC" and "abc" are all different.
5. The string "\nEnter radius:\a" contains a new line character (\n) in the beginning and alert character (\a) at the end.
6. The string "He said, \"I am fine\"" contains double quotes and uses the \" escape sequences. Its value is He said, "I am Fine".
7. The adjacent strings "ABC" "DEF" will be concatenated into a single string "ABCDEF".
8. The string "Unexpected\0 termination" contains a null character in it. It will be treated as the string "Unexpected" as null character terminates a string.

The following are invalid string literals:

"C:\TC\INCLUDE"	"Apple	'Enter radius:'
"ABC" DEF	12345"	"She said, "Hello""

Recall that to include a backslash character in a string literal we have to use the \\ escape sequence. Thus, the string "C:\TC\INCLUDE" is invalid. It can be correctly written as "C:\\TC\\INCLUDE". Also note that we should use the \" escape sequence to include a double quote in a string literal. Hence, the string "She said, "Hello"" is invalid.

## 2.6 Identifiers

An **identifier** is a name used for a program entity such as variable, symbolic constant, function, user-defined data type, etc. An identifier must start with a letter (a ... z, A ... Z) or underscore (\_) and may be followed by any combination of letters, digits (0 ... 9) and underscore. No other character is allowed. Also, C keywords cannot be used as identifiers. The following are examples of valid identifiers: `a, cost, marks, b1, _msg, c5d2.`

C is a case-sensitive language and treats lowercase and uppercase letters as distinct. Thus, the identifiers `COST, cost, Cost, COST` are all distinct and can be used to represent different entities in a program. However, this should usually be avoided as it makes a program difficult to understand.

Traditionally, C programmers use uppercase letters for symbolic constants (such as PI) and lowercase letters for all other identifiers. Often we have to use compound names for identifiers, e. g., to represent total salary of an employee, we may use the following names: `totsal, tottempsalary, totalsalary, etc.` However, such names are difficult to read and we can use underscore to make them more readable, as in `tot_sal, tot_emp_salary, etc.` Some programmers prefer *capitalized* identifier names such as `totsal, totsal, TotEmpSalary, etc.`

An identifier may be of any length. However, the number of significant characters in an identifier name is dependent on the compiler. ANSI C permits at least 31 significant characters for internal identifiers. However, short meaningful identifier names are preferable, e. g., the use of variable names such as `totsal, tot_sal, tot_salary, etc.`, is preferable to much longer names like `total_salary_of_an_employee.`

Finally, note that the identifiers starting with underscore are usually used in header files provided with C compilers. Hence, we should avoid the use of such identifiers for user-defined entities. We should also avoid the use of names of functions, macros, structures, etc., defined in the standard C library as identifiers for user-defined entities.

### Example 2.2 Identifiers

Several valid identifiers are given below. Observe that, the identifiers in the last line have C keywords embedded within them.

B	Q	_	_X55
Sum	num	d1p23	max3
VecAdd	Mincost	total_salary	CalcRoots
PI	SWAP	MAX_ITEMS	_X1_Y2
case1	constant	float_num	int12

The identifiers given below are invalid. Can you explain why?

`π`      `7wonders`      `%age_marks`      `-ve_number`      `°Celsius`

x.y	amount_in_\$	intr rate	a&b	marks(1)
const	case	short	long	register

The identifiers in first line do not start with a letter or underscore. The identifiers in the second line contain invalid characters and the last line contains C keywords.

## 2.7 Variables

A **variable** is an entity whose value can change during program execution. Each variable has a name and a type associated with it. An identifier is used as a variable name. Thus, the rules for naming variables are the same as those for identifiers. Usually lowercase names are used for variables. The type of a variable determines the range of values that can be stored in it and the memory required for it.

### 2.7.1 Variable Declaration

Each variable must be declared before it is used in a program. A simple variable **declaration statement** consists of a data type followed by one or more variable names separated by commas and terminated with semicolon as shown below.

```
data_type variable1, variable2, ... ;
```

This statement declares variables *variable1*, *variable2*, ... to be of specified *data\_type*. The ellipsis (...) indicates that additional variables may be specified.

Consider the variable declaration statement given below that declares three variables of type **float** to store radius, volume and surface area of a sphere:

```
float radius, volume, surf_area;
```

When a program is compiled this statement causes the required memory to be allocated to the variables declared in it. A declaration which allocates memory for a variable is also called its **definition**. Thus, we can also say that the statement given above defines three variables.

Many programmers prefer only one variable per declaration statement. This makes programs more readable and allows comments to be written explaining the purpose of each variable. Thus the declaration statement given above can be rewritten, along with comments, as shown below:

```
float      radius;      /* radius of sphere */
float      volume;      /* volume of sphere */
float      surf_area;   /* surface area of sphere */
```

C programs consist of one or more functions. A typical C function may require several variables which may be of the same or different data types. The variables used in a function are usually declared within it. As we already know, each variable must be declared before it is used. Also, the declaration statements must precede the executable statements such as assignment statements, control statements and function

calls. Hence, the declarations of all variables used in a function are usually written at the beginning of the function.

The variables may also be declared at the beginning of any block and not just the block defining the function body. Once a variable is declared, it can then be used within the block in which it is declared. Finally note that within a block, a variable cannot be *defined* more than once, not even with a different data type.

### Example 2.3 Variable declarations

a) Consider the following variable declarations:

```
char ch1, ch2;  
int a, b, c;  
float cost;
```

There are three variable declaration statements. These statements declares six variables: **ch1** and **ch2** of type **char**; **a**, **b** and **c** of type **int**; and **cost** of type **float**. Note that all the variable names are unique, i. e., each variable name is *defined* only once. Also observe the commas separating the variable names and semicolons at the end of each declaration statement.

b) Consider another example of variable declarations:

```
char a;  
int a, b, c  
flot x y;
```

There are four errors in these statements: (a) multiple declarations for variable **a**, (b) missing semicolon in the second statement, (c) incorrect data type **flot** instead of **float** and (d) no comma separator between variables **x** and **y**.

## 2.7.2 Variable Initialization

The C language allows us to initialize one or more variables in a declaration statement. The variable **initialization statement** takes the following form:

```
data_type variable1 = expr1, variable2 = expr2, ...;
```

This statement declares variables *variable1*, *variable2*, ... of the specified *data\_type* and initializes them with the values of expressions *expr1*, *expr2*, ..., respectively. Expressions are covered in the next chapter. In this section, we consider a restricted form of variable initialization in which constants are used to initialize variables.

To initialize two or more variables with the same value, we cannot use the initialization statement as shown below:

```
data_type variable1 = variable2 = expr, ...; /* wrong */
```

Note that the initialization expression must be written separately for each variable even if all the variables need to be initialized with the same expression. Also, while declaring multiple variables in a single statement, it is possible to initialize only a few of them. However, such declarations should be avoided, as they are difficult to read. It is good programming practice to initialize only one variable in each initialization statement.

The value of an initialization expression must be of appropriate type so that the variable is correctly initialized. Otherwise, the compiler will convert, if possible, the value to desired type using conversion rules that will be discussed in Section 4.4.1 (Mixed mode expressions).

Finally, note that the C language does not initialize the variables declared inside a block (i. e., **automatic variables**) with any default values. Such uninitialized variables will contain **garbage values** (random values). C compilers will usually show a warning message if we use the value of an uninitialized variable.

#### Example 2.4 Variable initialization

a) Consider the declarations given below.

```
int a = 10, b = 20;
float x = 1.25;
char ch1 = 'A', ch2;
```

These statements define five variables (**a** and **b** of type **int**, **x** of type **float**, and **ch1** and **ch2** of type **char**) and initializes four of them with literals of the appropriate type. The variables **a**, **b**, **x** and **ch1** are initialized with values **10**, **20**, **1.25** and **'A'**, respectively. However, variable **ch2** is not initialized. If these statements are written inside a block, variable **ch2** will have a garbage value.

b) Can you identify the errors in the initialization statements given below?

```
float c = d = 5.0;
int a = 10; b = 20;
```

There are two errors in these statements. In first statement, a single literal is used as the initializer for two variables; whereas in the second statement, two initializations are separated by a semicolon instead of a comma. These declarations can be written correctly as follows:

```
float c = 5.0, d = 5.0;
int a = 10, b = 20;
```

## 2.8 Symbolic Constants

A **symbolic constant** or a **macro** is a name given to a sequence of characters using the **#define** **preprocessor directive**. The symbolic constants allow us to give meaningful names to literal constants

and expressions making the programs more readable. The format of the `#define` statement, which is written on a line by itself, is as follows:

```
#define name replacement_text
```

where *name* is an identifier used as the macro and *replacement\_text* is any sequence of characters. The macro names are usually written in uppercase to distinguish them from variable names. For example, we can define symbolic constant **PI** to represent the mathematical constant  $\pi$  as

```
#define PI 3.14
```

Once a *name* is defined as a symbolic constant or a macro, it can be used in the rest of the program in place of *replacement\_text*. Thus, once the symbolic constant **PI** is defined in a program as shown above, we can use it in the rest of that program instead of the value 3.14. During program compilation, C preprocessor substitutes any occurrence of macro *name* with the corresponding *replacement\_text*. However, occurrences of *name* in string literals and in other identifiers will not be substituted. Note that this substitution does not affect the C source file, i. e., the program file.

The use of macros simplifies program maintenance as well. For example, if we wish to use a more accurate value of **PI** as 3.1415927, we need to modify it in only one place – the `#define` statement, rather than searching each occurrence of value 3.14 and replacing with 3.1415927. Note that this is very useful when we cannot automatically replace all occurrences of a value with another. For example, a program written with 100 as the value for maximum marks in a subject as well as the maximum number of students in a class and we wish to modify the program for a maximum of 200 students.

A program may define several macros. Although definitions of these macros may be placed anywhere in a program, it is a good practice to place them at the beginning of a program to improve its readability.

`#define` is not an assignment statement. Thus, the assignment operator (=) is not required between macro *name* and *replacement\_text*. Also, `#define` statement should not be terminated with a semicolon unless it is a part of the replacement text. A beginner should avoid these common mistakes.

Finally, note that C allows a macro to be defined in terms of other macros defined earlier. This feature enables us to write macros in more organized and readable form.

### Example 2.5 Symbolic constants

a) Consider the following definitions of symbolic constants:

```
#define MAX_MARKS 100
#define TAB '\t'
#define PI 3.1415927
#define INST "Indian Institute of Technology, Kharagpur"
```

These statements declare four symbolic constants: **MAX\_MARKS**, **TAB**, **PI** and **INST**. It is easy to

understand that the use of names such as `MAX_MARKS` and `PI` in subsequent programs instead of constants `100` and `3.1415927`, respectively, makes the program easier to understand.

**b)** Consider another example given below.

```
#define PI 3.1415927
#define PI_BY_2 PI/2
#define PI_STR "PI is a mathematical constant"
```

The constant `PI_BY_2` is defined in terms of constant `PI` defined in the previous line. The preprocessor substitutes the occurrence of `PI` in `PI/2` with the replacement text, i. e., `3.1415927`. Thus, the value (i. e., replacement text) of `PI_BY_2` becomes `3.1415927/2`. The preprocessor does not evaluate this expression. It simply substitutes each occurrence of `PI_BY_2` in the rest of the program with `3.1415927/2`.

The symbolic constants `PI_STR` and `INST` can be used as short forms for longer strings. Note that the preprocessor does not replace the occurrences of `PI` in symbolic constants `PI_BY_2` and `PI_STR` and in the string "`PI is a mathematical constant`".

## 2.9 Advanced Concepts

This section discusses some of the advanced (or less frequently used) concepts related to data representation that include *type suffixes*, *type modifiers* and *qualifiers*, character strings and constants in the octal and hexadecimal forms. A beginner may skip this section and return to it later after studying more important concepts presented in subsequent chapters.

### 2.9.1 Type Suffixes

The C language allows a **type suffix** to be specified immediately after an integer constant. These suffixes include *unsigned suffix* `U` or `U` and *long suffix* `L` or `L`. Thus, `100U`, `100L` and `100UL` are valid integer constants of type `unsigned int`, `long int` and `unsigned long int`, respectively. The C language also allows the suffixes (`F`, `f`, `L` or `l`) to be specified after a floating number. In the absence of a suffix, the type of a floating constant is `double` and typically requires 8 bytes. The `F` (or `f`) suffix specifies the type of constant as `float`, whereas the suffix `L` (or `l`) specifies the type as `long double`. Thus, `1.5`, `1.5F` and `1.5L` are constants of type `double`, `float` and `long double`, respectively.

### 2.9.2 Type Modifiers

The C language provides four data **type modifiers**, namely, `short`, `long`, `signed` and `unsigned`. These modifiers can be used in conjunction with the basic data types (`char`, `int`, `float` and `double`) to provide several additional data types with different range and memory sizes. These

additional data types are particularly useful when we require higher range compared to what is provided by the basic data types. They are also useful when we wish to optimize the performance of our program (i. e., minimize memory requirements and/or maximize execution speed). Table 2.6 summarizes the data type modifiers.

The **signed** and **unsigned** modifiers can be used only with the **char** and **int** types, as in **unsigned int**, **signed char**, etc. The **signed** type can store both positive and negative values, whereas an **unsigned** type can store only non-negative values. As a result, the largest value allowed by an **unsigned** type is almost twice that of the corresponding **signed** type.

**Table 2.6 Data type modifiers in the C language**

Basic type	Modifiers	Modified type
char	signed, unsigned	signed char, unsigned char
int	signed, unsigned, short, long	signed int, unsigned int short int, long int signed short int, unsigned short int signed long int, unsigned long int
float	---	---
double	long	long double

The **short** and **long** modifiers can be used with the integer types (**int**, **signed int** and **unsigned int**), as in **short int**, **unsigned long int**, **signed short int**, etc. The **long** type may provide a larger range at the cost of additional memory, whereas the **short** type may require lesser memory with a limitation of reduced range. Consider the variable declarations given below:

```
unsigned int a, b;
signed short int c;
unsigned char d, e;
long double f;
```

These statements declare six variables **a** and **b** of type **unsigned int**, **c** of type **signed short int**, **d** and **e** of type **unsigned char** and **f** of type **long double**. If these variables are declared within a block, they will have garbage values. We can also initialize some or all of these variables using suitable values.

Note the following points about type modifiers:

1. We cannot use both **signed** and **unsigned** modifiers in a type. Also, we cannot use both **short** and **long** modifiers in a type.
2. While using the modifiers with **int**, we can omit the **int** keyword. Thus, **long int** can be simply written as **long** and **unsigned short int** as **unsigned short**.

- 3. The type modifiers may be used in any order. Thus, `unsigned long int` may also be written as `long unsigned int`, `int long unsigned`, `long int unsigned`, etc. However, most programmers use the following sequence: `signed/unsigned` followed by `short/long` followed by `int`.
- 4. As floating-point numbers are signed by default, the `signed` and `unsigned` modifiers are not allowed with `float` and `double` types. The `short` modifier is also not used with these types.
- 5. No modifier can be used with the `float` type and only the `long` modifier can be used with `double`, as in `long double` (or `double long`) to provide a larger range and higher precision than `double`, of course at the cost of additional memory.

Does it sound quite complicated? Fortunately, most of these modified types are not really required in the programs that we usually write. Hence, to keep the things simple, we have used only three basic data types, namely, `char`, `int` and `float` in most of the programs in this book. However, a C programmer is expected to know the details (i. e., memory requirements, range, precision, etc.) of these modified data types.

Note that all integer and character data types are collectively referred to as **integral types**; the `float`, `double` and `long double` data types are referred as **floating-point types**; and the integral and floating-point data types are collectively referred as **arithmetic types**.

### **Memory Size, Range and Precision for Data Types**

ANSI C language does not specify the exact **range** and memory size for data types as well as precision for floating-point types. This allows the compiler designer to select the most appropriate values for these parameters for the target machine. Thus, these characteristics of data types are machine dependent. However, C standard specifies that integral data types have the following relation for range and memory requirements:

`char ≤ short int ≤ int ≤ long int`

Thus, the range (and memory requirement) of `long int` data type is at least equal to that of the `int` data type, which in turn must be at least equal to that of `short int`. Similarly, the range of `short int` must be at least equal to that of `char`. Table 2.7 gives the typical memory requirements for integral data types. Similarly, the relationship for range, precision and memory sizes for floating-point data types is as follows:

`float ≤ double ≤ long double`

**Table 2.7 Typical sizes and ranges for integral data types**

Data type	Size (bytes)	Range for signed	Range for unsigned
char	1	-128 ... 127	0 ... 255
short int	2	-32768 ... 32767	0 ... 65535
int	2 <sup>†</sup>	-32768 ... 32767	0 ... 65535
	4 <sup>‡</sup>	-2147483648 ... 2147483647	0 ... 4294967295
long int	4	-2147483648 ... 2147483647	0 ... 4294967295

<sup>†</sup> on 16-bit platforms such as DOS, e. g., Turbo C/C++(old versions)

<sup>‡</sup> on 32-bit platforms such as Windows, Linux, etc. (Code::Blocks, Dev-C++, Visual C++, etc.)

The typical characteristics of the `float` and `double` data types are given in Table 2.4. For a `long double` data type, the typical characteristics are 10 bytes of memory, 19-digit precision and a very large range from  $3.4 \times 10^{-4932}$  to  $1.1 \times 10^{4932}$ .

### 2.9.3 Type Qualifiers

ANSI C provides two **type qualifiers** (`const` and `volatile`) that control how the variables may be accessed or modified. The `const` qualifier is used to specify that the data element is constant, i. e., it does not change, whereas the `volatile` qualifier is used to specify that the data element may change at any time because of some external sources, e. g., the variables representing input ports of a computer. This section discusses the `const` qualifier.

We often require constants such as 3.1415927 (i. e.,  $\pi$ ) to be used in our programs. As extensive use of such literals makes a program difficult to read and maintain, it is a good idea to give suitable names to such literals. Although we can store such a value in variable, it is not a good idea to do so, as it may be unknowingly changed by a careless programmer as illustrated below:

```
double pi = 3.1415927;
pi = 1.12;      /* Oops! */
```

We can also define a **symbolic constant** (discussed in the next section) to represent such a literal using the `#define` preprocessor directive as shown below:

```
#define PI 3.1415927
```

However, the drawback of symbolic constants is that the compiler is not aware of them as they are processed by the preprocessor. As a result, we may face difficulties while debugging such programs. There is, however, a better way to represent such literals – using the `const` data type qualifier in the variable initialization statement as shown below:

```
const double pi = 3.1415927;
```

Once a variable is declared a **const variable**, the C compiler does not allow us to subsequently modify its value. Thus, the compiler will report an error for the following statement as `pi` is declared a **const** variable.

```
pi = 1.12; /* not permitted */
```

Note that as we cannot modify the value of a **const** variable after it is declared, we must initialize it when it is declared. We can define multiple **const** variables in a single declaration statement, using separate initializer expression for each variable, as follows:

```
const data_type variable1 = expr1, variable2 = expr2, ...;
```

#### 2.9.4 Character Strings

We often have to process **character strings** (textual information) in our programs, e. g., name and address of a person, description of an item in inventory, etc. We require variables to store such strings so that we can manipulate their values. However, C does not provide any basic data type to store textual data. Instead, we have to use an array of type **char**. This section quickly introduces how arrays can be used to store strings without going into too much detail. This will enable us to introduce strings quite early in this book.

#### Declaration

To represent a character string, we use a one-dimensional array of type **char** declared as

```
char str [size];
```

where `str` is the array name and `size` is a constant expression specifying the array size. As C uses null-terminated strings, the array should have one extra position for the null terminator. Thus, if we wish to store string constant "Hello" in variable `msg`, we should reserve (at least) six character positions as

```
char msg[6];
```

We can declare several array variables in a declaration statement as shown below.

```
char str1 [size1], str2 [size2], ...;
```

As in case of ordinary variables, character strings are declared at the beginning of the function (or block) in which they are used.

#### Initialization

A character array can be initialized with a string constant as in

```
char str [size] = string_constant;
```

The programmer should ensure that the array *size* is sufficient for *string\_constant* (including the null terminator). If the array size is more than what is required, the elements at the end of the array are unused. This allows the string to grow during the execution of a program.

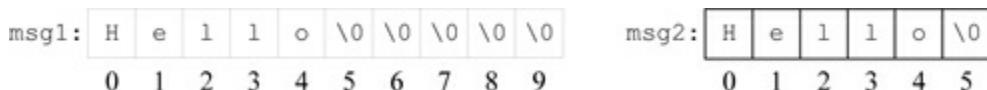
It is also possible to omit the array size during initialization as in

```
char str[ ] = string_constant;
```

In this case, the C compiler allocates the exact number of character positions required to store *string\_constant* in string *str* including the null terminator. Consider the initialization statements given below:

```
char msg1[10] = "Hello";
char msg2[] = "Hello";
```

The first statement declares array *msg1* of size 10 and initializes it with string constant "Hello". Note that the last four array elements are unused and are initialized with null character as their values are not specified. The second statement, however, declares array *msg2* of the required size, i. e., 6, and initializes it with string constant "Hello". The contents of arrays *msg1* and *msg2* are shown below.



C also permits initialization of several character arrays as in:

```
char str1 [] = string_constant1, str2 [] = string_constant2, ... ;
```

where *str1*, *str2*, ... are character arrays which are initialized with the string constants *string\_constant1*, *string\_constant2*, ..., respectively.

## 2.9.5 Constants in Octal and Hexadecimal Form

The octal number system uses eight digits 0 ... 7. A hexadecimal number system uses sixteen digits: 0 ... 9 and A ... F (or a ... f). The values of hexadecimal digits A ... F (or a ... f) are from 10 to 15 as A = 10, B = 11, C = 12, D = 13, E = 14 and F = 15. In this section, we study the integer and character constants in octal and hexadecimal form.

An **octal integer constant** starts with digit 0 and may be followed by one or more octal digits (0 ... 7). Thus, the following are valid octal integer constants:

0        012        0123        0543212345

The value of an octal constant is computed base 8. The value of octal constant 012 is 10 in decimal

system ( $1 \times 8 + 2 = 10$ ) and value of 0123 is 83 ( $= 1 \times 8^2 + 2 \times 8 + 3$ ) in decimals. The following are invalid octal integer constants.

123      05678      098765

A **hexadecimal integer constant** starts with either **0x** or **0X** and is followed by one or more hexadecimal digit. Thus, the following are valid hexadecimal integer constants:

0x12      0X3AB      0x12af

The value of a hexadecimal constant is computed at base 16. Thus, the value of **0x12** is 18 decimal ( $= 1 \times 16 + 2$ ) and the value of **0X3AB** is 939 decimal ( $= 3 \times 16^2 + 10 \times 16 + 11$ ). The following are invalid hexadecimal integer constants:

1A3B      0xBCG      010B      X123

An **octal character constant** is represented as an octal number (i. e., backslash followed by one, two or three octal digits) enclosed in single quotes. Thus, the following are valid octal character constants:

'\0'      '\7'      '\16'      '\123'

A **hexadecimal character constant** is a hexadecimal number (**0x** or **0X** followed by a sequence of hexadecimal digits) enclosed in single quotes. The following are valid hexadecimal character constants:

'0x10'      '0X1A'      '0x3b'      '0xAB'

An *octal integer constant* starts with digit 0 and may be followed by one or more octal digits (0 ... 7), e.g., **0**, **012**, **07654**, etc., whereas a *hexadecimal integer constant* starts with either **0x** or **0X** and is followed by one or more hexadecimal digits (0 ... 9 and A ... F or a ... f), e. g., **0x12**, **0X3AB**, **0xA10**, etc. These forms are convenient while performing bitwise operations on data and while dealing directly with machine hardware, e. g., values of registers and memory locations, memory and I/O port addresses, etc. The octal and hexadecimal constants are not required in most simple programs.

## Exercises

1. State the names of the following characters and explain their use in C:

\*    %    ?    ,    !    ~    ^    \n    \t    \0

2. Which of the following are not reserved words in C?

- a. **for**
- b. **integer**

- c. printf
  - d. newline
  - e. If
  - f. signed
  - g. bell
  - h. constant
  - i. pi
  - j. structure
  - k. short
  - l. unary
3. Which of the following are invalid integer constants in decimal notation? Give reasons.
- a. 10!
  - b. 12,345
  - c. ±100
  - d. 10+20
  - e. 0
  - f. 10.
  - g. 123L
  - h. -12345
  - i. Rs. 125
  - j. 55 Kg
  - k. 25°
  - l. 165cm
  - m. 0XAB
  - n. 0123
  - o. 2000H
  - p. 9876543210
  - q. 100000
  - r. 32768

- s. **1UL**
- t. "123"
- l. Several constants are given below. State which of them are invalid floating constants in decimal form along with reasons.
  - a. 123
  - b. 23,456.0
  - c. 1.5e10
  - d. 1.2+3.2
  - e. .123E5
  - f. 1.333333333
  - g. 0.000
  - h. ±.125
  - i. 5.
  - j. .12 34
  - k. 012
  - l. -1E-5
- j. Several floating constants are given below. State with reasons, which of them are invalid constants in exponential (or scientific) notation?
  - a. 100
  - b. 2,456.0
  - c. 1e1
  - d. 1.2d3
  - e. -0.123E5
  - f. 1.333333333
  - g. .E3
  - h. .125E
  - i. 5.e0
  - j. .134e±15
  - k. 1.2e-5.25

- l. `5e-5000`
5. State with reasons which of the following are invalid character constants?
- `'+'`
  - `"A"`
  - `'\n'`
  - `'\k'`
  - `char`
  - `'a'`
  - `"char_const"`
  - `'''`
  - `' Z '`
  - `'3'`
  - `'\'`
  - `'''`
  - `p`
  - `'π'`
  - `'/'`
  - `"""`
7. Which of the following are invalid string literals and why?
- `'A'`
  - `""`
  - `"a, b, c"`
  - `" "AB" and "CD" "`
  - `"Hello"`
  - `"\n\t\ a"`
  - `s1" "s2"`
  - `"Orange'`
  - `{str_const}`
  - `" "`

- k. Hello World
  - l. ABC'
  - m. 'const'
  - n. ("+-\*/")
  - o. "Transpose a'"
  - p. /\* Comment"
3. State the types of following C constants.
- a. 1.00E-010
  - b. -123
  - c. "345"
  - d. 'R'
  - e. -9876.123
  - f. 123456789
  - g. "Hello"
  - h. '\n'
4. Which of the following are invalid identifiers and why?
- a. ABC
  - b. "Id"
  - c. \_X\_
  - d. long
  - e. A5B1
  - f. 3sum\_avg
  - g. Marks
  - h. interest\_rate
  - i. alpha
  - j. unit-cost
  - k. \_123
  - l. H.R.A.
  - m. payment 1

- n.  $\pi$ by2
  - o. PI
  - p. student's\_marks
- ). Using appropriate variable names, give the declarations for the quantities given below.
- a. Radius, diameter, area and circumference of a circle
  - b. Marks in five subjects and their sum and average
  - c. Initial velocity, firing angle, maximum height attained and the maximum distance traveled by a projectile
  - d. Coordinates of a line segment and its length
  - e. Amount deposited in a bank, interest rate, duration of deposit and compound interest earned
- l. Declare suitable variables for the quantities given below and initialize them with specified values:
- a. Three integer numbers with values 0.
  - b. Two integer numbers with values 10 and 20, two real numbers with values 1.2 and 2.3
  - c.  $\alpha = 100$ ,  $\beta = 125.50$ , ch = 'A'
  - d. First number =  $1.2345 \times 10^{10}$ , second number =  $1.0 \times 10^7$  and operator = '+'
  - e. Deposit = Rs. 10,000.00, interest rate = 10%, duration = 5 years
  - f. Age = 20 years, height = 1.65 meters, weight = 60 kg
2. Define symbolic constants for the following values:
- a.  $\pi = 3.142$ ,  $\pi/2$ ,  $\pi/4$
  - b. Gravitational constant  $g = 9.81$ , speed of light =  $3 \times 10^8$  m/s
  - c. Days per week, months per year
  - d. For a subject, minimum marks = 0, maximum marks = 100, passing marks = 40
  - e. For a school, maximum number of classes = 7 (first to seventh), maximum number of divisions per class = 5 (divisions A to E) and maximum number of students in a class = 30
3. Select correct option for the following questions:
- a. What is the minimum value of a variable of type int in Turbo C?
    - i. 0
    - ii. -32767
    - iii. -32768
    - iv. -1e34

- b. How many bytes are required to store a character constant?
- i. 1
  - ii. 2
  - iii. 3
  - iv. 4
- c. How many bytes are required to store a newline character, i. e., '\n' ?
- i. 1
  - ii. 2
  - iii. 3
  - iv. 4
- d. Which of the following C keyword is not related to control flow?
- i. **for**
  - ii. **return**
  - iii. **volatile**
  - iv. **break**
- e. Which of the following is not a C keyword?
- i. **struct**
  - ii. **continue**
  - iii. **long**
  - iv. **stop**
- f. How many bytes are required to store a character string?
- i. 16
  - ii. One more than number of characters in it
  - iii. Equal to number of character in it
  - iv. Three more than number of characters in it
- l. Identify the errors, if any, in the declaration given below.
- a. **int abc;**
  - b. **Float a, b;**
  - c. **char CH1, CH2;**
  - d. **int single, double, triple;**

- e. int a = 10; b = 20;
  - f. float const = 9.81;
  - g. double d = -.1e-10;
  - h. single s = 1.5;
  - i. char char1=char2='\\n';
  - j. int \_a, \_b, \_c;
  - k. #define MAX\_STUD = 100
  - l. #define newline \\n
  - m. #define PI = 3.14;
  - n. #define HELLO printf("Hello world")
5. State whether the following statements are true or false. Also rewrite the false statements to make them correct.
- a. A reserved word cannot be used as a variable name but it may be a part of a variable name.
  - b. When variables are declared, the arithmetic variables (char, int, float, etc.) are initialized to zero and strings are initialized to null values.
  - c. The char data actually stores an integer number.
  - d. The float and double are exactly identical data types.
  - e. We must declare every variable in our program before we use it.
  - f. Use of very long variable names often results in memory inefficient programs.
  - g. All variables used in a block of C code must be declared at the beginning.
  - h. The variable names must be written using only the lowercase letters.
  - i. The symbolic constants may be written using either lowercase or uppercase letters.
  - j. The maximum number of characters in a C string is limited by the range of int data type, e. g., in Turbo C, the string can have maximum of 32767 characters in it.

## Exercises (Advanced Concepts)

1. State whether the following statements are true or false. Also rewrite the false statements to make them correct.
- a. The short type modifier enables us to save space in our programs.
  - b. As the basic data types in C language are sufficient for most programming situations, we need not

study the other types provided by the type modifiers and qualifiers.

- c. Using escape sequence, we can include only non-graphic characters in a string literal.
- 2. Several integer constants in octal and hexadecimal notation are given below. Identify invalid constants. Justify your answers.
  - a. 0123
  - b. 01, 234
  - c. 5432
  - d. 0x0000
  - e. 0678
  - f. 0
  - g. 0X12AB
  - h. oxA0B1
  - i. 0xF GH
  - j. 01234UL
  - k. 0x10H
  - l. 0xFFFFFFFF
- 3. State the types of the following C constants.
  - a. 1.0F
  - b. 12UL
  - c. 100u
  - d. 5L
  - e. -1.23E-100
  - f. 123456U
  - g. 1.2L
  - h. 1.23
- 4. Declare suitable variables and initialize them with the specified values:
  - a. Name of a country (India) and its capital (Delhi)
  - b. Name of the K&R book (The C Programming Language) and names of its authors (Brian Kernighan and Dennis Ritchie)
  - c. Names of popular games (Cricket, Football, Basketball, Chess)

5. Select correct option for the following questions:
- Which of the following is not a type modifier?
    - long
    - const
    - unsigned
    - signed
  - Which of the following data type can be used to represent the value  $-1.2e-400$ ?
    - float
    - double
    - long double
    - none of these
  - How many bytes are required to store a data of signed character?
    - 1
    - 2
    - 3
    - 4
  - Which of the following is a constant of type double?
    - 1.2D
    - 1.2
    - 1.2F
    - 1.2L
  - Which of the following integer constant is incorrect?
    - 123
    - 123U
    - 123L
    - 123S
5. Identify the errors, if any, in the declaration given below.
- int unsigned abc;
  - long float a = 1F, b = 2f;
  - long char CH1, CH2;

- d. `short unsigned su = 123su;`
  - e. `int a = 0x10;`
  - f. `long double const = 9.81L;`
  - g. `short Long;`
  - h. `unsigned u = 100U;`
7. State whether the following statements are true or false. Also rewrite the false statements to make them true.
- a. A type suffix can be used to modify the value of a constant.
  - b. The type modifiers are provided in C to enable a programmer to save memory.
  - c. The `const` qualifier can be used to define a symbolic constant.

# 3 Arithmetic Operators and Expressions

Almost all nontrivial C programs contain data manipulations. The purpose of this chapter is to introduce the elements of C language useful for data manipulation. These include the operators and expressions.

The commonly used operators are introduced first with an emphasis on arithmetic operators. An elaborate treatment of expressions in the next section will enable reader to understand properly the intricacies of writing expressions, which is one of the basic foundations of C programming. The next section covers assignments. Numerous short examples are presented throughout this chapter. The complete programming examples will be presented in subsequent chapters when the input-output facilities are covered.

The *Advanced Concepts* section covers several topics to help the reader better gain a better understanding of the intricacies of operators and expressions. A beginner may skip this section in the first reading.

## 3.1 Operators

**Operators** are used to connect **operands**, i. e., constants and variables, to form **expressions**. For example, in the expression `a+5`, `a` and `5` are operands and `+` is the operator. Depending on the number of operands on which an operator operates, the operators in C language can be grouped into three categories: *unary* operators, *binary* operators and *ternary* operators. A **unary operator** takes only one operand, as in `-x`, whereas a **binary operator** operates on two operands, as in `a+b`. A **ternary operator** takes three operands, as in `(a > b) ? a : b`, where the conditional expression operator `(?:)` is a ternary operator.

Depending on how the operators are used with the operands, there are three forms: *infix* operators, *prefix* operators and *postfix* operators. An **infix operator** is used between the operands, as in `a+b` and `c*d`. A **prefix operator** is used before an operand, as in `-a`, whereas a **postfix operator** is used after an operand, as in `y--`, where `--` is the decrement operator.

C language provides a very powerful set of operators. The commonly used operators include *arithmetic*, *relational*, *equality*, *logical* and *assignment* operators. They are summarized in Table 3.1. The arithmetic and assignment operators are discussed in this chapter. The relational, equality and logical operators are discussed in Chapter 5.

**Table 3.1** Commonly used operators in the C language

Operator category	Description	Operators
Arithmetic operators	Perform arithmetic operations, namely, addition (+), subtraction (-), multiplication (*), division (/), modulo arithmetic (%), increment (++) , decrement (--), negation (-) and unary plus (+)	+ - * / % ++ --
Relational operators	Perform comparison operations, namely, less than (<), greater than (>), less than or equal to (<=) and greater than or equal to (>=)	< > <= >=
Equality operators	Perform equality (==) and inequality (!=) tests	== !=
Logical operators	Join simple conditional expressions using AND (&&), OR (  ) and NOT (!) operations to form compound conditional expressions (Boolean expressions)	&&
Assignment operators	Assign a value (of a constant, variable or expression) to a variable	= += -= *= /= %=

### 3.1.1 Arithmetic Operators

**Arithmetic operators** are used to perform arithmetic operations on arithmetic operands, i. e., operands of integral as well as floating type. Recall that an integral type includes all forms of `char` and `int` types, whereas the floating-point types include the `float`, `double` and `long double` types. These operations include addition (+), subtraction (-), multiplication (\*), division (/), modulo arithmetic (%), increment (++) , decrement (--), unary plus (+) and unary minus (-). They can be grouped into three categories: *unary* operators, *multiplicative* operators and *additive* operators. The arithmetic operators are summarized in Table 3.2.

#### Additive and Multiplicative Operators

The **additive operators** include *addition* (+) and *subtraction* (-) operators. They are binary infix operators that operate on two arithmetic operands, as in `a + b` and `a - b`.

The **multiplicative operators** include three operators: *multiplication* (\*), *division* (/) and *modulo arithmetic* (%), i. e., remainder after integer division. These are binary infix operators that operate on two arithmetic operands, as in `a * b`, `a / b` and `a % b`.

The multiplication and division operators operate on any arithmetic operands—both integral and floating-type. As the multiplication symbol is not explicitly written in mathematical expressions, we tend to forget it in C expressions. However, remember that the multiplication operator must be explicitly written in a C expression. Thus, the mathematical expression *abc* should be written in C language as `a * b * c`.

**Table 3.2** Arithmetic operators in the C language

Operator group	Operator	Description	Example	Associativity
Unary operators	+	Unary plus	+a	Right to left
	-	Unary minus	-a	
	++	Prefix increment Postfix increment	++x x++	
	--	Prefix decrement Postfix decrement	--x x--	
Multiplicative operators	*	Multiplication	a * b	Left to right
	/	Division	a / b	
	%	Modulo arithmetic	a % b	
Additive operators	+	Addition	a + b	Left to right
	-	Subtraction	a - b	

The result of the division operator (/) depends on the type of operands. If both the operands are integral, the result is truncated to the integer value, e. g., 7/4 evaluates as 1. However, if either or both operands are floating-type, the result is also floating-type, e. g., the expressions 7.0/4, 7/4.0 and 7.0/4.0 evaluate as 1.75. The modulo arithmetic operator works only with integral operands such as the various `int` and `char` types. It gives the remainder after integer division, e. g., 7 % 5 evaluates as 2, whereas 13 % 5 evaluates as 3.

The C language does not provide exponentiation operator available in programming languages like FORTRAN and BASIC. Quantities involving small powers such as  $a^2$  and  $b^3$  are represented using direct multiplication, as in `a*a` and `a*a*a`, respectively. The expression  $x^y$  is written using standard library function `pow` (for *power*) as `pow(x, y)`. The commonly used standard library functions in C language are discussed in the next chapter.

## Unary and Arithmetic Operators

The unary arithmetic operators include *unary plus* (+), *unary minus* (-), *increment* (++) and *decrement* (--). These operators operate on arithmetic operands. Unary plus and unary minus are prefix operators, e. g., `-a`, `+10.25`, etc. The unary minus operator negates the value of its operand, whereas the unary plus operator has no effect on the value of its operand. Note that the symbols for the unary plus and unary minus operators are the same as those of the addition and subtraction operators, respectively.

The **increment operator** (++) increments the value of the operand by 1, while the decrement operator (--) decrements the value of the operand by 1. These operators must be written without a space between the two '+' or '-' symbols. They can only be used with modifiable **Ivalues** (such as variables) and not with constants and expressions.

The increment and decrement operators can be used either in prefix form, as in `++x`, or in postfix form, as in `x++`. Although both the forms modify the value of the operand by 1, there is a subtle difference. In the prefix form, the value of the operand is first incremented and then used, whereas in

the postfix form, the value of operand is first used and then incremented. These operators are explained in detail in Sections 3.3.6 and 3.4.2.

### 3.1.2 Precedence and Associativity of Operators

The operator **precedence** and **associativity** rules specify the order in which operators in an expression are bound to the operands. These rules enable us to interpret the meaning of an expression in an unambiguous manner.

Table 3.3 gives the arithmetic and assignment operators in the order of their precedence which is as follows: unary, multiplicative, additive and assignment. Thus, the unary arithmetic operators have the highest precedence. They are followed by multiplicative operators, which in turn have higher precedence than additive operators. The assignment operators have lower precedence than all the arithmetic operators.

**Table 3.3** Precedence and associativity of arithmetic and assignment operators in C language

Operator group	Operators	Associativity
Unary operators	+ - ++ --	Right to left
Multiplicative operators	* / %	Left to right
Additive operators	+ -	
Assignment operators	= += -= *= /= %=	Right to left

An operator having higher precedence is bound to its operand(s) before the operators having a lower precedence. For example, in the expression  $a + b * c$ , the  $*$  operator, which has higher precedence than the  $+$  operator, is bound first to its operands. Thus the expression is evaluated as  $a + (b * c)$  and not as  $(a + b) * c$ .

All operators in an operator group have equal precedence. Thus, all unary operators ( $+ - ++ -$ ) have equal precedence. The same is true with multiplicative operators ( $* / %$ ), additive operators ( $+ -$ ) and assignment operators ( $= += -= *= /= %=$ ).

The **associativity** of operators decides the order in which operators at the same precedence level are bound to operands. The unary and assignment operators have *right-to-left* associativity. If an expression contains more than one unary operator, they are bound to operands from right to left. Remember that only unary operators, assignment operators and conditional operator ( $?:$ ) have *right-to-left* associativity. All other operators in C language have *left-to-right* associativity.

The multiplicative operators have *left-to-right* associativity. They are bound to operands in the order of their occurrence in an expression from left to right. For example, the expression  $a*b/c*d$  is interpreted as  $((a*b)/c)*d$ , i. e., multiplication of  $a$  and  $b$  is performed first followed by division and second multiplication. Similarly, the additive operators also have *left-to-right* associativity.

## 3.2 Expressions

An **expression** is a proper sequence of operators and operands that often specifies computation of a value, as in  $a + b * c$ . This expression contains two operators (+ and \*) and three operands (a, b and c). The operands may be constant literals, symbolic constants, variables, function calls or parenthesized sub-expressions. Note that a single constant, variable or a function call returning a value is also considered an expression. In this chapter, we study the *arithmetic expressions*, i. e., the expressions containing only arithmetic operators and operands.

### 3.2.1 Simple Arithmetic Expressions

An **arithmetic expression** contains only arithmetic operators and operands. We know that the arithmetic operators in C language include unary operators (+ - ++ --), multiplicative operators (\* / %) and additive operators (+ -). The arithmetic operands include integral operands (various `int` and `char` types) and floating-type operands (`float`, `double` and `long double`).

#### Example 3.1 Simple arithmetic expressions

a) Several valid arithmetic expressions are given below. Assume that variables a and b are of type `int`.

5	-1.23	-a + b / 5	3.142 * a * b
'A' + 2	a++	a % 10 + b % 10	a * a + b * b

Observe that the operators are used correctly. The unary operators in expressions  $-1.23$ ,  $a++$  and  $-a + b / 5$  correctly operate on a single operand. The unary minus (-) is used as a prefix operator whereas the increment operator (++) is used in the postfix form. All other operators (multiplicative and additive) are used as infix operators and they operate on two operands. Also, the modulo-arithmetic (%) operator operates on integer operands as required. The expression ' $A$ ' + 2 is valid and its value is equal to 67, as the ASCII value of character ' $A$ ' is 65.

Note that the operands and operators need not be separated by spaces. For example, the last two expressions in the first row can also be written as  $-a+b/5$  and  $3.142*a*b$ . However, the spaces are often used to improve readability.

b) The expressions given below are invalid.

ab      + a b      a++ --b a // b      a % 10 - / b

The first two expressions are invalid because the operands  $a$  and  $b$  are not connected by any operator. This is true even for the third expression ( $a++ --b$ ) as the  $++$  and  $--$  operators operate on operands  $a$  and  $b$ , respectively, but there is no operator connecting these operands.

The last two expressions are invalid as they contain two binary operators in succession. These

operators operate on a single operand instead of two. Note that the `-` operator in last expression cannot be interpreted as a unary operator as it does not precede an operand.

c) Can you tell which of the following expressions are valid?

`+a +b`    `a++ - b`    `a++ + ++b`

Surprisingly, all are valid C expressions although they do not appear to be so. In expression `+a +b`, the first `+` operator is a unary operator that operates on variable `a`, whereas the second `+` is a binary operator that connects variables `a` and `b`.

Now observe the space between the two `'-'` symbols in the second expression. They cannot be interpreted as a decrement operator. As a result, the second `-` symbol is interpreted as a unary minus operator, whereas the first `-` is interpreted as a subtraction operator that operates on two operands `a` and `-b`.

The two consecutive operators in expression `a % 10 /- b` does not pose a problem as the `-` operator is interpreted as a unary operator and the `/` operator operates on two operands (`a % 10` and `-b`). However, the space between `-` and `b` is misleading.

In the last expression, observe the spaces separating the middle `+` symbol from the increment operators that operate on variables `a` and `b`. The middle `+` is interpreted as a binary operator that operates on `a` and `b`.

d) Several simple mathematical equations are given below along with equivalent C expressions. Assume that the symbolic constant `PI` has been defined as

```
#define PI 3.1415927
```

Quantity	Mathematical expression	C expression
Simple interest on fixed deposit	$\frac{pnr}{100}$	<code>p * n * r / 100.0</code>
Surface area of a sphere	$4\pi r^2$	<code>4 * 3.1415927 * r * r</code>
Volume of a sphere	$\frac{4}{3}\pi r^3$	<code>4.0 / 3 * PI * r * r * r</code>
Conversion of temperature from $^{\circ}\text{C}$ to $^{\circ}\text{F}$	$\frac{9}{5}c + 32$	<code>9.0 / 5 * c + 32</code>
Sum of digits of a two digit integer number <code>n</code>		<code>n / 10 + n % 10</code>

Observe that the literal 4 in expression 3 is written as floating constant 4.0 to avoid the integer division of 4 and 3. Similarly, constant 9 in expression 4 is also written as a floating constant. The sum of digits of a two-digit integer number is written as `n / 10 + n % 10`. Since `n` is assumed to be an integer variable, the expression `n / 10` performs integer division and determines the digit at the 10's place,

whereas the expression  $n \% 10$  determines the digit at the unit's place.

## Evaluation of Simple Arithmetic Expressions

We use the operator *precedence* and *associativity* rules to determine the meaning and value of an expression in an unambiguous manner. Recall that the operators in an expression are bound to their operands in the order of their precedence. If the expression contains more than one operator at the same precedence level, they are associated with their operands using the associativity rules. Table 3.3 summarizes these rules for arithmetic and assignment operators.

If the given expression is simple, we can often directly convert it to its mathematical form and evaluate it. However, if the given expression is complex, i. e., it contains several operators at different precedence levels, we need a systematic approach to convert it to a mathematical equation and evaluate it. The steps to convert a given valid C expression to its mathematical form (if possible) and to evaluate it are as follows:

1. First determine the order in which the operators are bound to their operands by applying the precedence and associativity rules. Note that after an operator is bound to its operand(s), that sub-expression is considered as a single operand for the adjacent operators.
2. Obtain the equivalent mathematical equation for given C expression (if possible) by following the operator binding sequence (obtained in step 1).
3. Determine the value of the given expression by evaluating operators in the binding sequence.

The steps to determine operator binding in an arithmetic expression are explained below with the help of the expression  $-a + b * c - d / e + f$ .

1. The unary operators (unary +, unary -, ++ and --) have the highest precedence and right-to-left associativity. Thus, the given expression is first scanned from right to left and unary operators, if any, are bound to their operands. The order is indicated below the expression as follows:

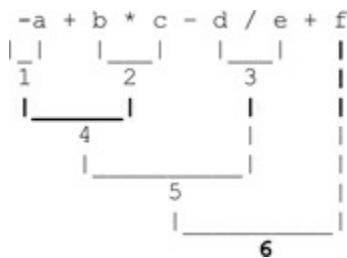
$$\begin{array}{c} -a + b * c - d / e + f \\ | \quad | \\ 1 \end{array}$$

2. The multiplicative operators (\*, / and %) have the next highest precedence and left- to-right associativity. Thus, the expression is scanned from left-to-right and the multiplicative operators, if any, are bound to their operands as shown below. (Observe that after completion of the above step, sub-expressions  $-a$ ,  $b * c$  and  $d / e$  will be operands for the remaining operator bindings.)

$$\begin{array}{c} -a + b * c - d / e + f \\ | \quad | \quad | \quad | \\ 1 \quad 2 \quad 3 \end{array}$$

3. The additive operators (+ and -) have the next highest precedence and left-to-right associativity. Hence, the expression is scanned from left-to-right and the additive operators, if any, are bound to their operands as shown below. Observe that the operands for the first + operator are the sub-

expressions  $-a$  and  $b * c$ . Similarly, the operands for the  $-$  operator are  $-a + b * c$  and  $d / e$ .



Now we can write the mathematical equation for the given C expression by following the operator binding sequence as shown below:

$$\begin{aligned}
 & -a + b * c - d / e + f \\
 & \quad | \quad | \quad | \quad | \\
 & -a \quad bc \quad \frac{d}{e} \quad | \\
 & \quad | \quad | \quad | \\
 & \quad \underline{-a+bc} \quad | \quad | \\
 & \quad \quad | \quad | \\
 & \quad \quad \underline{-a+bc-\frac{d}{e}} \quad | \\
 & \quad \quad \quad | \quad | \\
 & \quad \quad \quad \underline{-a+bc-\frac{d}{e}+f} \quad |
 \end{aligned}$$

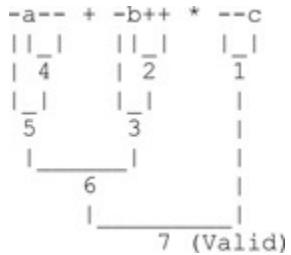
Now the given expression can be evaluated, again by following the operator binding sequence, as shown below. Assume that the variables  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$  and  $f$  are of type `float` and are initialized with values as  $a = 1.5$ ,  $b = 2.0$ ,  $c = 3.5$ ,  $d = 5.0$ ,  $e = 2.5$  and  $f = 1.25$ .

$$\begin{aligned}
 & -a + b * c - d / e + f \\
 & \quad | \quad | \quad | \quad | \\
 & -1.5 \quad 7.0 \quad 2.0 \quad | \\
 & \quad | \quad | \quad | \\
 & \quad \underline{5.5} \quad | \quad | \\
 & \quad \quad | \quad | \\
 & \quad \quad \underline{3.5} \quad | \\
 & \quad \quad \quad | \quad | \\
 & \quad \quad \quad \underline{4.75} \quad |
 \end{aligned}$$

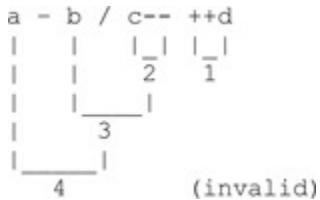
Remember that except for some operators (`&&` `||` `?:` and the comma operator), the C language does not specify the order of evaluation of sub-expressions. Thus, the sub-expressions at the same level can be evaluated in any order. For example, the sub-expressions  $-a$ ,  $b * c$  and  $d / e$  in the above expression can be evaluated in any order.

The procedure explained above can also be used to check the validity of an expression. The given expression is valid if we arrive at a single operand (or value) after all the operators in the given

expression are considered. Consider a more complex arithmetic expression:  $-a- + -b++ * --c$ . This expression appears to be invalid due to the excessive use of operators. It contains three operands  $a$ ,  $b$  and  $c$  and seven operators, five of which are unary ( $-$ ,  $++$  and  $--$ ) and the other two are binary operators ( $+$  and  $*$ ). However, using the operator binding steps, we can easily verify that it is a valid expression:



Now consider the expression  $a - b / c-- ++d$ . Let us now apply the operator binding steps to this expression.



As we did not arrive at a single operand, the expression is invalid. Observe that the cause of invalidity is the missing operator between  $c--$  and  $++d$ , which should connect steps 4 and 1.

### 3.2.2 Parenthesized Expressions

We have seen that the order in which the operators in an expression are bound to the operands is decided by the operator *precedence* and *associativity* rules. We can change this order using **balanced parentheses** in an expression. For example, in the expression  $a + b * c$ , multiplication is performed first followed by addition operator. However, in expression  $(a + b) * c$ , addition is performed first followed by multiplication.

An expression may contain one or more balanced parentheses. Moreover, the parentheses may be nested, i. e., one or more pairs of parentheses may be included in an other pair of parentheses. Note that we cannot use curly braces  $\{ \}$ , square brackets  $[ ]$  or angle braces  $<>$  in place of parentheses.

#### Simple Parenthesized Expressions

This section considers the expressions that do not contain nested parentheses. Two examples of such expressions are given below.

$$(a + b) / c - d \quad a + b * (c + d) / (c + 1 / d)$$

The number of left parentheses in each expression is equal to the number of right parentheses and they are properly balanced, i. e., for every left parenthesis, there is a corresponding right parenthesis.

Also observe that all operators are used as binary infix operators, i. e., they have two operands, one on either side. Note that a parenthesized sub-expression acts as a single operand, e. g., the division operator in the first expression operates on  $(a+b)$  and  $c$ .

### Example 3.2 Simple parenthesized expressions

Several valid expressions containing non-nested pairs of parentheses are given below, along with their equivalent mathematical equations:

- a)  $-x / (b + c) + y / (b - c)$
- b)  $(x + a * b) / (y - a / b)$
- c)  $(a + b + c) / a * b + (b * b + c * c)$
- d)  $a + b / (-a + b) * (a * d + e) - c / d$

The equivalent mathematical equations for these C expressions are given below.

$$\text{a) } \frac{-x}{b+c} + \frac{y}{b-c} \quad \text{b) } \frac{x+ab}{y-\frac{a}{b}} \quad \text{c) } \frac{(a+b+c)b}{a} + b^2 + c^2 \quad \text{d) } a + \frac{b(ad+e)}{b-a} - \frac{c}{d}$$

Now consider the following expressions:

$(a + b) (c - d)$	$a + (b - c) d$
$a \% (b * (c + d))$	$(a + b) / c - d)$
$a / [b * c]$	$\{a + b\} / \{c + d\}$

All the expressions are invalid. The expressions on the first line have missing operands. A pair of parentheses is considered a single operand. Thus, two pairs of parentheses in the first expressions must be connected by a suitable operator. The same is true with the pair of parentheses and variable  $d$  in second expression.

The expressions on the second line are invalid as the parentheses are not properly balanced. Finally the expressions on the last line are invalid as square brackets and curly braces are used instead of parentheses.

### Evaluation of simple parenthesized expression

The evaluation of simple parenthesized expressions is performed as follows. If the given expression contains a single pair of parentheses, all operators contained in it are bound first to their operands using precedence and associativity rules as explained earlier. If the expression contains two or more pairs of parentheses which are not nested within one another, these pairs are considered in the left-to-right order. Finally, the remaining operators in the expression, if any, are bound to their operands using precedence and associativity rules.

### Example 3.3 Evaluation of simple parenthesized expressions

Consider the evaluation of expression  $a + b / (c - d)$ . It contains a single pair of parentheses. The subtraction operator (which is the only operator in this pair of parentheses) is bound first to operands  $c$  and  $d$ . Then remaining operators in the expression are bound to their operands in the order of precedence: division first followed by addition. The order in which the operators are bound to operands is shown in Fig. 3.1a. The given expression is thus equivalent to the mathematical equation  $a + \frac{b}{c-d}$ . If we know the values of variables  $a$ ,  $b$ ,  $c$  and  $d$ , the value of the given expression can now be determined.

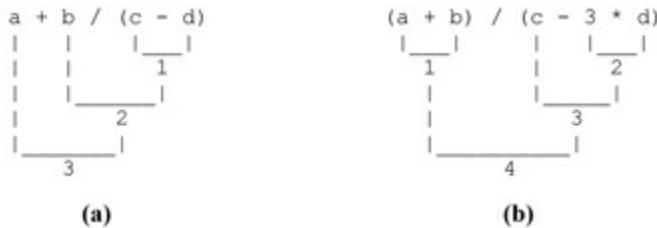


Fig. 3.1 Order of binding operators to operands in simple parenthesized expressions

Now consider another expression containing two pairs of parentheses:  $(a + b) / (c - 3 * d)$ . The order in which operators in this expression are bound to operands is shown in Fig. 3.1b. Initially, the sub-expression in first pair of parentheses is considered followed by sub-expression in second pair of parentheses. Note that the multiplication operator in second pair of parenthesis is bound before the subtraction operator as it has higher precedence. Finally, the division operator is bound to its operands (two parentheses). Given expression is thus equivalent to the  $a+b$  mathematical equation  $\frac{a+b}{c-3d}$ .

### Expressions Containing Nested Parentheses

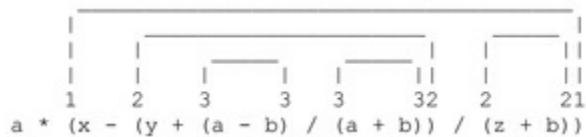
An expression may contain **nested pairs of parentheses**, i. e., pair(s) of parentheses within other pair(s) of parentheses. This allows us to write C expressions representing more complex mathematical equations. Two examples of such expressions are given below along with their mathematical equivalent equations:

- $a + 1 / (b + 1 / (c + 1 / d))$
- $a * (x - (y + (a - b) / (a + b)) / (z + b))$

These expressions are equivalent to the following mathematical equations:

$$\text{a) } a + \frac{1}{b + \frac{1}{c + \frac{1}{d}}}$$
$$\text{b) } a \left( x - \frac{y + \frac{a-b}{a+b}}{z+b} \right)$$

To evaluate such expressions, we must first determine how the parentheses are nested. To determine the nesting level of each pair of parentheses in an expression, we number the parentheses as shown in Fig. 3.2 (for the expression (b) above), in which the matching left and right parentheses are assigned the same number. The nesting of parentheses is also indicated above the expression for better understanding of this numbering method. The parenthesis numbers indicate the level of nesting, level 1 being the outermost pair of parentheses and the highest level being the innermost pair of parentheses. Also note that the pairs of parentheses at the same level of nesting have the same number assigned to them.



**Fig. 3.2 Numbering nested parentheses in an expression**

The parentheses are numbered using the following procedure:

1. The leftmost parenthesis is assigned level 1 if it is an opening parenthesis; otherwise, it is assigned level 0.
2. If an opening parenthesis is followed by another opening parenthesis, the level is increased by 1.
3. If a closing parenthesis is followed by closing parenthesis, the level is reduced by 1.
4. In other cases (i. e., an opening parenthesis followed by a closing parenthesis or vice versa), the level remains unchanged.

If the parentheses in an expression are properly balanced, all parenthesis levels will be nonzero positive integers. A zero or a negative parenthesis level indicates an error in the expression. Also note that the leftmost and the rightmost parentheses should have level 1 assigned to them.

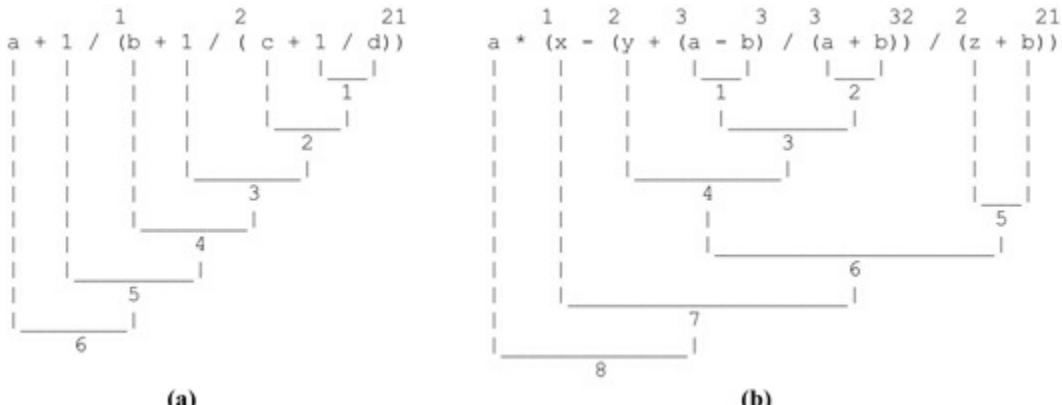
### ***Evaluation of expressions containing nested parentheses***

Writing mathematical equation for a C expression containing nested pairs of parentheses as well as its evaluation is quite difficult, at least for a beginner. To simplify this process, follow the steps given below:

1. First assign the level numbers to the parentheses as explained above.
2. Start from the innermost pair of parentheses and bind the operators to their operands by applying the precedence and associativity rules. If an expression contains two or more pairs of parentheses at the same level of nesting, they can be considered in the left-to-right order. Thus, for the expression in Fig. 3.2, the parenthesized sub-expression  $(a - b)$  can be considered before  $(a + b)$ .
3. Then repeat the above step for the next innermost pairs of parentheses until the all the operators in the outermost pair of parentheses are bound to their operands.
4. Finally, the remaining operators in the expression are bound to their operands.

### Example 3.4 Evaluation of expressions containing nested parentheses

Consider the expression  $a + 1 / (b + 1 / (c + 1 / d))$ . This expression contains two pairs of parentheses nested within one another. The parentheses are numbered as shown in Fig. 3.3a. First the operators in the inner pair of parentheses (level-2) are bound to their operators using the precedence rules (/ followed by +). Next the operators in level-1 parentheses are bound to their operands (/ followed by +). After all the operators in outermost parentheses are bound to operands, the remaining operators in the expression are bound to their operands (/ followed by +). The order in which the operators in this expression are bound to operands is also shown Fig. 3.3a.



**Fig. 3.3 Order of bounding operands to operators in expressions containing nested parentheses**

Note that the expression in the inner pair of parentheses is equivalent to the mathematical equation  $c + \frac{1}{d}$ . The expression in the pair of parentheses at level 1 is equivalent to equation  $b + \frac{1}{c + \frac{1}{d}}$  and the given expression is equivalent to equation  $a + \frac{1}{b + \frac{1}{c + \frac{1}{d}}}$ . If we know the values of variables  $a$ ,  $b$ ,  $c$  and  $d$ , we can now evaluate the given expression.

Next, consider the expression  $a * (x - (y + (a - b) / (a + b)) / (z + b))$ . We first assign the level numbers to parentheses as shown in Fig. 3.3b. This expression contains a pair of parentheses at level 1, which in turn contains two pairs of parentheses at level 2. The first level 2 pair of parentheses contains two pairs of parentheses at level 3 (the innermost level).

The operators in the two innermost pairs of parentheses (level 3) are bound to their operands first followed by the remaining operators in the first pair of parentheses at level 2. The operators in the second pair of parentheses at level 2 are then bound to their operands followed by the remaining operators in the pair of parentheses at level 1. Finally, the remaining operators in the expression are bound to their operands. Fig. 3.3b shows the sequence of operator binding. The given expression is equivalent to the mathematical equation

$$a \left( x - \frac{y + \frac{a-b}{a+b}}{z+b} \right).$$

## Redundant parentheses

The C language provides a large number of operators and it is quite difficult to remember the precedence and associativity rules for all of them. As a result, sometimes the meaning of C expressions may not be very obvious, particularly when the expressions are written using only the required parentheses. If we do not properly remember (and apply) the precedence and associativity rules, we may misinterpret such expressions.

C programmers often use additional pairs of parentheses in expressions to clarify the meaning and improve readability. For example, the expression `a + b * c` can be written more clearly as `a + (b * c)`. As another example, consider the expression `n / 10 + n % 10`, which calculates the sum of the digits of a two-digit integer number `n`. The clarity of this expression can be improved greatly using redundant pairs of parentheses, as `(n / 10) + (n % 10)`. Note that the inclusion of such redundant parentheses does not increase the time required for the evaluation of these expressions.

### 3.2.3 Initializing Variables Using Expressions

In Section 2.7.2, we have looked at a restricted form of variable initialization in which variables are initialized using constant values. Now we make a study of a more general format of variable initialization in which variables can be initialized with values of expressions as shown below.

```
data_type variable] = expr1, variable2 = expr2, ... ;
```

This statement declares variables `variable]`, `variable2`, ... to be of specified `data_type` and initializes them with the values of expressions `expr1`, `expr2`, ..., respectively.

Note that we can also initialize `const` variables with the values of expressions using the format given below.

```
const data_type variable1 = expr1, variable2 = expr2, ... ;
```

#### Example 3.5 Initializing variables with values of expressions

a) Consider the initialization statements given below:

```
int a = 10, b = 20;
int c = a + b / 5;
```

The first statement initializes variables `a` and `b` with values 10 and 20 and the second statement initializes variable `c` with the value of expression `a + b / 5`, i. e., 14. These initializations can be written in a single statement as shown below:

```
int a = 10, b = 20, c = a + b / 5;
```

However, the following initialization statement is incorrect as it uses variables **a** and **b** before they are declared.

```
int c = a + b / 5, a = 10, b = 20;
```

**b)** Now consider the following initialization statements:

```
float x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);  
float x2 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);
```

These statements declare variables **x1** and **x2** of type **float** and initialize them with the values of expressions  $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$  and  $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$ , the roots of the quadratic equation  $ax^2 + bx + c = 0$ . Note that the **sqrt** function calculates the square root of the expression written within a pair of parentheses following it.

### 3.2.4 Constant Expressions

An expression that involves only constant operands is called a constant expression. Thus, a constant expression may contain literal constants as well as symbolic constants. The compiler can evaluate such expressions at compile time.

Several examples of constant expressions are given below. Assume that symbolic constants **MAX** and **PI** are defined with values 100 and 3.1415927, respectively.

10	PI	$2 * PI$	$24 * 60$
'T'	$4.0 * PI / 3$	$MAX + 10$	$(MAX / 12) \% 3$

Note that some of these are mixed-mode expressions as they contain operands of different types. Such expressions are covered in more detail in Section 3.4.1.

### 3.2.5 Avoiding Common Mistakes While Writing C Expressions

The rules of writing expressions in C are quite different from those used in mathematics. As a result, a beginner often makes mistakes while writing C expressions. The common mistakes that we should be aware of and should avoid are discussed below.

#### Incorrect Use of Operators

An expression cannot contain two consecutive operands without any operator between them, i. e., the operands must be connected by operators. Beginners often forget to write the multiplication symbol in expressions. The expressions given below are thus invalid.

$$x \cdot y \quad p \cdot r \cdot n / 100 \quad a \cdot (b + c) \quad (a + b) \cdot (a - b)$$

These expressions should be written correctly as shown below.

$$x * y \quad p * r * n / 100 \quad a * (b + c) \quad (a + b) * (a - b)$$

Beginners also make mistakes while writing expressions containing division operators as shown below.

$$\frac{p * r * n}{100}$$

$$p * r * n / 100$$

The division operator should be written as in  $a / b$ . Thus, a correct way to write above expressions is  $p * r * n / 100$ .

Mathematical equations also contain special operators such as '!' for factorial, ' $\pm$ ' for both addition and subtraction, '.' and 'x' for multiplication, ' $\div$ ' for division, etc. Such operators cannot be used in C expressions. We can only use the operators permitted by the C language.

Another error that we frequently come across is the problem of integer division. For example,  $4/3$  evaluates as 1 and not 1.333333 as expected. To avoid such problems, we should convert at least one of the operands to the floating type using either a constant, as in  $4.0/3$ , or a typecast, as in `(float) sum / n`. The typecasts are covered in Section 3.3.5.

The use of increment and decrement operators along with other arithmetic operators may also lead to errors due to inadequate understanding of the side effects of these operators. Consider the expressions  $x = ++a * b++$  and  $x = a++ * ++b$ . It is best to avoid such expressions.

If the precedence and associativity rules are not clearly understood, the expressions will often be incorrect. When in doubt, we do include redundant parentheses in expressions to avoid such errors.

## Use of Superscripts and Subscripts

Powers are represented in mathematical equations using superscript notation, as in  $a^2$ ,  $b^3$ ,  $c^{10}$ ,  $a^b$ ,  $x^{211}$ , etc. C does not provide exponentiation operator to represent such powers. Nor does it allow the use of superscripts. We can represent small integer powers using the multiplication operation. Thus,  $a^2$  and  $b^3$  can be written as  $a * a$  and  $b * b * b$ , respectively. However, for large integer as well as floating values of power, C language provides the `pow` library function. For example,  $c^{10}$  and  $x^{2.11}$  are written as `pow(c, 10)` and `pow(x, 2.11)`, respectively.

The elements of an array are represented in mathematical equations using subscript notation, e. g.,  $x_1$ ,  $y_{12}$ , etc. However, C does not allow the use of subscript notation. An element of an array is represented by writing each index in a separate square bracket, as in  $x[1]$ ,  $y[1][2]$ , etc. Alternatively, we can use simple variables to represent such subscripted variables, as in  $x1$ ,  $y12$ , etc.

## Use of Special Symbols and Units

Due to restrictions on identifier names in C, Greek symbols used in mathematical equations such as  $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $\delta$ ,  $\lambda$ ,  $\mu$ ,  $\theta$ ,  $\pi$ ,  $\phi$ ,  $\varphi$ , etc. cannot be used as variable or constant names. Instead, we can use their names spelt out as **alpha**, **beta**, **gamma**, **delta**, **lambda**, **mu**, **theta**, **pi**, **phi**, and **psi**, respectively. Also, Roman numerals cannot be used in C expressions. Thus, **VIII** will not be interpreted as value 8 but as an identifier.

C does not provide operators for several mathematical operations such as summation ( $\Sigma$ ), product ( $\Pi$ ), square root ( $\sqrt{}$ ), integration ( $\int$ ), logarithm, absolute ( $| |$ ), floor ( $\lfloor \rfloor$ ), ceil ( $\lceil \rceil$ ), etc. Beginners often use these mathematical symbols in their expressions, particularly the square root symbol. C language, however, provides library functions to evaluate many of these mathematical operations, e. g., **sqrt** (square root), **abs** (absolute value), **log** (logarithm to base e), **log10** (logarithm to base 10), **ceil**, **floor**, etc. The mathematical library functions are discussed in the next chapter. The summation ( $\Sigma$ ) and product ( $\Pi$ ) operations can be performed using loops. C provides three loops, namely, **for**, **while** and **do while**. They were introduced in Chapter 1 and are discussed in detail in Chapter 6.

Another type of error is the use of units for various quantities such as length (cm, m, km), area ( $\text{cm}^2$ ,  $\text{m}^2$ ) volume (ml, l,  $\text{cm}^3$ ,  $\text{m}^3$ ), time (s, ms), temperature ( $^\circ\text{C}$ ,  $^\circ\text{F}$ ), currency (\$, £, Rs., etc.), angle ( $^\circ$ ), etc. Such units should not be used in C expressions.

### Incorrect Use of Parentheses

Yet another common source of errors in C expressions is the incorrect use of parentheses and mismatch in parentheses. After writing C expressions, we can use parenthesis numbering and also convert them to mathematical form to verify their correctness.

We often use different types of brackets in mathematical equations such as parentheses ( ), curly brace { }, square brackets [ ] and angle braces < >. Moreover, we can use them in any desired size. However, we can use only parentheses in C expressions and we can use them in only one size—the same as that of any other character.

## 3.3 Assignment

C provides an assignment operator (=) to assign the value of an expression to a variable. It also provides several compound assignment operators of the form  $op=$ , where  $op$  is an arithmetic (+ - \* / %) or bitwise operator (<< >> & | ^). Thus, the compound assignment operators include

$+=$	$-=$	$*=$	$/=$	$%=$
$<<=$	$>>=$	$&=$	$ =$	$^=$

All assignment operators have equal precedence. As shown in Table 3.3, they have lower precedence than all arithmetic operators and right-to-left associativity. In fact, assignment operators have lower precedence than all other operators except the *comma* operator.

### 3.3.1 Simple Assignment Expressions

A **simple assignment expression** is used to assign the value of an expression to a variable. It takes the following form:

*variable* = *expression*

Note that the assignment expression does not have a terminating semicolon. This assignment expression causes the *expression* on the right-hand side of the assignment operator to be evaluated and its value to be assigned to the *variable*, overwriting the previous value contained in it. Thus, the assignment expression *x* = *a* + *b* \* *c* evaluates the expression *a* + *b* \* *c* and assigns its value to variable *x*.

Consider another interesting assignment expression.

sum = sum + num

Although this expression appears to be mathematically incorrect, it is a valid assignment expression in C. As assignment operators have lower precedence than arithmetic operators, this expression evaluates as *sum* = (*sum* + *num*), i. e., the sum of current values of variables *sum* and *num* is assigned to variable *sum* as its new value.

Several examples of assignment expressions are given below. Assume that PI is a symbolic constant with value 3.1415927.

<i>n</i> = 0	<i>a</i> = <i>a</i> + 1	<i>interest</i> = <i>deposit</i> * <i>rate</i> * <i>year</i> /:
<i>ch</i> = 'A'	<i>num</i> = <i>num</i> / 10	<i>z</i> = ( <i>a</i> + <i>b</i> ) / ( <i>c</i> + <i>d</i> )
<i>x</i> = <i>y</i>	<i>cost</i> = <i>cost</i> *1.5	<i>hundreds</i> = ( <i>num</i> / 100) % 10
<i>c</i> = <i>a</i> + <i>b</i>	<i>n</i> = <i>n</i> * <i>n</i>	<i>area</i> = PI * <i>r</i> * <i>r</i>

### 3.3.2 Simple Assignment Statement

**Assignment statement** is an assignment expression followed by a semicolon, as in

*variable* = *expression* ;

Several assignment statements are given below.

```
a = 0;  
c = a + b;  
hundreds = (num / 100) % 10;  
sum = sum + num;
```

Each assignment statement is usually written on a line by itself as shown above. However, as C is a free-format language, we can write more than one statements on a single line or a single assignment

statement can be written on multiple lines. However, this should be avoided as such statements are difficult to read and may introduce difficult to trace bugs in a program.

### Example 3.6 Using assignment statements to perform simple calculations

#### a) Sum and average of integer numbers

First consider the calculation of the sum and average of three integer numbers. Let us use variables **a**, **b** and **c** to represent three numbers and variables **sum** and **avg** to store their sum and average, respectively. These variables are declared as shown below.

```
int a, b, c;      /* three given numbers */
int sum;          /* their sum and */
float avg;        /* average */
```

As a first attempt, let us write the statements to calculate the sum and average directly as:

```
sum = a + b + c;
avg = (a + b + c) / 3.0;
```

Note the use of parentheses in the second statement and constant 3.0 to avoid integer division. Also note that the sequence of these statements can be altered. However, the code is slightly inefficient as the sum of three numbers is calculated twice. An alternative efficient approach in which we determine the average from the variable sum is given below. However, the sequence of these statements cannot be altered.

```
sum = a + b + c;
avg = sum / 3.0;
```

#### b) Exchange the values of two variables

Let **a** and **b** be the variables whose values are to be exchanged. Both the variables are assumed to be of the same type. To exchange the values of these variables, we might think of a straightforward solution:

```
a = b;
b = a;
```

However, this is wrong! The first statement assigns the value of variable **b** to variable **a** (overwriting its original contents). This new value of variable **a** is assigned to variable **b** in the second statement. Thus, after the execution of these statements, both the variables have same values (that of variable **b**) and the original value in variable **a** is lost.

To avoid this problem, we use a three-step procedure depicted in Fig. 3.4. In this procedure, we first store the value of variable **a** (which was lost in the previous solution) in a temporary variable, say

`temp`, whose type is the same as that of variables `a` and `b`. Now we can safely assign the value of variable `b` to variable `a`. Finally, the value of variable `temp` (i. e., the original value of variable `a`) is assigned to variable `b`. Thus, the values of two variables are exchanged correctly. These steps are given below:

```
temp = a;  
a = b;  
b = temp;
```

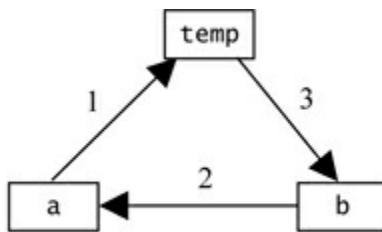


Fig. 3.4

### c) Sum of digits of a three-digit non-negative integer number

To determine the sum of digits of a three-digit non-negative integer number stored in a variable `num` of type `int`, we first separate the individual digits at `unit's`, `ten's` and hundred's place in integer variables `units`, `tens` and `hundreds`, respectively. The digit at unit's position is determined as

```
units = num % 10;
```

and the digits at the ten's and hundred's place are determined as

```
tens = (num / 10) % 10;  
hundreds = (num / 100) % 10;
```

The expression `num / 10` performs integer division to determine the number of tens in the number `num` and the expression `(num / 10) % 10` determines the digit at the ten's place. Similarly, the expression `num / 100` determines the number of hundreds in number `num` and the expression `(num / 100) % 10` determines the digit at the hundred's place. Note that as the multiplicative operators `/` and `%` have left-to-right associativity, the parentheses in the above statements are unnecessary. The sum of digits can now be determined as

```
sum = units + tens + hundreds;
```

The above steps are summarized below.

```
units = num % 10;
```

```

tens = (num / 10) % 10;
hundreds = (num / 100) % 10;
sum = units + tens + hundreds;

```

Observe that the first three steps, used to determine digits in number num, are independent of each other and can be written in any order. However, the sum of digits can be calculated only after all the digits are separated.

It may be noted that if we do not intend to use individual digits for any purpose other than determination of digit sum, we can directly add the values of digits to sum as shown below:

```

sum = num % 10; /* sum equals unit's digit */
sum = sum + (num / 10) % 10; /* add ten's digit to sum */
sum = sum + (num / 100) % 10; /* add hundred's digit to sum */

```

We can even write a single statement to determine the sum of a three-digit number as

```
sum = num % 10 + (num / 10) % 10 + (num / 100) % 10;
```

### 3.3.3 Compound Assignments

The general format of a compound assignment expression is given below.

*variable op= expression*

where *op=* is a compound assignment operator (*+=* *-=* *\*=* */=* *%=*, etc.). This expression is equivalent to the following simple assignment expression

*variable = variable op ( expression )*

Thus, expression *x += 5*, which is equivalent to expression *x = x + 5*, means increase the current value of variable *x* by 5. Similarly, the expression *y /= 5* means divide the value of *y* by 5.

Note the parentheses in the format given above. Likewise, the expression *x \*= a + b / c* is equivalent to *x = x \* (a + b / c)* but not *x = x \* a + b / c*.

Several examples of compound assignment expressions are given below along with the equivalent simple assignment expressions. Observe that the expressions containing compound assignment operator are more natural and easier to read.

<i>x += 10</i>	<i>x = x + 10</i>
<i>sum += num</i>	<i>sum = sum + num</i>
<i>k *= a + b - 10</i>	<i>k = k * (a + b - 10)</i>
<i>x %= num1 + num2</i>	<i>x = x % (num1 + num2)</i>
<i>num /= (a + b) * c</i>	<i>num = num / ((a + b) * c)</i>

A compound assignment statement is a compound assignment expression followed by a semicolon as shown below:

*variable op= expression ;*

### Example 3.7 Sum of digits of a non-negative integer number

We now present another method to determine the sum of digits of a non-negative integer number. This method can be generalized using a loop for any non-negative integer number.

Assume that we wish to determine the sum of digits of a five-digit integer number **num** which has a maximum value of 32767. Turbo C **int** type is sufficient for this. We first separate the least significant digit (the digit at unit's place) from the given number as

```
units = num % 10;
```

and then remove this digit from the given number using integer division as follows:

```
num /= 10;
```

i. e., **num** = **num** / 10. Thus, if the value of **num** is 12345, **units** evaluates as 5 and **num** as 1234. Now the least significant digit in **num**, which is actually the digit at the ten's position in the given number, is separated from **num** and then removed from it as

```
tens = num % 10;
num /= 10;
```

Note that the above steps to separate the two digits are identical except for the variable name for the separated digit. Now the variable **num** contains value 123. We can continue to separate the digits and remove them from **num** until all the digits are separated. The program segment to determine the sum of digits of a five-digit integer number is given below.

```
units = num % 10;
num /= 10;
tens = num % 10;
num /= 10;
hundreds = num % 10;
num /= 10;
thousands = num % 10;
num /= 10;
ten_thou = num % 10;

sum = units + tens + hundreds + thousands + ten_thou;
```

### 3.3.4 Nested Assignments

C permits **nested assignment expressions** (multiple assignments in a single assignment expression) of the form

```
variable1 = variable2 = ... = variableN = expression
```

This assignment expression assigns the value of *expression* to all the variables *variable1*, *variable2*, ..., *variableN*. The value and type of the nested assignment expression is the value and type of the leftmost variable. The nested assignment expression given below assigns value 10 to variables *x*, *y* and *z*.

```
x = y = z = 10
```

Several examples of nested assignment statements are given below.

```
sum1 = sum2 = 0;  
x = y = z = a + b;  
root1 = root2 = - b / ( 2 * a );  
p = q = (a + b) / (c + d);
```

### 3.3.5 Explicit Type Conversion

A typecast operator is a unary prefix operator that can be used to convert the value of an expression to a desired type as shown below:

```
( type ) expression
```

Consider the example given below:

```
avg = (float) (a + b + c) / 3;
```

In this example, we have used the `(float)` typecast to convert the value of expression `a + b + c` to `float` type to avoid integer division.

As the typecast is a unary operator, it has higher precedence than the division operator. Thus, the typecast in the above example converts the value of the parenthesized expression `(a + b + c)` to `float` type before the division operation is performed.

Next, consider the conversion of the given length in meters to feet-inch units. Assume that the variables `feet` and `meters` are of type `int` and `float`, respectively. To determine the value of `feet` we use the equation given below.

```
feet = meters * 100 / 30;
```

In this example, the expression `meters * 100 / 30` converts the length in meters to feet (a

`float` value). The automatic conversion during assignment causes this value to be converted to an `int` type before it is assigned to variable `feet`. However, the compiler may report a warning message indicating loss of data. We can use an explicit typecast to avoid this message as shown below.

```
feet = (int) (meters * 100 / 30);
```

### 3.3.6 Assignments Containing Increment and Decrement Operators

To increment the value of a variable, the increment operator can be used in either the prefix or postfix form as shown below:

```
++x;  
x++;
```

These statements are equivalent to either of the following assignment statements:

```
x = x + 1;  
x += 1;
```

In a similar way, we can use the decrement operator to decrement the value of a variable. The increment and decrement operators can also be used with variables appearing on the right side of the assignment operator. Consider the assignment statement given below:

```
x = a++;
```

As the increment operator has higher precedence than the assignment operator, it is bound to variable `a` before the assignment operator. Thus, the given statement is interpreted as

```
x = (a++);
```

Since the increment operator is used as a postfix operator, the value of variable `a` is used first (i. e., assigned to variable `x`) and then incremented. Thus, the given statement is equivalent to the following statements:

```
x = a;  
a = a + 1;
```

Now consider the statement given below that contains the increment operator in prefix form:

```
y = ++a;
```

In this statement, the value of variable `a` is incremented first and then used (i. e., assigned to variable `y`). Thus, the statement is equivalent to the following statements:

```
a = a + 1;
```

```
y = a;
```

Finally, note that if the value of a variable is modified more than once in an expression, the behaviour is undefined and will depend on the compiler used. A compiler may report a warning error in such situations.

```
a = a++;
b = a++ + ++a;
```

## 3.4 Advanced Concepts

This section discusses advanced concepts related to the topics covered in this chapter, i. e., expressions and assignments. A beginner may skip this section and return to it later after studying more essential concepts presented in the subsequent chapters.

### 3.4.1 Expressions

#### Writing Complex Parenthesized Expressions

We have seen that the precedence and associativity rules along with parenthesis numbering can be used to obtain an equivalent mathematical equation for a given C expression. However, the situation we face more often is that of writing the C expressions for a given mathematical equations. Often we wish to write such expressions using minimum number of parentheses.

If a given mathematical equation is simple, writing the C expression for it is relatively easy. However, it is very difficult, at least for a beginner, to write the correct C expression for a complex mathematical equation. A simple technique to write C expressions for such complex mathematical equations using minimum number of parenthesis is given below.

1. Start with the given mathematical equation and parenthesize the numerators and denominators containing additive operators (addition or subtraction).
2. Parenthesize denominators containing multiplicative operators (multiplication, division and modulo arithmetic). Note that the denominators containing small powers such as  $x^2$  and  $x^3$  should also be parenthesized as they will usually be written using the multiplication operator.
3. Write the C expression by following the parenthesized mathematical equation from left-to-right and top-to-bottom, with the parentheses included at appropriate places.

If you have trouble in identifying the terms to be parenthesized, you can simply parenthesize all additive and multiplicative terms in the given equation. The resulting expression will still be correct but it will contain some redundant pairs of parentheses.

**Example 3.8** Writing parenthesized expressions

**a)** Consider the simple mathematical equations given below:

$$(i) \frac{a+b}{c} \quad (ii) \frac{a}{b+c} \quad (iii) \frac{a+b}{a+c} \quad (iv) \frac{a}{bc} \quad (v) \frac{a}{b/c} \quad (vi) \frac{ab-cd}{a+b+bc}$$

First, rewrite these equations by parenthesizing numerators and denominators containing additive terms and denominators having multiplicative terms as shown below.

$$(i) \frac{(a+b)}{c} \quad (ii) \frac{a}{(b+c)} \quad (iii) \frac{(a+b)}{(a+c)} \quad (iv) \frac{a}{(bc)} \quad (v) \frac{a}{(b/c)} \quad (vi) \frac{(ab-cd)}{(a+b+bc)}$$

Note that the denominator  $bc$  in equation (iv) and  $b/c$  in equation (v) are parenthesized. However, the terms  $ab$ ,  $cd$  and  $bc$  in equation (vi) are not. Now directly write the C expressions for these equations using the left-to-right and top-to-bottom order as shown below.

- (i)  $(a + b) / c$
- (ii)  $a / (b + c)$
- (iii)  $(a + b) / (a + c)$
- (iv)  $a / (b * c)$
- (v)  $a / (b / c)$
- (vi)  $(a * b - c * d) / (a + b * c)$

**b)** Now consider more complex mathematical equations given below.

$$(i) \frac{a+\frac{1}{b+\frac{1}{c+\frac{1}{d}}}}{} \quad (ii) \frac{a^2+b^2}{c+\frac{a-b^2}{a^2}} \quad (iii) \frac{\frac{a+b}{ab}}{\frac{bc}{a^2+b^2}+\frac{bc}{a+\frac{bc}{c+d}}}$$

It will be quite difficult to write C expressions for such complex mathematical equations. However, as you can see, it becomes very easy using the technique explained above. The parenthesized equations are given below. Observe that in the second equation, the denominator term  $a^2$  is parenthesized as it a multiplicative term.

$$(i) a + \left( b + \frac{1}{\left( c + \frac{1}{d} \right)} \right) \quad (ii) \frac{\left( a^2 + b^2 \right)}{c + \frac{\left( a - b^2 \right)}{\left( a^2 \right)}} \quad (iii) \frac{\left( \frac{a+b}{ab} \right)}{\left( \frac{bc}{a^2+b^2} + \frac{bc}{a + \frac{bc}{c+d}} \right)}$$

Now we can easily write C expressions from these parenthesized equations as follows:

- (i)  $a + 1 / (b + 1 / (c + 1 / d))$
- (ii)  $(a * a + b * b) / (c + (a - b * b) / (a * a))$
- (iii)  $(a / (b*c) + (a+b) / (a*b) / (a*a + b*b)) / (a + b*c/(c+d))$

## Simplifying Complex Expressions

Writing C expressions for complex mathematical equations is quite difficult, particularly for beginners. In Section 3.4.1, we learnt how such complex equations can be written easily. Here we present another simple approach in which the given mathematical equation is written in terms of smaller equations contained in it. Consider, for example, the complex equation given below.

$$x = \frac{\frac{a+b}{ab}}{\frac{bc}{a^2+b^2} + \frac{bc}{c+d}}$$

This equation can be simplified as

$$x = \frac{p+q}{r+s} \text{ where } p = \frac{a}{bc}, q = \frac{a+b}{ab}, r = a^2 + b^2 \text{ and } s = \frac{bc}{c+d}.$$

Now it is quite easy to write the C assignment statements for the evaluation of x. However note that the assignments for p, q, r and s should be written before the assignment of x as shown below.

```
p = a / (b * c);
q = (a + b) / (a * b);
r = a * a + b * b;
s = (b * c) / (c + d);
x = (p + q / r) / (a + s);
```

Compare the simplicity and readability of these statements with the single assignment statement written for the evaluation of the original equation:

```
x = (a / (b*c) + (a + b) / (a*b) / (a*a + b*b)) / (a + b*c / ..)
```

### Mixed Mode Expressions

An expression may contain operands of different types. Such expressions are called **mixed mode expressions**. During the evaluation of such expressions, the operators cause conversions of operands to bring them to a common type, which is also the type of the result. These conversions are called **arithmetic conversions**.

The rules for arithmetic conversions are quite complex, particularly when an expression involves **unsigned** operands. To simplify the situation, we consider here the arithmetic conversion rules for operands of four basic data types only (**char**, **int**, **float** and **double**).

For an expression **a op b**,

- 1. If either operand is **double**, the other is converted to **double**.
- 2. Otherwise, if either operand is **float**, the other is converted to **float**.

3. Otherwise, *integral promotions* are performed on both operands, i. e., operands of type `char` (and `short int`) are converted to `int`. Now both the operands are of type `int`.

It can be realized from these rules that if an expression contains at least one operand of type `double`, the result is `double`. Otherwise, if it contains at least one operand of type `float`, the result is of type `float`. Otherwise, the result is of type `int`.

### Example 3.9 Mixed-mode expressions

Assume that the variables `c`, `i`, `f` and `d` are of type `char`, `int`, `float` and `double`, respectively. Consider the following expressions:

<code>d / 1.0E7</code>	<code>d * 2</code>	<code>f - 'A'</code>	<code>c - 'A'</code>
<code>3.142*d</code>	<code>i + 2.0</code>	<code>i * f</code>	<code>32 + c</code>
<code>10 + i % 5</code>	<code>d / i</code>	<code>(1 + f) / f</code>	<code>c % 16</code>

The expressions in the left-most column are not mixed mode expressions as constants `1.0E7` and `3.142` are of type `double`. Thus, no conversions are necessary for their evaluation and the type of the result is the same as that of the operands.

For the expressions in the second column, one of the operands is of type `double`. Thus, during evaluation of these expressions, the other operand will be converted to type `double`, which is also the type of the result.

The expressions in the third column do not have any operand of type `double`. However, they contain one or more operands of type `float`. Thus, the other operand(s) will be converted to type `float`, which is also the type of the result. For example, in expression `f - 'A'`, the value of character '`'A'`' (ASCII 65) is converted to `float`.

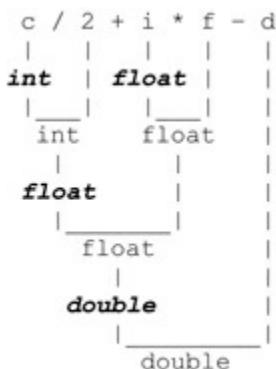
The expressions in the last column do not contain any operand of type `double` or `float`. Thus, integral promotions are performed on the operands of type `char` during the expression evaluation. The result is of type `int`.

### Example 3.10 Evaluation of mixed-mode expressions

Consider the mixed-mode expression `c / 2 + i * f - d`, where the variables `c`, `i`, `f` and `d` are of type `char`, `int`, `float` and `double`, respectively. The evaluation of this expression is illustrated in Fig. 3.5 along with the arithmetic conversions performed during the evaluation. Note that the type of the result is shown in normal font, whereas arithmetic conversions are shown in bold italic.

In this expression, the operators will be bound to their operands in the order `/ * +` and `-`. The division operator operates on operands `c` and `2` of type `char` and `int`, respectively. Thus, integral promotion will be performed on variable `c` before evaluation of the division operator. The result of this operation (integer division) is of type `int`.

The multiplication operator operates on two operands, **i** of type **int** and **f** of type **float**. Hence, operand **i** is first converted to **float** and the result is of type **float**.



**Fig. 3.5**

The next operator to be evaluated is `+`, for which the first operand (value of expression `c / 2`) is of type **int** and other operand (value of expression `i * f`) is of type **float**. Thus, the first operand is converted to **float**, which is also the type of the result.

Finally, subtraction will be performed. This operator has two operands, one of which (`d`) is of type **double**. Hence, the other operand (i. e., value of expression `c / 2 + i * f`) is converted to **double**. The value of the given expression is of type **double**.

## Understanding Tokens

A C program is a sequence of tokens. A **token** is a logical unit (a sequence of characters) that cannot be separated without changing its meaning. C language has six types of tokens: keywords, identifiers, constants, strings, operators and punctuation. An expression can contain all these tokens. Hence, to understand the intricacies of expressions (and the C language), we should understand how tokens are extracted from a C program.

Consider the expression `a+++b`. Note that there are no spaces between the three `+` symbols. Is this a valid expression? Even after reading all about operators and expressions, we get confused when we see such a simple expression (after all we are studying the C language!). Most of us will bet that this is not valid, as C language does not have the `+++` operator. But if I tell you it is, then the question arises what is the meaning of this expression? `a + ++b`? or `a++ + b`? Note that both these expressions (containing spaces) are valid C expressions. Do not get confused here about the right-to-left associativity of the `++` operator and conclude that the given expression (`a+++b`) is equivalent to `a + ++b`.

The answer lies in the process of extracting tokens from a program in which the longest possible tokens are extracted. Thus, after separating variable `a` as a token from this expression, the next longest possible token is `++` (and not `+` or `+++`), which is followed by two more tokens: `+` and `b`. Thus, the

expression `a+++b` is interpreted as `a++ + b`, which is equivalent to the addition of variables `a` and `b` followed by increment of variable `a`.

Now consider the expression `a+++++b`. This expression is invalid as it will be interpreted as `a++ ++ +b` (according to the longest token rule). However, the expression `a+++ ++b` (note the space after the third `+` symbol) is valid as it will be interpreted as `a++ + ++b`.

Now it is easy to understand that the expressions of the form `a++-++b` and `a++*++b` are valid. Finally note that expression `a/+b` is valid as `+` will be interpreted as a unary operator.

### 3.4.2 Assignments

#### Understanding lvalues

The quantity on the left-hand side of an assignment operator must be a *modifiable lvalue*. The term **lvalue** is used to refer to a quantity that can appear on the left-hand side of an assignment operator. A **modifiable lvalue** has memory allocated to it, whose value can be modified, e. g., a non-const variable.

An arithmetic expression containing one or more arithmetic operators has a value but no memory allocated to it. On the other hand, a literal constant has memory allocated to it, but it is not modifiable. Hence, arithmetic expressions and literals are not lvalues and they cannot appear on the left side of an assignment operator. The following expressions are thus invalid:

```
5 = a + b  
a + b = c + d  
++a /= 1  
a-- *= b * c
```

Observe that the increment and decrement operators cannot be used with variables to which a value is being assigned.

#### Type and Value of an Assignment Expression

The type of an **assignment expression** is the same as that of the variable to the left of the assignment operator and its value is equal to the value of that variable after the assignment operation is performed. For example, if variable `x` is of type `double`, the type of the assignment expression `x = 1.25` is `double` and its value is 1.25.

Now consider the nested assignment expression

```
x = y = z = 10
```

As the associativity of assignment operators is right to left, the assignment operators in this expression are assigned to operands from right to left. Thus, the expression evaluates as `x = (y = (z = 10))`.

Initially, the value 10 is assigned to variable `z`. The value of expression `z = 10`, which is equal to the value of `z` is then assigned to variable `y`. Next, the value of expression `y = z = 10`, (which is the value of variable `y`, i. e., 10) is assigned to variable `x`. Thus, all the variables are initialized with value 10. The value of the given nested assignment expression is the value of variable `x`, i. e., 10.

## Conversions in Assignments

When the value of an expression is assigned to a variable of the same type, the value remains unchanged. Otherwise, the assignment operator causes the value of the expression to be converted to the type of the variable. These conversions are summarized below for the basic data types.

1. When a value of type `float` is assigned to a variable of type `double`, the value is unchanged. Likewise, when a value of type `char` is assigned to a variable of type `int`, the value is unchanged.
2. If a floating type value is assigned to an integral variable, the fractional part is discarded. Thus, if variable `x` is of type `int`, the assignment `x = 2.9` actually assigns value 2 to variable `x`. However, the behaviour is undefined if the resulting value cannot be represented in the data type of variable `x`.
3. An integral value may not be exactly representable in a floating type. When an integral value is assigned to a floating type variable, the actual value may be either the next lower or the next higher representable value.

## More on Assignments Containing Increment and Decrement Operators

The increment and decrement operators can be used to form more complex assignment expressions than what we have studied so far. Consider the assignment statement given below:

```
x = a++ * --b;
```

As the `++` and `--` operators have higher precedence than the multiplication operator, they are bound to operands first. Thus, the given expression is equivalent to the statement

```
x = (a++) * (--b);
```

As the `--` operator is used as a prefix operator, variable `b` is decremented before its use. On the other hand, as the `++` operator is used as a postfix operator, variable `a` is used first and then incremented. Thus, the given statement is equivalent to the following statements:

```
b = b - 1;  
x = a * b;  
a = a + 1;
```

However, such complex expressions are difficult to understand and should be avoided as far as possible. Finally note that the increment and decrement operators can only be used with lvalues (such as variables). Thus, the following statements are invalid:

```
++5;  
(a + b)++;  
++a++;
```

The last statement, which is interpreted as,  $b = (++(a++))$ , is invalid as the prefix `++` operator operates on `a++` which is not a lvalue.

### Nested Compound Assignments

C permits the nesting of compound assignment operators of the form

*variable1 op1= variable2 op2= ... op3= variableN op4= expression*

Such assignment expressions are quite difficult to read and should usually be avoided. Note that the simple assignment operator (`=`) can also be used along with the compound assignment operators.

Assume that `a`, `b` and `c` are variables of type `int` having values 1, 5 and 10, respectively. Now consider the nested assignment statement given below.

```
a += b *= c /= 3;
```

As the assignment operators have right-to-left associativity, this expression is evaluated as  $a += (b *= (c /= 3))$ , i. e., the value of variable `c` is first divided by 3 (integer division). Variable `c` now contains value 3. Next, variable `b` is multiplied by the value of expression `c /= 3`, which is the value of variable `c`, i. e., 3. Thus, variable `b` now has a value of 15. Finally, the value of expression `b *= (c /= 3)`, i. e., value of variable `b`, is added to variable `a`. The value of variable `a` now becomes 16. Thus, after the execution of the above expression, the values of variables `a`, `b` and `c` become 16, 15 and 3, respectively.

Two more examples of nested compound assignment expressions are given below.

```
n1 *= n2 -= 10 a += b = c *= e * f
```

### 3.4.3 The Comma Operator

The **comma operator** is used to write multiple expressions where a single expression is allowed. For example, consider the statements given below to exchange values of variables `a` and `b`:

```
temp = a;  
a = b;  
b = temp;
```

These statements can be written in a single statement using the comma operator as

```
temp = a, a = b, b = temp;
```

The comma operator has the lowest precedence of all operators including the assignment operator. Thus, the above statement is evaluated correctly as

```
(temp = a), (a = b), (b = temp);
```

Note that the comma operator has left-to-right associativity and also guarantees the evaluation of contained expressions in the left-to-right order. The comma operator is also useful when we wish to assign different values to several variables and increment/decrement several variables:

```
a = 1, b = 2, c = 3, d = 5;  
i++, j++, k--;
```

The first statement assigns values 1, 2, 3 and 5 to variables **a**, **b**, **c** and **d**, respectively, whereas the second statement increments variables **i** and **j** and decrements variable **k**. Without the use of the comma operator, these statements would normally require seven lines, as we usually write only one statement per line. However, since C is a free-format language, these statements can also be written concisely without the use of comma operator as shown below. However, this is not a good programming practice and should be avoided.

```
a = 1; b = 2; c = 3; d = 5;  
i++; j++; k--;
```

As we have seen above, the comma operator enables us to write concise code. However, such code becomes somewhat difficult to read. Hence, it is usually used only in situations involving closely related operations as illustrated above. A beginner should avoid excessive use of this operator.

Another situation where the comma operator is often used is the initialization and update (increment/decrement) expressions of a **for** loop, enabling us to write certain **for** loops in a concise way. This is discussed in Chapter 6.

### Example 3.11 Comma operator

For further insight into the workings of the comma operator, several statements involving its use are given below.

```
p = a, b, c;
```

In this statement, initially the value of variable **a** is assigned to variable **p** as assignment operator has the higher precedence than the comma operator. Then the expressions **b** and **c** are evaluated, with no effect on the value of variable **p**.

```
q = a + b, b / c, c * d;
```

The arithmetic operators in this statement are first bound to their operands, followed by the assignment operator and the comma operators. Thus, the value of expression **a + b** is first assigned to variable **q**.

The expressions `b / c` and `c * d` are evaluated but they have no effect on any variable.

```
r = a++, b++, c++;
```

In this example, the value of variable `a` is first assigned to variable `r` and then the values of variables `a`, `b` and `c` are incremented.

```
s = (a + b, a - b, a * b, a / c);
```

In this statement, the pair of parentheses containing the comma-separated list of expressions is evaluated first in the left-to-right order. Thus, the value of this pair of parentheses is that of the last expression, i. e., `a / c`. This value is assigned to variable `s`. The evaluation of the first three expressions in a pair of parentheses has no effect on the value of variable `s`.

### 3.4.4 Operations on Strings: Element Access and Assignment

We can access (read or write) individual characters in a string variable (i. e., an array of `char`) using the notation

```
str [ index ]
```

where `str` is a string variable and `index` specifies the array position of the character being accessed. Recall that array elements are numbered from zero. To assign a new value to a character position, we can use the assignment expression/statement. Consider the example given below.

```
char my_str[10] = "Cat";
char ch = my_str[0];
my_str[2] = 'r';           /* modify string as Car */
```

The first statement initializes a string variable `my_str` with string literal "Cat". This string has four characters: 'C', 'a', 't' and '\0'. The second statement assigns `my_str[0]`, i. e., character 'C', to variable `ch` of type `char`. Finally, the last statement assigns character 'r' to `my_str[2]`. Thus, the variable `name` now contains the string "Car". Remember that while assigning a new character to a character position in a string, care must be exercised not to overwrite the null terminator; otherwise, the program may have unpredictable behaviour.

We use an assignment statement to copy the value of an expression to a variable. However, C does not allow assignment of a string (constant or variable) to a string variable, i. e., we cannot copy strings using the assignment operator. Thus, the assignment statements given below are invalid:

```
char msg1[10], msg2[10];
msg1 = "Hello";           /* wrong */
msg2 = msg1;              /* wrong */
```

To assign a string to a string variable, we have to use the `strcpy` (string copy) library function as shown below.

```
char msg1[10], msg2[10];
strcpy(msg1, "Hello"); /* Ok */
strcpy(msg2, msg1);    /* Ok */
```

`strcpy` and other string manipulation functions are discussed in the next chapter.

## Exercises

1. Select the correct option for the following questions:
  - a. Which of the following operators can be used as unary as well as binary operator?
    - i. =
    - ii. -
    - iii. /
    - iv. ++
  - b. Which of the following is not an assignment operator?
    - i. =
    - ii. \*=
    - iii. <=
    - iv. -=
  - c. Which of the following operators cannot be used on variables of type `float` and `double`?
    - i. %
    - ii. +
    - iii. ++
    - iv. \*
  - d. Which of the operator sequences given below indicates the correct precedence for the following operators: `%`, `+`, `*=`, `++`?
    - i. ++ % \*= +
    - ii. ++ + \*= %
    - iii. ++ % + \*=

- iv. None of these
- e. What is the value of expression 'A' + a if a is an integer variable with value 1?
  - i. 66
  - ii. B
  - iii. Can't say
  - iv. Expression has error
- 2. Identify the errors, if any, in the arithmetic expressions given below and rewrite them correctly, if possible. Assume variables a, b, c of type `int` and variables x1, x2, x3 of type `float`.
  - a.  $\pm 10$
  - b. +a
  - c.  $a^{**} b + c$
  - d.  $10 * x1 + x2$
  - e.  $a.b/c$
  - f.  $a \% b + c$
  - g.  $x_1 + x_2 + x_3$
  - h.  $c! * (c - 1)!$ 
    - i.  $a \div b + c$
    - j.  $10 \times a + 15 \times b$
    - k.  $\sqrt{a+b}$
    - l.  $|a^2 - b^2|$
  - m.  $\sum_{i=1}^3 x_i$
  - n.  $\int x \, dx$
  - o.  $x1 \% x2$
- 3. Several simple arithmetic expressions are given below. Identify errors, if any, in these expressions. Assume that the variables are declared as follows:  
`int a, b, c;`  
`float x, y;`  
`char name[20];`
  - a. abc/100
  - b.  $x / y * d$

- c.  $a \% x + c$
  - d.  $5.0 \% 2 * 10$
  - e.  $+a - +b$
  - f.  $-a^* - b$
  - g.  $- -a + -b$
  - h.  $a^* - x / +b / -y$
  - i.  $a + /b * c$
  - j.  $'A' + b$
  - k.  $'+' - '+'$
  - l.  $"Hi" + name$
- f. State which of the following parenthesized arithmetic expressions are invalid. Justify your answers.  
Assume variables  $a$ ,  $b$  and  $c$  of type `int` and variables  $x$ ,  $y$  and  $z$  of type `float`.
- a.  $(\theta)$
  - b.  $(x)$
  - c.  $(a+b) (a+b)$
  - d.  $((x + y))$
  - e.  $<a+b+c> / 3$
  - f.  $a \% [b + c]$
  - g.  $\{a + (-b /c)\}$
  - h.  $('A'+5)%10$
  - i.  $a (b + c)$
  - j.  $-(a + b)$
  - k.  $(a) + ((b)-(c))$
  - l.  $((a)+b)/(b+c)$
  - m.  $(a^2+b^2) / b$
  - n.  $(a/b)/c)/d$
  - o.  $(xyz) / a$
  - p.  $x + (a - b)++$
- i. Which of the following are invalid assignment expressions? Give reasons.

- a. `sum=num+sum`
- b. `a %= 5.0`
- c. `num *= 1`
- d. `a = a`
- e. `a = b = a + b`
- f. `x /= a * b`
- g. `a == a + b`
- h. `a = a + b`
- i. Write C expressions for the following mathematical equations. Also verify, by converting these C expressions back to equivalent mathematical equations, that the C expressions are correct.
- a.  $a + bc$
- b.  $abc/100$
- c.  $1 + x + x^2 + x^3$
- d.  $a^3 - b^3$
- e.  $\frac{a+b}{c}$
- f.  $\frac{a+b}{c}$
- g.  $\frac{a+b}{c+d}$
- h.  $\frac{a^2-b^2}{c-d^2}$
- i.  $1 + \frac{1}{a} + \frac{1}{a^2} + \frac{1}{a^3}$
- j.  $a + \frac{(x+y)^2}{ab}$
- k.  $\frac{x}{(a+b)^2} + \frac{y}{(a-b)^2}$
- l.  $\frac{x^3-y^3}{(xy)+(x^2+y^2)}$
- m.  $\frac{a^2+\frac{1}{b^2+\frac{1}{c^2+d^2}}}{}$
7. Draw the diagrams depicting the operator binding sequence for the C expressions given below and

convert them to equivalent mathematical equations.

- a.  $a * b - c / d$
  - b.  $a + b / c - d * e$
  - c.  $a / (b - c)$
  - d.  $(x + y * z) / (x - y)$
  - e.  $(x + y) * (x + y) / 2 * a$
  - f.  $(x * x + y * y) / (z * z)$
  - g.  $(a + (b-d) / (c+e)) / (x*x*x)$
  - h.  $(a - b * c) / ((d * e - f) * (a + b))$
3. Evaluate the expressions given below. Also state the values of the modified variables, if any. Assume that the variables are initialized as shown below:
- ```
int a = 10, b = 20;  
float p = 1.5, q = -2.5;  
char c = 'A', d = 'a';
```
- a.  $a+b / 5$
  - b.  $p-q / a+b$
  - c.  $a + c - d$
  - d.  $a+b \% 13$
  - e.  $c / a+q$
  - f.  $a / b + b / a$
  - g.  $- p+q$
  - h.  $a\%6 / b\%3$
  - i.  $++c + b$
  - j.  $d + b++$
  - k.  $b \% a++$
  - l.  $-(a * p) - -(b * q)$
  - m.  $(8*a+b)/ b\%3$
  - n.  $++a+b-- / 2.5$
  - o.  $-a+(-p-q) / 4$
  - p.  $a/b + (a/(2*b))$

7. Assume that the variables are declared and initialized as follows:

```
int i = 2, j = 5, x = 10;  
float a = 1.25, b = 5.5, y = 1.0;
```

Determine the values of variables x and y in each of the following C assignment expressions.

- a.  $x = (\text{float}) x - j / 2$
- b.  $y = (y + b) / 5$
- c.  $x = -5 * i + j \% 3$
- d.  $x *= (i+j)/-3 + 3*j/4$
- e.  $x /= (b / 2.75) * (j - 3)$
- f.  $x -= 3 * (i+j) + j / 2*i$
- g.  $y /= x -= (a - b) - j * a$
- h.  $y *= x /= (a + b) / ((i + j) \% 5))$

8. Rewrite the following expressions by removing the redundant parenthesis, if any:

- a.  $(a + (b + c))$
- b.  $(a + b) * c + (c + d) * e$
- c.  $((a) / ((b) - (-c)))$
- d.  $(a - (b / c)) / (d + (e / f))$
- e.  $(a + b) * ((c) / d)$
- f.  $(a + b / c) + (-a) / (x * y)$
- g.  $((((a * a) + (b * b)) / (c * c))$
- h.  $((a \% ((b / c)) * d) + x) / z$

9. Declare the variables and write the assignment statements to perform the following operations. Give suitable comments in your code.

- a. Calculate the area and circumference of a circle
- b. Calculate the volume and surface area of a cuboid
- c. Convert the temperature in Fahrenheit scale to Centigrade and Kelvin scales
- d. Calculate the sum of squares of the differences between each pair of three given numbers
- e. Given three numbers, assign the value of the first number to the second, the second number to the third and the third number to the first
- f. Calculate the sum of squares of the digits of a three digit number.
- g. A number of currency notes of various denominations (100, 50, 20, 10 and 5) are given. Calculate

the total amount.

2. Several program segments are given below along with their purpose on the first line. The required variables are declared first and then the executable statements are written. Identify the errors, if any, in these program segments and rewrite the incorrect program segment correctly.

- a. *Sum of cubes of a two-digit number.*

```
int num = 12;  
int u, t; /* units and tens */  
  
u = num % 10;  
t = num / 10;  
sum = t3+u3;
```

- b. *Volume and surface area of a cylinder.*

```
int radius, volume, surf-area;  
  
r = 10;  
v = 4 / 3 * π * r * r * r;  
sa = 4 * π * r * r;
```

- c. *Reverse a 3-digit number.*

```
int num = 123, rev;  
rev += 100 * num % 10;  
rev += 10 * num % 100;  
rev += num % 1000;
```

- d. *Interchange values of two variables.*

```
int a = 10; b = 20;  
temp = a;  
b = a;  
a = temp;
```

- e. *Convert lowercase letter to uppercase.*

```
char ch = 'e';  
  
ch = ch - 32;
```

- f. *Convert elapsed time in m to hh:mm format.*

```
int min /*      elapsed time      in min */  
int hh, mm /*      converted      time */  
  
min = 275  
hours = min % 60  
minutes = min / 60
```

- g. *Convert length in feet-inch to m-cm.*

```
int feet, inch; /* given length
```

```

*/
int cm; /*given length in>
cm*/
int m, cm; /*converted length */
feet = 3; inch = 2;
cm = (feet * 30 + inch)*2.54;
m = cm / 100;
cm = cm - m * 100;

```

- h. Determine strike rates of a cricket batsman.

```

int runs1, runs2;           /*      runs in 2 1-Days*/
int balls1, balls2; /*      balls faced */
int tot_runs;   /*      total runs */
float strike1, strike2;
float avg_strike; /*      avg strike rate*/

strike1 = runs1 / balls1;
strike2 = runs2 / balls2;
tot_runs = runs1 + runs2;
avg_strike = tot_runs/ balls1 + balls2;

```

## Exercises (Advanced Concepts)

1. Which of the following are invalid assignment expressions? Give reasons.
  - a.  $a /= b *= c$
  - b.  $\theta = sum$
  - c.  $a += a += a$
  - d.  $x + y += 10$
  - e.  $a = b +- c$
  - f.  $a + b = c$
  - g.  $a =+ b + c$
  - h.  $(a * b) += 4$
  - i.  $a = b+c = d$
  - j.  $a = b+(c = d)$
2. Several simple arithmetic expressions involving increment and decrement operators are given below. State which of them are invalid. Justify your answers. Assume variables  $a$ ,  $b$  and  $c$  of type `int` and variables  $x$ ,  $y$  and  $z$  of type `float`.

- a. `++10`
- b. `1.5++`
- c. `'A'++`
- d. `"January"++`
- e. `--abc`
- f. `a+++b`
- g. `a++ b++ c++`
- h. `a * x++ / y`
- i. `num++ / ++k`
- j. `x++ + ++y`
- k. `-a * b - c--`
- l. `a++%--b`
- m. `-x++`
- n. `++-items*cost`
- o. `++-a`
- p. `++a++`

3. Simplify the following mathematical expressions by using intermediate variables for suitable sub-expressions and write the code segment for evaluation of the given mathematical expressions.

$$x = \frac{a^2 + b^2}{c + \frac{a-b^2}{a^2}}$$

a. 
$$x = a^2 + \frac{1}{b^2 + \frac{1}{c^2 + d^2}}$$

b. 
$$z = \frac{\frac{x}{a+b} + \frac{y}{a-b}}{\frac{x}{a-b} - \frac{y}{a+b}}$$

c. 
$$x = \frac{\left(\frac{a+b}{b-c}\right)^2 + \frac{(a+b+c)^2}{(a-b-c)^3}}{\left(a + \frac{b-c}{c+d}\right)^2}$$

d. Write C expressions for the following mathematical equations. Also verify, by converting these C

expressions back to equivalent mathematical equations, that the C expressions are correct.

a.  $a + \frac{(x+y)^2}{ab}$

b.  $\frac{(x+y)^2}{ab} + \frac{(x-y)}{a-b}$

c.  $\frac{x^3 - y^3}{(xy) + (x^2 + y^2)}$

d.  $\frac{a^2 + \frac{1}{b^2}}{c^2 + d^2}$

5. Select the correct option for the following questions:

a. The number of tokens in the parenthesized expression (a+++b++) is

- i. 5
- ii. 6
- iii. 7
- iv. 9

b. If **a** and **b** are variables of type **int**, which of the following is an lvalue?

- i. 0
- ii. a
- iii. a+b
- iv. None of these

c. What is the output of the code segment given below?

```
char s[10] = "ABCDE";
s[6] = 'G';
printf("%s", s);
```

- i. "ABCDEG"
- ii. "ABCDG"
- iii. ABCDE
- iv. ABCDE G

d. What is the output of following code segment?

```
enum Dir {East, West, South, North}
enum Dir d = West;
printf("%d", d++);
```

- i. West
  - ii. South
  - iii. 1
  - iv. 2
- e. What is the output of the following code segment?
- ```
int a = 1, b = 2, c = 3;  
a, b, c = 4;  
printf("%d%d%d", a, b, c);
```
- i. 444
  - ii. 4 4 4
  - iii. 124
  - iv. Compiler error
5. Declare the variables and write assignment statements to perform the following operations. Give suitable comments in your code.
- a. Convert time elapsed in seconds to *hh:mm:ss* format, where *hh*, *mm* and *ss* are hours, minutes and seconds, respectively.
  - b. Determine the maximum number of circular stampings of specified size that can be cut in a rectangular metal sheet. Also, determine the percentage of sheet wasted.
  - c. Determine the area of largest ellipse that can be drawn in a given rectangular area.
  - d. Convert a given length in meters to feet and inches.
  - e. Given the length and width of a rectangular plot of a land in meters, calculate its area in acres.

# 4      The C Standard Library

The C language provides a rich standard library that mainly comprises a large number of functions and macros for performing commonly required operations. This functionality enables C programmers to speed up the program development process. Hence, it is worthwhile understanding the capabilities of the C standard library and study the functionality provided in it. The objective of this chapter is to present the commonly used functionality provided in the C standard library.

We begin with an introduction to the standard library and discuss how library functions can be used in programs. The console input/output facilities are presented next; these include functions for formatted input/output (`scanf` and `printf`), character input/output (`getchar`, `putchar`, `getch` and `getche`) and string input/output (`gets` and `puts`). Then the functionality in the mathematical library is presented, which includes the functions for calculating of powers, logarithms, trigonometric and hyperbolic functions.

The *Advanced Concepts* section first presents the concepts of streams and function prototypes followed by details about format specification in the `printf` function. The functions for character test and classification and string manipulation are discussed, followed by some utility functions. A beginner may skip this section in the first reading.

Several examples and complete programs are provided throughout this chapter to enable the reader to thoroughly understand the concepts presented.

## 4.1 About the C Standard Library

A programmer using a particular programming language expects it to provide a rich set of features and functionality that includes data types, input/output operations, control structures, user-defined functions and/or procedures, arrays, pointers, structures or records, data files, mathematical operations, string and character manipulations, graphics operations and much more. However, providing all these bells and whistles makes a programming language arbitrarily complex. Hence, to keep the language simple, some functionality is usually provided in a library supplied along with the language compiler.

The C library as described in the ANSI standard is referred to as the *C standard library*. Every C compiler that conforms to the ANSI standard must provide all the functionality in the standard library. Thus, using standard library facilities will make programs portable, i. e., they can be compiled and run on almost any computer. However, a compiler developer may provide additional functionality in its library, e. g., graphics functionality in Turbo C/C++. The use of such library facilities may severely limit the portability of programs.

### 4.1.1 Header Files

The C standard library provides the *executable code* and *declarations* for functionality provided in it. The executable code for the library is provided in separate files, usually in the `lib` directory, in the installation directory of the compiler. The library files in Turbo C/C++ are named `*.LIB`, whereas those in the MinGW compiler provided with Dev-C++ and Code::Blocks are named `lib*.a`.

The library can be subdivided into several categories, each containing related functionality. The declarations of functionality in each category are provided in separate text files called **header files**. The important categories (and the corresponding header files) include input/output facilities (`stdio.h`), mathematical operations (`math.h`), string manipulations (`string.h`), character test and classification (`ctype.h`) and utility operations (`stdlib.h`). These categories are summarized in Table 4.1. Note that the header file names have a ".h" extension. These files are usually kept in a directory named "include" in the installation directory of the C compiler.

**Table 4.1** Important categories in the C standard library

Category	Header file	Example functions
Input/output facility	<code>stdio.h</code>	<code>scanf, printf, getchar, putchar, gets, puts</code>
Mathematical operations	<code>math.h</code>	<code>sqrt, sin, cos, log, pow</code>
String manipulations	<code>string.h</code>	<code>strcpy, strcat, strlen, strcmp</code>
Character test and classification	<code>ctype.h</code>	<code>isalpha, isupper, toupper</code>
Utility operations	<code>stdlib.h</code>	<code>abs, bsearch, qsort, exit, rand</code>

When we use some functionality from the standard library, e. g., a standard library function, the corresponding header file should be included in the program using the `#include` preprocessor directive, as in

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

Observe that the standard header file names are enclosed in angle brackets, `< and >`. Usually header files are included at the beginning of a program (in any order). When we compile the program, another program called *C preprocessor* prepares an intermediate program file by replacing each `#include` directive with the contents of the specified header file. Of course, the original C program file is not affected as a result by this replacement. The intermediate program file is then translated by the *C compiler* to generate an object code (i. e., machine code). This is then processed by another program called the *linker* which links the standard library function calls in the program with its executable code provided in the library and generates an executable file. This file can then be executed to perform the operations specified in the C program.

## 4.1.2 Functions

A **function** is a subprogram that is used to perform a predefined operation and optionally return a value. Using functions, we can avoid repetitive coding in programs and simplify as well as speed up program development.

The C language provide two types of functions: **library functions** and **user-defined functions**. Library functions relieve a programmer from writing code for commonly used functionality. If the desired functionality is not available in a library, programmer can define new function(s) to perform the desired operations. Such functions are called *user-defined functions*. They are discussed in Chapter 8.

Once a function is defined (either in the standard library or by the user) it can be *called* (i. e., used) to perform its intended operation. During a function call, a function may accept one or more inputs as **arguments**, process them and return the result to the calling program. For example, in function call `sqrt(x)`, the `sqrt` function accepts the value of `x` as input and returns its square root. Similarly, function call `log(x + y)` returns the natural logarithm of the argument `x+y`, and function call `pow(x, y)` returns the value of  $x^y$ . Note that the function arguments are enclosed within parentheses and are separated by commas.

### Function Call

A **function** call takes the following form:

*func\_name ( arg1, arg2, ... )*

where *func\_name* is the name of the function and *arg1, arg2, ...* are argument expressions. Note that the arguments are separated by commas and are enclosed within parentheses. The arguments usually provide the input to a function. However, we can use argument variables to obtain output from a function as well.

The number of arguments and their types in a function call must correspond to the parameters specified in the declaration or prototype<sup>†</sup> of that function (discussed in Section 4.4.2). The sequence of arguments in a function call must also be correct. For example, to calculate  $a^b$ , we call the standard library function `pow` as `pow(a, b)` but not as `pow(b, a)`.

When a function is called, the argument expressions are evaluated and assigned to the function parameters. If the type of an argument is different from that of the corresponding function parameter, the argument value is converted, if possible, to that type. Otherwise, the compiler reports a *type mismatch* error.

### Example 4.1 Function calls

The `sqrt`, `log` and `pow` functions calculate and return the square root, natural logarithm and power ( $x^y$ ), respectively, of the argument expression(s) specified in the function call. Their prototypes

provided in the `math.h` header file are given below.

```
double sqrt(double);
double log(double);
double pow(double, double);
```

Observe that the `sqrt` and `log` functions accept one argument of type `double`, whereas the `pow` function accepts two arguments of type `double` (values of  $x$  and  $y$ ) and that each of these functions returns a `double` value.

Now assume that the variables `i`, `f` and `d` are of type `int`, `float` and `double`, respectively. First, consider the following function calls:

```
sqrt(d)  log(10.0)      pow(d, 5.0)
```

Recall that the constants `10.0` and `5.0` are of type `double`. The number of arguments and their types in each function call are exactly as specified in their prototypes. Thus, these function calls are valid except that the values returned by them are not used. The function calls given below are also valid as the expressions `f * 1.25` and `d + 2` evaluate as `double` values.

```
log(f * 1.25)  pow(d + 2, 3.0)
```

Next consider the function calls

```
sqrt(i)      pow(5 * f, 2)
```

These calls have the correct number of arguments but are mismatched in the parameter and argument types. However, note that the expressions `i` and `2`, which are of type `int`, can be converted to type `double` and the expression `5 * f`, which evaluates as `float`, can also be converted to `double`. Thus, these function calls are also valid.

Finally, consider the following function calls which are invalid for the reasons mentioned.

```
sqrt()          /* required argument missing */
sqrt(d, 2.0)    /* more arguments than required */
pow(d + 5.0)    /* fewer arguments than required */
log("Hello")    /* argument type mismatch */
```

The call to the `log` function is invalid because a string constant cannot be converted to type `double`. The compiler will report a *type mismatch* error.

## Expressions Containing Function Calls

If a function returns a value, we can assign it to a variable of appropriate type. We can also call such a function from within an expression. An expression may contain one or more function calls. C also allows function calls to be nested, i. e., one function call to be written inside another.

During the evaluation of an expression containing function calls, the calls are considered as single operands. Hence, they should be evaluated before the evaluation of adjacent operators. If an expression contains nested function calls, the inner function calls are evaluated before the outer ones.

### Example 4.2 Expressions containing function calls

Assume that the variables `x`, `y` and `z` are of type `double`. First, consider the assignment statement in which the value returned by the `sqrt` function is assigned to variable `z`:

```
z = sqrt(x);
```

Next consider the expression containing a call to the `sqrt` function: `x + sqrt(2 * y)`. Note that the function call `sqrt(2 * y)` is correct and the value returned by the `sqrt` function, which is of type `double`, is used as an operand for the addition operator. Thus, the given expression is valid and is equivalent to the mathematical expression  $x + \sqrt{2y}$ .

Now consider another expression that contains more than one function calls:

```
(sin(x + y) - sin(x - y)) / sqrt(x * y)
```

In this expression, the argument expressions for the `sin` and `sqrt` functions are evaluated before the functions are called. Thus, expression `x * y` will be evaluated before the `sqrt` function is called. Observe that the calls to the `sin` and `sqrt` functions are correct and the values returned by these calls, which are of type `double`, are correctly used as operands for arithmetic operators. The given expression is equivalent to the mathematical expression

$$\frac{\sin(x+y)-\sin(x-y)}{\sqrt{xy}}.$$

Next consider the following expression containing nested function calls:

```
sin(x * y) + pow(abs(x), y)
```

The `abs` function from the standard mathematical library determines the absolute value of the given argument expression. In this expression, function call `abs(x)` is the first argument for the `pow` function, i. e., call to the `abs` function is nested within the call to the `pow` function. Thus, the function call `abs(x)` should be evaluated before the `pow` function. The given expression is equivalent to the mathematical expression  $\sin(xy) + |x|^y$ .

Several expressions containing nested function calls are given below along with the equivalent mathematical expressions.

1. `log(pow(y, 3) * sqrt(abs(x)))`

$$\begin{aligned}
 & \text{1. } \sqrt{\log_e(y^3 \sqrt{|x|})} \\
 & \text{2. } \frac{\sin\left(\log_e xy - \log_e \frac{x}{y}\right)}{(x+y)^5} \\
 & \text{3. } \frac{(x^{a+b} + y^{a-b})}{(x+y)^{1.0/3}} \frac{\sqrt[3]{x+y}}{\sqrt[3]{x+y}}
 \end{aligned}$$

### Function Call as C Statement

If a function does not return a value (or if we are not interested in the value returned by it), a function call takes the form of a C statement in which the function call is followed by a semicolon as shown below.

```
printf("Enter values of a and b: ");
scanf("%d %d", &a, &b);
printf("The values are a = %d b = %d", a, b);
```

In these statements, the values returned by the `scanf` and `printf` functions (discussed shortly) are ignored.

## 4.2 Input and Output Facilities

C does not provide language constructs for input and output (I/O) operations. Instead, it provides this functionality in the standard library. The commonly used console (keyboard and display) I/O functionality includes the functions and macros for formatted I/O (e. g., `scanf` and `printf` functions), character I/O (e. g., `getchar` and `putchar` macros) and string I/O (e. g., `gets` and `puts` functions). Each of these functions returns a value. However, except for the `getchar` function, we usually ignore the values returned by these functions.

The `scanf` and `printf` functions perform formatted I/O operations. The `scanf` function is used to read the values for one or more variables from the keyboard. The `printf` function, on the other hand, is used to write the values of one or more expressions on the display.

The `getchar` and `putchar` macros perform character I/O. The `getchar` macro reads a character from the keyboard, whereas the `putchar` macro writes a character on the display. The `gets` and `puts` functions are used for string I/O. The `gets` function reads a string from the keyboard and the `puts` function writes a string on the display.

The declarations for input/output functions are provided in the standard input/output header file, `stdio.h`. As every program requires some input or output, this header file is included in almost every C program, as shown below.

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

#### 4.2.1 Formatted Output – the `printf` Function

The `printf` (print formatted) standard library function is used to print the values of expressions on standard output (i. e., display) in a specified format. A typical call to the `printf` function takes the form of a C statement as

```
printf(format_string, expr1, expr2, ...);
```

where *expr1*, *expr2*, ... are expressions whose values are to be printed and **format\_string** specifies how they should be printed. The ellipsis (...) indicates that the `printf` function call may contain variable number of expressions. These expressions are optional and may be omitted.

The format string may contain **conversion specifications** that begin with the % character followed by ordinary characters. The ordinary characters are simply copied to the output stream, i. e., they are printed on the display screen. However, each conversion specification causes the conversion and printing of the next argument expression (*expr1*, *expr2*, ...) in the format specified by it. For example, the `%d` conversion specifier prints the next argument expression as an integer, the `%f` conversion specifier prints it as a floating number (six digits after the decimal point), whereas the `%10.2f` conversion specifier prints it as a floating number using a minimum field-width of 10 characters and two digits after the decimal point.

The conversion specifications in the format string of the `printf` function call should match the argument expressions to be printed with regard to the number as well as type. This is the programmer's responsibility. C compilers usually do not report any errors for mismatch in conversion specifications and arguments. If the conversion specifications are more in number than the argument expressions, the result will be unpredictable. However, excess argument expressions are simply ignored.

The `printf` function returns an integer value indicating the number of bytes printed. This value is usually ignored in the `printf` function call.

##### Example 4.3 Using the `printf` function

Consider the `printf` statement given below.

```
printf("C is a very easy programming language\n");
```

The format string in this call does not contain any conversion specifications. Thus, the given string (C is ... language) is printed followed by a new line (\n) causing the cursor to move to the next line as shown below.

```
C is a very easy programming language
```

Now consider another `printf` statement given below.

```
printf("a = %f b = %f ratio = %8.3f", a, b, a/b);
```

Observe that the format string contains three conversion specifications (%f, %f and %8.3f) corresponding to three values to be printed (a, b and a/b). All other characters are ordinary characters that are simply copied to the display. Thus, if the values of variables a and b are 2.0 and 3.0, respectively, the printf statement displays the output as shown below.

```
a = 2.000000      b = 3.000000      ratio = 0.667
```

Observe that the values of a, b and a/b have appeared in place of the corresponding conversion specifiers. The values of a and b are printed using six digits after the decimal places, which is the default for values of float type. The value of expression a/b is printed in a field of eight characters and three digits after the decimal point, as specified in the format 8.3f.

### Printing Strings

The ordinary (i. e., non-formatting) characters in the format string of a printf function call may include graphic as well as non-graphic characters. The graphic characters (letters, digits and symbols such as +, \*, /, =, &, \$, etc.) in the format string are printed on the display exactly as they appear in the format string. On the other hand, when a non-graphic character is printed, the corresponding effect is obtained.

The most common non-graphic characters included in the format string of a printf function call are newline (\n) and horizontal tab (\t). Newline (\n) causes the cursor to move to the beginning of next line. If the cursor is already on the last line, the display contents are scrolled up and the cursor is moved to the next line. The horizontal tab (\t) causes the cursor to move to the next tab position, which occurs at every eight character positions on each line (i. e., at character positions 1, 9, 17, 25, ...). If the cursor is beyond the last tab position on a line, it usually moves to the beginning of the next line. When the alert character (\a) is printed, the computer responds with a short beeping sound.

#### Example 4.4 Printing strings using the printf function

##### a) Printing a single string

```
printf("Hello, world");
```

This statement prints the string "Hello, world" (without quotes). The cursor remains on same line immediately after the last character printed (d in this case). The next printf statement in the program, if any, will begin its output at this position.

##### b) Printing strings using multiple printf statements

Consider the printf statements given below:

```
printf("Monday");
printf("Tuesday");
printf("Wednesday");
```

These statements produce the following output:

MondayTuesdayWednesday

Note that there are no spaces between these names. This is because the output of a `printf` statement appears immediately after the output of previous `printf` statement. If spaces are desired between the strings, they should be included in the format strings as shown below:

```
printf("Monday ");
printf("Tuesday ");
printf("Wednesday");
```

These statements produce the following output:

Monday Tuesday Wednesday

Of course, we can obtain the same output using a single `printf` statement, given below:

```
printf("Monday Tuesday Wednesday");
```

#### c) Using escape sequences in format strings

```
printf("Hello Friends\n");
printf("C is a very easy programming language\aa");
```

The first `printf` statement prints the string `Hello Friends` followed by a newline (`\n`). The newline causes the cursor to move to the next line where the output of the next `printf` statement will begin. Note that the alert or audible bell (`\a`) causes a short beeping sound to be made after the string in second `printf` statement is printed. The output is given below.

Hello Friends  
C is a very easy programming language (beep)

#### d) Using tabs to separate output

```
printf("One\tTwo\tThree\tFour\tFive\tSix");
```

Each horizontal tab character (`\t`) in this statement causes the cursor to move to the next tab stop which occurs at every eight characters. The output of this statement is as follows:

One        Two        Three     Four      Five      Six

We can include multiple tab characters to increase the spacing between the items printed. Moreover, we

can also include newline characters to print the output on multiple lines, as follows:

```
printf("One\t\tTwo\t\tThree\nFour\t\tFive\t\tSix\n");
```

The output of this statement is as follows:

```
One      Two      Three  
Four     Five     Six
```

### Program 4.1 Print a mailing label

Write a program to print a mailing label.

**Solution:** Let us assume that a mailing label is to be printed as shown below. Feel free to change the contents and include your personal details.

```
Dr. R. S. Bichkar  
Professor (Dept. of E&TC)  
G. H. Raisoni College of Engineering and Management  
Wagholi, Pune  
MS 412207 INDIA
```

The program to print this mailing label is given below. It starts with a comment that states the purpose of the program. The second line includes the ...<stdio.h> header file (as we will use the `printf` function to print the output). This is followed by the `main` function, which contains `printf` statements to print the mailing label. Note that one `printf` statement is used to print each output line.

```
/* Print a mailing label */  
#include <stdio.h>  
int main()  
{  
    printf("Dr. R. S. Bichkar\n");  
    printf("Professor (Dept. of E&TC)\n");  
    printf("G. H. Raisoni College of Engineering and Management\n");  
    printf("Wagholi, Pune\n");  
    printf("MS 412207 INDIA\n");  
    return 0;  
}
```

## Using Conversion Specifications

To print the values of variables and expressions using a `printf` statement, we use **conversion specifications** in the *format\_string* of the `printf` function call. In its simplest form, a conversion specification consists of a *percent* character (%) followed by a **conversion character**, as in `%c`, `%d`, `%f`, etc. The commonly used conversion characters are given in Table 4.2 along with the type of

argument expected and conversion performed.

**Table 4.2** The commonly used conversion characters for the `printf` function

Conversion character	Argument type	Argument converted to
d, i	int	Signed decimal notation of the form [-]ddd
f	double	Decimal notation of the form [-]ddd.ddddddd. The default precision (i. e., digits after decimal point) is six.
c	int	Single character, after conversion to <code>unsigned char</code> .
s	char *	The characters from given string are printed until a '\0' is encountered.

The %d (decimal) and %i (integer) conversion specifications expect an argument of type `int` which is printed in signed decimal notation as [-]ddd, where d represents a decimal digit. The numbers are printed using minimum digits and the sign is printed only for negative numbers. Note that these conversion specifications can also be used to print arguments of type `char`.

The %f (floating-point) conversion specification expects an argument of type `double` and prints it in decimal form as [-]ddd.ddddddd, where d represents a decimal digit. The sign is printed only for negative numbers. Also, the numbers are printed using minimum digits before the decimal point but six digits after it. Note that the %f conversion specification can also be used to print arguments of type `float`.

The %C (character) conversion specification expects an argument of type `int` and prints it as a character. We can also use %C for arguments of type `char` as they have integral values.

The conversion specification %S (string) expects an argument of type *pointer to char* (i. e., `char *`). Thus, we can use either a string constant or a character array as an argument. The characters in the string or character array are printed until a null character (\0) is encountered. Pointers are discussed in Chapter 11.

The format string in the `printf` function call may contain more than one conversion specifications. The first conversion specification causes the conversion and printing of the first argument expression, i. e., `expr1` in the format of the `printf` statement. Each subsequent conversion specification causes the conversion and printing of the next argument expression.

The conversion specifications and the expressions to be printed must match in number as well as type. Thus, for every expression to be printed, there must be an appropriate conversion specification in the format string. We can use ordinary characters (both graphic and non-graphic) in the format string to improve the readability of the output. The ordinary characters included in the format string are printed on the screen along with the values of arguments in the specified order.

#### Example 4.5 Using conversion specifications in the `printf` statement

### a) Printing values of different types

Consider the `printf` statements given below:

```
printf("%d %d %d\n", 10, -100, 'A');
printf("%f %f %f %f\n", 1.23, 1.23454321, -1.23456789, 1.23f);
printf("%c %c\n", 65, 'B');
printf("%s %s\n", "Hello,", "world");
```

Observe that a separate format specification is used for each value being printed. Thus, the number of format specifications in each format string equals the number of values being printed in that `printf` statement. The use of the newline character (`\n`) at the end of each format string causes the output of each `printf` statement to be printed on a separate line. Also, the spaces between the format specifications enable us to avoid mixing of output values. The output is given below.

```
10 -100 65
1.230000 1.234543 -1.234568 1.230000
A B
Hello, world
```

The first `printf` statement prints values of type `int` and `char` using the `%d` format specifications. Note that the positive numbers are printed without any sign or leading space and character constant '`'A'`' is printed as 65, its ASCII code.

The second `printf` statement prints the values of type `double` (first three values) and `float` (`1.23f`). Observe that each value is printed using six digits after the decimal point and that the numbers are rounded to the sixth digit. Thus, values 1.23, 1.23454321 and 1.23456789 are printed as 1.230000, 1.234543 and 1.234568, respectively.

Third `printf` statement uses the `%c` format specification to print values of type `int` and `char`. Note that character constants are printed without single quotes and integer value 65 is printed as A (character representing ASCII value 65).

The last `printf` statement prints the strings "`Hello,`" and "`world`" using the `%s` format specification. Observe that the quotes surrounding the strings are not printed in the output.

### b) Printing expressions

In the previous example, the values to be printed are literal constants. The `printf` function can also be used to print values of variables and expressions. Consider the example given below.

```
char c = 'A';
int i = 10;
double d = 1.25;
char str1[] = "Hello", str2[] = "World";
```

```
printf("%c %c\n", c + 1, c + i);
printf("%d %d %d\n", i, -2 * i, c + 1);
printf("%f %f %f\n", d, d * d, d + i);
printf(%s, %s!\n", str1, str2);
```

The argument expressions in the first and second `printf` statements are of type `int`, whereas those in the third and fourth statements are of type `double` and `string`, respectively. Note that the format specifications have been correctly used in these statements with regard to number and type. The output printed by these `printf` statements is given below.

```
B K
10 -20 66
1.250000 1.562500 11.250000
Hello, world!
```

#### c) Mixing conversion specifications in the format string (Printing values of different types)

A `printf` statement may be used to print the values of expressions of different types, i. e., we can mix various conversion specifications in a format string of the `printf` statement. Consider the `printf` statement given below:

```
printf("%d %f %s %c", 100, 10.5, "January", '*');
```

In this statement, the format string contains four conversion specifications (`%d`, `%f`, `%s` and `%c`) used to print constants `100`, `10.5`, `"January"` and `'*'`, respectively. Note that the first conversion specification (`%d`) is correct for the first argument (`100` of type `int`). Similarly, subsequent conversion specifications (`%f`, `%s` and `%c`) are also correct for the next three arguments (`10.5`, `"January"` and `'*'`, respectively). The output of this statement is given below.

```
100 10.500000 January *
```

#### d) Making the output more meaningful (Printing text along with values)

Consider the code given below.

```
int a = 10, b = 20;
printf("%d %d %d %d", a, b, a + b, a * b);
```

This `printf` statement displays the output as shown below:

```
10 20 30 200
```

We can make this output more meaningful by including text specifications as illustrated below, along with conversion

```
int a = 10, b = 20;
printf("Given numbers are a = %d and b = %d\n", a, b);
printf("Their sum is %d and product is %d\n", a + b, a * b);
```

These statements produce the following output:

```
Given numbers are a = 10 and b = 20
Their sum is 30 and product is 200
```

#### e) Mismatch in format specifications and argument expressions

As we already know, proper care must be taken while writing the `printf` statements as mismatch in format specification and argument expressions regarding number and type may cause unpredictable results as illustrated below.

```
int a = 10, b = 20;
float x = 1.2;
printf("%d %d\n", a, b, a + b);
printf("%d %d %d\n", a, b);
printf("%d %f\n", x, a);
```

Observe that the format specifications and the argument expressions in these `printf` statements do not match properly. The first `printf` statement has an extra argument expression, the second has an extra format specification, whereas the third one has a type mismatch. Note that the compiler does not report any error or warning. When the code is executed using Turbo C, the following output is obtained.

```
10 20
10 20 2396
0 0.000000
```

whereas the output obtained using Dev-C++ is as follows:

```
10 20
10 20 30
1073741824 0.000000
```

Observe that extra argument in first `printf` statement is ignored in both cases. However, the extra format specification in second `printf` statement and the mismatch in format and argument types in third `printf` statement causes incorrect output to be displayed in both cases.

#### Program 4.2 Area and circumference of a circle with radius 10

Write a program to calculate the area and circumference of a circle with radius 10.

**Solution:** For a circle with radius  $r$ , the area and circumference are given as  $\pi r^2$  and  $2\pi r$ , respectively. Let

us use variables `radius`, `area` and `circum` of type `float` and a symbolic constant `PI` to represent the constant  $\pi$ . The program given below first includes the `stdio.h` header file and defines the symbolic constant `PI`.

The `main` function initially declares the required variables and assigns value 10.0 to variable `radius`. It then calculates the values of `area` and `circum` and prints them using the "%f" format specification.

```
/* calculate area and circumference of circle */
#include <stdio.h>
#define PI 3.1415927
int main()
{
    float radius, area, circum;
    radius = 10.0;
    area = PI * radius * radius;
    circum = 2.0 * PI * radius;
    printf("Radius of circle = %f\n\n", radius);
    printf("Area = %f\n", area);
    printf("Circumference = %f\n", circum);
    return 0;
}
```

The program output is given below.

```
Radius of circle = 10.000000
Area = 314.159271
Circumference = 62.831656
```

This program has been written for a specific value of radius. If we wish to calculate the area and circumference of a circle having a different radius, we need to edit the program and recompile it before execution. This inconvenience can be avoided by using the `scanf` function to accept the value of `radius` from the keyboard. The `scanf` function is discussed below.

#### 4.2.2 Formatted Input – the `scanf` Function

The `scanf` standard library function is used to read one or more values from the standard input (keyboard) and assign them to specified variables. A typical call to the `scanf` function takes the following form:

```
scanf( format_string, &var1, &var2, ... );
```

where *var1*, *var2*, ... are the names of variables to which values are to be assigned. Note the *address of* (&) operator before each variable name<sup>1</sup>. Thus, we actually specify the memory addresses of variables into which the values are to be read. A beginner often forgets to write the address-of operator.

The *format\_string* is similar to that in the `printf` function. It specifies the format in which the values should be entered from the keyboard. It usually contains conversion specifications (such as %d, %f, %s, %C, etc.) that determine the interpretation of input values.

For each value being read from the keyboard, the `scanf` function reads the required number of characters from the input field, converts this value to the internal representation of the corresponding argument variable and stores the converted value in that variable. On success, the `scanf` function returns the number of fields successfully scanned, converted and stored. This value is usually ignored in a `scanf` function call.

Besides the conversion specifications, the format string may contain whitespace characters (blanks, tabs or newline characters) which cause the whitespace characters in the input to be ignored until a non-whitespace character is encountered. The format string may also contain non-whitespace characters other than % which cause matching characters to be read (but not stored) from the input.

## Using Conversion Specifications

A conversion specification in a `scanf` function call determines how the next input field will be interpreted. In its simplest form, a conversion specification consists of the character % followed by a *conversion character*. The commonly used conversion characters are given in Table 4.3 along with the type of argument and the input data expected.

**Table 4.3** The commonly used conversion characters for the `scanf` function

Character	Argument type	Input data
d	int *	Decimal integer
f	float *	Floating point number (in decimal or scientific notation)
c	char *	Character(s)
s	char *	String of non-whitespace characters (i. e., a sequence of characters until a newline, tab or space is encountered)

The %d and %f conversion specifications are used to read values for integer and floatingpoint variables, respectively. The %s specification is used to read a string of non-whitespace characters. Whereas, the %C specification is used to read one or more characters that may include whitespace characters. Note that these conversion characters are similar to those used in the `printf` function.

When using the %d conversion specification, the input should be in the format [ ±]ddd, i. e., a sequence of digits optionally preceded by a '+' or '-' sign. Similarly, when using the %f conversion specification, the input should be a floating-point number either in decimal or scientific notation. More

specifically, the input should be in the format  $[\pm]dd[.]dddd[E|e[\pm]dd]$  (without any spaces in between), where the fields in square brackets are optional. Note that an integer value is also accepted for the % specification. The reading of input stops when a whitespace or any character other than those in the specified format is encountered. The unread characters in the input stream are used for subsequent conversion specifications or `scanf` statements.

#### Example 4.6 Reading input from the keyboard

##### a) Reading numeric data

Consider the statements given below:

```
int i;  
float x;  
scanf("%d%f", &i, &x);
```

Here, the `scanf` statement reads values for two variables, `i` of type `int` and `x` of type `float`, using the `%d` and `%f` conversion specifications, respectively. The numbers entered from the keyboard should be separated by whitespace i. e. spaces, tabs or newlines:

100 12.5

Note that the data entered from the keyboard is shown with an underline. Also note that as the whitespace in the format string causes the whitespace in the input to be ignored, the above `scanf` statement may be written in a more readable form as shown below.

```
scanf("%d %f", &i, &x);
```

##### b) Reading character data

Consider another example given below.

```
char a, b, c;  
scanf("%c%c%c", &a, &b, &c);
```

The `scanf` statement reads the values for three variables `a`, `b` and `c` of type `char`. If we wish to assign the characters 'B', 'A' and 'T' to these variables, these characters should be entered from the keyboard, without any whitespace before and between these characters, as follows:

BAT

However, if we enter these characters separated by a single space character as shown below

B A T

the values are assigned to variables as `a = 'B'`, `b = ' '` and `c = 'A'`. This is because the `%c` conversion specifier does not ignore whitespace in the input stream while reading values from it.

Note that if we wish the `scanf` statement to correctly read the input containing whitespace (before and between characters), the format string should be modified to include the spaces as shown below:

```
scanf(" %c %c %c", &a, &b, &c);
```

#### c) Mixing numeric and character input

Consider the `scanf` statement given below used to read two integer numbers separated by an operator (such as +, -, \*, /, etc.).

```
int a, b;
char op;
scanf("%d%c%d", &a, &op, &b);
```

Now assume that the data is entered as shown below:

10+20

The input for variable `a` stops when character '+' is encountered. Thus, value 10 is assigned to variable `a`. Note that the character '+', which has not yet been read, is assigned to variable `op` of type `char`. Subsequently, value 20 is assigned to variable `b`.

As the user may include spaces while entering the data for this `scanf` statement, we should include at least one space before the `%C` specification in the format string as shown below.

```
scanf("%d %c%d", &a, &op, &b);
```

#### d) Reading character strings

Consider the code segment given below.

```
char name[20];
printf("Enter your name: ");
scanf("%s", name);
printf("\nHi %s! Welcome to the wonderful world of C.", name);
```

Here, `name` is an array of type `char` that is used to store the name entered from the keyboard. Please refer to Section 2.9.4 for more information on character arrays or strings. As one character will be required for the null terminator, the name entered can have at most 19 characters in it. Observe that the *address-of* operator is not required before array `name` as the array `name` itself is a pointer to the first element, i. e., `a[0]`. A `scanf` statement is used to read the user's name from the keyboard. This name is then printed using the `printf` statement along with a message. The output is shown below.

Enter your name: Rajan

Hi Rajan! Welcome to the wonderful world of C.

### e) Reading input in specific format

Consider that we have to read a date in the dd/mm/yyyy format. For this, we can include '/' characters in the format string as shown below.

```
int dd, mm, yy;  
scanf("%d/%d/%d", &dd, &mm, &yy);
```

This `scanf` statement expects the user to enter three integer values separated by the '/' character, which is the desired format for the date. Another example where this feature is useful is entering the value of a complex number. The `scanf` statement given below reads a complex number in the format  $re + i im$  where  $re$  and  $im$  are the real and imaginary parts, respectively.

```
float re, im;  
scanf("%f +i %f", &re, &im);
```

Note that the space before `+i` is required to accept the data containing whitespace.

### Prompting Users for Data Entry

When a `scanf` statement is executed, the computer waits for user input. However, the number of input values, their types and the sequence in which these values are expected is not known to the user/operator. Thus, the user is likely to make mistakes while entering the data. To avoid such mistakes, it is a good programming practice to display a message prompt before reading the input as illustrated below.

```
printf("Enter deposit, rate and years: ");  
scanf("%f %f %d", &deposit, &rate, &years);
```

When program containing these statements is executed, a message requesting data from the user is displayed before accepting the data as shown below:

Enter deposit, rate and years: 10000 10 5

### Program 4.3 Calculate simple interest on deposit

Write a program to calculate simple interest on the amount deposited in a bank. Also calculate the total amount.

**Solution:** The simple interest on deposit is given as  $i = P r n / 100$ , where  $P$  is the principle (i. e., amount deposited),  $r$  is the interest rate per annum and  $n$  is the number of years (i. e., duration of deposit). The total amount to be returned to the depositor is given as  $t = p + i$ .

Let us use the variable names `deposit`, `rate`, `years`, `interest` and `tot_amt`. The variable `years` is assumed to be of type `int` and all other variables to be of type `float`. The

program given below first accepts the values of **deposit**, **rate** and **years**. Then it calculates the values of **interest** and **tot\_amt** and prints them.

```
/* Calculate simple interest on deposit */
#include <stdio.h>
int main()
{
    float deposit, rate;
    int years;
    float interest, tot_amt;
    printf("Enter deposit, rate and years: ");
    scanf("%f %f %d", &deposit, &rate, &years);

    interest = (deposit * rate * years) / 100.0;
    tot_amt = deposit + interest;
    printf("\nSimple interest = %6.2f\n", interest);
    printf("Total amount = %6.2f\n", tot_amt);
    return 0;
}
```

The program output is given below.

```
Enter deposit, rate and years: 1000 12 5
```

```
Simple interest = 600.00
Total amount = 1600.00
```

#### Program 4.4 Convert length in meters to feet and inches

Write a program to accept length in meters and convert it to the feet and inches (FPS system).

**Solution:** Let us use variable **meters** of type **float** to represent length in meters and variables **feet** and **inches** of type **int** and **float**, respectively, to represent length in the FPS system. Let us also use variable **tot\_inches** of type **float** to represent the given length in inches.

To convert the length in meters to the FPS system, we first determine **tot\_inches**, i. e., given length in inches as

```
tot_inches = meters * 100.0 / CM_PER_INCH;
```

where **CM\_PER\_INCH** is a symbolic constant, with a value **2.54**, used to represent the centimeters per inch. Then the length in feet and inches is calculated as follows:

```
feet = tot_inches / 12;
```

```
inches = tot_inches - feet * 12;
```

As the variable **feet** is of type **int**, only the integer part of the division **tot\_inches/12** is assigned to it. The program is given below. It first accepts the value of **meters** and calculates the values of **feet** and **inches** as explained above. Finally, the values of **feet** and **inches** are printed.

```
/* Convert length in meters to feet and inches */
#include <stdio.h>
#define CM_PER_INCH 2.54

int main()
{
    float meters;          /* length in meters */
    int feet;              /* length in feet */
    float inches;          /* and inches */
    float tot_inches;       /* total inches */

    printf("Enter length in meters: ");
    scanf("%f", &meters);
    tot_inches = meters * 100.0 / CM_PER_INCH;
    feet = tot_inches / 12;
    inches = tot_inches - feet * 12;
    printf("%.2fm = %d' %.2f\"\n", meters, feet, inches);

    return 0;
}
```

The program output is given below.

```
Enter length in meters: 1
1.00m = 3' 3.37"
```

Take a close look at the last **printf** statement in this program. Note the use of the escape sequence **\\"** to print a double quote in the output. Also note that we have omitted the minimum field width in the conversion specifications for variables **meters** and **inches** as **%.2f**. This causes the output values to be printed using minimum field width, i. e., closely spaced.

#### Program 4.5 Write a person's name in abbreviated form

Write a program to accept the full name of a person and print it in abbreviated form using initials for the first and middle names.

**Solution:** Let us use character arrays **fname**, **mname** and **lname** to store the first name, middle name and last name of a person. The program given below first accepts the full name, i. e., values of

`fname`, `mname` and `lname` using a `scanf` function. Then the name is printed in the desired abbreviated form using a `printf` function. The initial letters of `fname` and `mname` are printed using `fname[0]` and `mname[0]`, respectively. The program is given below.

```
/* Write person's name in abbreviated form */
#include <stdio.h>
int main()
{
    char fname[20], mname[20], lname[20]; /* person's name */

    /* accept full name */
    printf("Enter full name (first middle last): ");
    scanf("%s %s %s", fname, mname, lname);

    /* print abbreviated name */
    printf("Abbreviated name: ");
    printf("%c. %c. %s\n", fname[0], mname[0], lname);

    return 0;
}
```

The program output is given below.

```
Enter full name (first middle last): Rajan Sadashiv Bichkar
Abbreviated name: R. S. Bichkar
```

#### 4.2.3 Character I/O

##### The **getchar** and **putchar** Macros

The standard C library provides several functions and macros for character I/O. Here we consider the `getchar` and `putchar` macros. As these macros read or write a single character, they are typically used in a loop to read/write a sequence of characters.

A macro call is similar to a function call. Thus, it consists of a macro name followed by a comma-separated argument list enclosed in a pair of parentheses. If a macro does not require any arguments, the pair of parentheses must still be used.

The `getchar` macro is used to read a single character from the standard input stream, i. e., the keyboard. A call to `getchar` takes the following form: `getchar()`. This macro waits until a key is pressed and then returns its value (after converting to integer). The value returned can be assigned to a variable of type `char`. However, note that this macro actually reads the data from the input buffer which is processed by program only when the *Enter* key is pressed. Thus, `getchar` does not return a

value until the user presses the *Enter* key. As a result, it is not suitable for interactive programs. We can instead use the `getch` and `getche` functions which directly read the data from the keyboard.

The **putchar** macro is used to write a single character on the standard output stream (i. e., display). A call to this macro takes the following form: `putchar(c)`, where `c` is an integer expression representing the character being written. Although this macro expects an argument of type `int`, we usually pass a character to it. The argument value (`c`) is converted to `unsigned char` and written to the standard output stream.

### The **getch** and **getche** Functions

The `getchar` macro is not suitable in interactive environments due to the use of line buffering. The C standard library does not provide any facility that is guaranteed to provide an interactive character input. Hence, most C compilers provide alternative functions for use in such interactive environments. These include the **getch** and **getche** functions provided in `<conio.h>` (i. e., console I/O) header file. Like the `getchar` function, calls to these functions take the following forms: `getch()` and `getche()`. These functions are available in Turbo C/C++ as well as Dev-C++. In Visual C++, these functions are named `_getch` and `_getche`.

The **getch** and **getche** functions wait for a key press and immediately return its value. The only difference between these functions is that the **getche** function echos (i. e., displays) the entered character on the screen, whereas the **getch** function does not echo it.

#### 4.2.4 String I/O - the **gets** and **puts** Functions

We can read a string using the `%S` conversion specification in the `scanf` function. However, it has a limitation that the strings entered cannot contain spaces and tabs. To overcome this problem, the C standard library provides the **gets** function. It allows us to read a line of characters (including spaces and tabs) until the newline character is entered, i. e., the *Enter* key is pressed. A call to this function takes the following form:

```
gets(s);
```

where `s` is an array of `char`, i. e., a character string. The function reads characters entered from the keyboard until newline is entered and stores them in the argument string `s`. The newline character is read and converted to a null character (`\0`) before it is stored in `s`. The value returned by this function, which is a pointer to the argument string `s`, can be ignored.

The C standard library provides another function named **puts** to print a string on the display. A typical call to this function takes the following form:

```
puts(s);
```

where **S** is an array of **char**, i. e., a character string. This string is printed on the display followed by a newline character.

#### Example 4.7 String I/O using the **gets** and **puts** functions.

Consider the code segment given below.

```
char str[81];  
  
puts("Enter a line of text:\n");  
gets(str);  
puts("You entered:\n")  
puts(str);
```

A line of text typically contains 80 characters. Thus, the array **str** has been declared to store 81 characters with a provision to store the null terminator. The output of this code segment is shown below.

```
Enter a line of text:  
Programming in C language is fun!  
You entered:  
Programming in C language is fun!
```

### 4.3 Mathematical Library

The C standard library provides several functions for commonly used mathematical operations. These include trigonometric, logarithmic, exponential and other functions as summarized in Table 4.4. Each function takes argument(s) of type **double** and returns a **double** value. The declarations for these functions are provided in the **math.h** header file. Thus, programs using one or more of these functions should include this header file as

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

**Table 4.4** Functions in the standard mathematical library of the C language

Function category	Functions
Powers	<b>sqrt</b> , <b>pow</b> , <b>exp</b>
Logarithmic	<b>log</b> , <b>log10</b>
Trigonometric	<b>sin</b> , <b>cos</b> , <b>tan</b> , <b>asin</b> , <b>acos</b> , <b>atan</b> , <b>atan2</b>
Hyperbolic	<b>sinh</b> , <b>cosh</b> , <b>tanh</b>
Other	<b>ceil</b> , <b>floor</b> , <b>fabs</b> , <b>fmod</b> , <b>modf</b> , <b>ldexp</b> , <b>frexp</b>

### 4.3.1 Powers and Logarithms

The mathematical library provides three functions for evaluation of powers, namely, `sqrt`, `pow` and `exp`. Typical calls to these functions take following form: `sqrt(x)`, `pow(x, y)` and `exp(x)`. The `sqrt` function returns the square root of a given argument. The function call `pow(x, y)` is used to evaluate  $x^y$ , whereas the function call `exp(x)` returns the value  $e^x$ .

The argument of the `sqrt` function must be non-negative; otherwise, it gives a *domain error*. The `pow` function gives a domain error if the value of  $x$  is zero and  $y$  is less than or equal to zero or if  $x$  is negative and  $y$  is not an integer.

The mathematical library provides two functions to evaluate logarithms—`log` for natural logarithm ( $\log_e x$ ) and `log10` for logarithm to base 10. Typical calls to these functions take the following forms: `log(x)` and `log10(x)`. These functions are summarized in Table 4.5.

**Table 4.5** Power and logarithmic functions in the standard mathematical library of the C language

Function name	Typical call	Explanation	Function prototype
<code>sqrt</code>	<code>sqrt(x)</code>	$\sqrt{x}$ , $x \geq 0$ .	<code>double sqrt(double x)</code>
<code>pow</code>	<code>pow(x, y)</code>	$x^y$ . A domain error occurs if $x = 0$ and $y \leq 0$ or if $x < 0$ and $y$ is not an integer	<code>double pow(double x, double y)</code>
<code>exp</code>	<code>exp(x)</code>	exponential function $e^x$	<code>double exp(double x)</code>
<code>log</code>	<code>log(x)</code>	natural logarithm, $\ln(x)$ , $x > 0$	<code>double log(double x)</code>
<code>log10</code>	<code>log10(x)</code>	base 10 logarithm, $\log_{10}(x)$ , $x > 0$	<code>double log10(double x)</code>

#### Example 4.8 Expressions using power and logarithmic functions

Several examples of mathematical expressions involving powers, exponents and logarithms are given below along with the equivalent C expressions.

$$\begin{array}{ll} \sqrt{x+y} & \text{sqrt}(x + y) \\ 5x^7 - 3y^6 & 5 * \text{pow}(x, 7) - 3 * \text{pow}(y, 6) \\ ye^{2x} + (x+y)^3 & y * \text{exp}(2 * x) + \text{pow}(x + y, 3) \\ \log_{10}(\sqrt{x}-\sqrt{y}) & \text{log10}(\text{sqrt}(x) - \text{sqrt}(y)) \\ \frac{x^5 + y^4}{\sqrt{x-y}} & (\text{pow}(x, 5) + \text{pow}(y, 4)) / \text{sqrt}(x - y) \end{array}$$

#### Program 4.6 Real roots of quadratic equation $ax^2 + bx + c = 0$

Write a program to determine the real roots of a quadratic equation  $ax^2 + bx + c = 0$ .

**Solution:** The given quadratic equation has two roots that are given as

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \text{ i. e., } x_{1,2} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } x_{1,2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

The program given below uses these equations directly to evaluate the roots of a quadratic equation. The **main** function declares variables **a**, **b** and **c** of type **float** to represent the coefficients of the quadratic equation, and variables **x1** and **x2** also of type **float** to represent the roots. The values of coefficients are first accepted from the keyboard using the **scanf** statement. Then the roots **x1** and **x2** are calculated and printed. Note that in addition to **stdio.h**, the header file **math.h** is also included as the program contains a declaration for the **sqrt** function.

```
/* Determine roots of a quadratic equation */
#include <stdio.h>
#include <math.h>

int main()
{
    float a, b, c; /* coefficients of quadratic equation */
    float x1, x2; /* roots */
    /* read coefficients */
    printf("Enter coefficients of quadratic equation\n");
    scanf("%f %f %f", &a, &b, &c);

    /* determine and print roots */
    x1 = (-b + sqrt(b * b - 4.0 * a * c)) / (2.0 * a);
    x2 = (-b - sqrt(b * b - 4.0 * a * c)) / (2.0 * a);
    printf("Roots are x1 = %4.2f x2 = %4.2f\n", x1, x2);

    return 0;
}
```

The program output is given below.

```
Enter coefficients of quadratic equation
1 3 2
Roots are x1 = -1.00 x2 = -2.00
```

Recall that the argument of the **sqrt** function must be non-negative. Thus, when discriminator  $b^2 - 4ac$  is negative, i. e., the roots of the quadratic equation are complex conjugate, the **sqrt** function gives *domain error*, as illustrated below.

```
Enter coefficients of quadratic equation
```

```
1 2 3
sqrt: DOMAIN error
sqrt: DOMAIN error
Roots are x1 = -1.00 x2 = -1.00
```

In the next chapter, we will study how such erroneous situations can be avoided (using **if** statement) and complex conjugate roots can be evaluated.

The above output is obtained by running the program in Turbo C. Observe that the values of the roots are incorrect. When the program is executed in Turbo C++, a domain error is reported as before. However, the values of the roots are displayed as **+NAN**, as shown below.

```
Roots are x1 = +NAN x2 = +NAN
```

where **NAN** stands for *not a number*. Finally, when the program was executed in Dev++, the domain error was not reported and the roots were printed as shown below.

```
Roots are x1 = -1.#J x2 = -1.#J
```

Note that we can make this program more efficient by avoiding repetitive calculations in the evaluation of equations for  $x_1$  and  $x_2$  as shown below.

```
p = -b / (2.0 * a);
q = sqrt(b*b - 4.0 * a*c) / (2.0 * a);
x1 = p + q;
x2 = p - q;
```

### 4.3.2 Trigonometric and Hyperbolic Functions

The trigonometric and hyperbolic functions provided in the standard mathematical library are listed in Table 4.6. Except for the **atan2** function, which takes two arguments, all other functions take a single argument and each function returns a single value. Note that the function parameters and return values are of type **double** as mentioned earlier.

The **sin**, **cos** and **tan** functions are used for the evaluation of sine, cosine and tangent, respectively, of a given angle, which must be specified in radians.

The **asin**, **acos** and **atan** functions return arc sine (i. e.,  $\sin^{-1}$ ), arc cosine (i. e.,  $\cos^{-1}$ ) and arc tangent (i. e.,  $\tan^{-1}$ ), respectively. The function call **atan2** ( $y$ ,  $x$ ) returns the arc tangent of  $y/x$ . The angle returned by these functions is in radians. Note that the argument of the **asin** and **acos** functions must be in the range  $-1.0$  to  $1.0$ , both inclusive; otherwise, they give domain error. Similarly, the argument of **atan** function must be in the range  $-\pi/2$  to  $\pi/2$ . The functions **sinh**, **cosh** and **tanh** are used to calculate the hyperbolic sine, hyperbolic cosine and hyperbolic tangent, respectively.

**Table 4.6** Trigonometric and hyperbolic functions in the standard library of the C language

Function name	Typical call	Explanation	Function prototype
sin	sin(x)	sine of x	double sin(double x)
cos	cos(x)	cosine of x	double cos(double x)
tan	tan(x)	tangent of x	double tan(double x)
asin	asin(x)	$\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$ , $x \in [-1, 1]$	double asin(double x)
acos	acos(x)	$\cos^{-1}(x)$ in range $[0, \pi]$ , $x \in [-1, 1]$	double acos(double x)
atan	atan(x)	$\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$	double atan(double x)
atan2	atan2(y, x)	$\tan^{-1}(y/x)$ in range $[-\pi, \pi]$	double atan2(double y, double x)
sinh	sinh(x)	hyperbolic sine of x	double sinh(double x)
cosh	cosh(x)	hyperbolic cosine of x	double cosh(double x)
tanh	tanh(x)	hyperbolic tangent of x	double tanh(double x)

While using functions in this group, we may have to convert the angles between degrees and radians. These conversions are given as  $\theta_r = (\pi/180)\theta_d$  and  $\theta_d = (180/\pi)\theta_r$ , where  $\theta_d$  and  $\theta_r$  are angles in degrees and radians, respectively.

#### Example 4.9 Expressions using trigonometric functions

Assume that angles  $\theta$  and  $\phi$  are in degrees and angle  $\psi$  is in radians. Let us use variables **theta**, **phi** and **psi** (all of type **float** or **double**) to represent these angles. The symbolic constants to convert angles between degrees and radians are defined as follows:

```
#define PI 3.1415927
#define DEG_TO_RAD (PI / 180)
#define RAD_TO_DEG (180 / PI)
```

Several mathematical equations are given below along with the equivalent C expressions.

- |                                       |  |
|---------------------------------------|--|
| a) $\sin \theta$                      | <b>sin(PI / 180 * theta)</b>           |
| b) $\cos(\theta + \phi)$              | <b>cos(DEG_TO_RAD * (theta + PHI))</b> |
| c) $\theta = \sin^{-1}(x/y)$          | <b>theta = RAD_TO_DEG * asin(x/y)</b>  |
| d) $\sin \psi$                        | <b>sin(psi)</b>                        |
| e) $\psi = \sin^{-1} x + \cos^{-1} y$ | <b>psi = asin(x) + acos(y)</b>         |

Note that the arguments in examples (a) and (b) are converted to radians. Also, the angle returned by the **asin** function in example (c) is converted to degrees before it is assigned to **theta**. Finally, note that the expressions in examples (d) and (e) do not require any angle conversions.

#### Program 4.7 Calculate the length of a belt around two pulleys

Fig. 4.1 shows two pulleys having diameters  $d_1$  and  $d_2$ , respectively, and the distance between their centers as  $c$ . Write a program to accept the values of  $d_1$ ,  $d_2$  and  $c$  and calculate the length of the belt ( $L$ ) around the pulleys which is given as

$$L = \sqrt{4c^2 - (d_1 - d_2)^2} + \frac{\pi}{2}(d_1 + d_2) + (d_1 - d_2)\sin^{-1}\frac{d_1 - d_2}{2c}$$

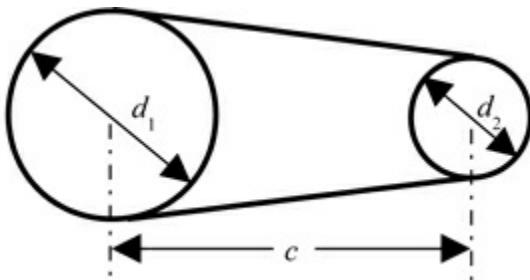


Fig. 4.1 Two pulleys with a belt around them

Solution: As the term  $d_1 - d_2$  appears thrice in the equation for belt length  $L$ , let us substitute  $d = d_1 - d_2$ . The equation now becomes

$$L = \sqrt{4c^2 - d^2} + \frac{\pi}{2}(d_1 + d_2) + d \sin^{-1}\frac{d}{2c}$$

This equation is still quite complex. Hence let us split this equation in three parts as

$$L = \sqrt{4c^2 - d^2}, \quad L = \frac{\pi}{2}(d_1 + d_2) \quad \text{and} \quad L = d \sin^{-1}\frac{d}{2c}$$

Now the equation for belt length is given as  $L = L_1 + L_2 + L_3$ .

The program given below first accepts the value of  $d_1$ ,  $d_2$  and  $c$ . Then it evaluates the values of  $d$ ,  $L_1$ ,  $L_2$ ,  $L_3$  and  $L$  in that order. Finally, the belt length is printed. Observe that the program includes the standard header file `math.h` as it uses the `sqrt` and `asin` functions from the mathematical library.

```
/* Calculate length of belt around two pulleys */
#include <stdio.h>
#include <math.h>

#define PI 3.1415927

int main()
{
```

```

float d1, d2;          /* diameters of pulleys */
float c;                /* distance between pulleys */
float L;                /* length of belt around pulleys */
float L1, L2, L3;        /* terms in eqn for belt length */
float d;                /* difference in pulley diameters (d1-d2) */
/* read data */
printf("Enter diameters of pulleys: ");
scanf("%f %f", &d1, &d2);
printf("Enter distance between pulleys: ");
scanf("%f", &c);

/* calculate belt length */
d = d1 - d2;
L1 = sqrt(4.0 * c * c - d * d);
L2 = PI * (d1 + d2) / 2.0;
L3 = d * asin(d / (2.0 * c));
L = L1 + L2 + L3;

printf("Belt length = %6.2f\n", L);
return 0;
}

```

The program output is given below:

```

Enter diameters of pulleys: 20.5
Enter distance between pulleys: 50
Belt length = 140.40

```

#### 4.3.3 Other Mathematical Functions

In addition to the functions discussed thus far, the mathematical library provides seven other functions. These include `ceil`, `floor`, `fabs`, `fmod`, `modf`, `ldexp` and `frexp`. Each of these functions takes argument(s) of type `double` and returns a `double` value. In this section, we study the first four of these functions. They are summarized in Table 4.7.

**Table 4.7** *Other functions in the mathematical library of the C language*

Function name	Typical call	Explanation	Function prototype
ceil	ceil(x)	$\lceil x \rceil$ , Smallest value greater than or equal to $x$	double ceil(double x)
floor	floor(x)	$\lfloor x \rfloor$ , Largest value less than or equal to $x$	double floor(double x)
fabs	fabs(x)	$ x $ , Absolute value of $x$	double fabs(double x)
fmod	fmod(x, y)	$x$ modulo $y$ , i. e., Remainder of $x/y$	double fmod(double x, double y)

The **ceil** and **floor** functions perform rounding operations. The **ceil** function rounds up, i. e., it returns the smallest integer greater than or equal to the argument specified. For example, both the calls, `ceil(1.1)` and `ceil(1.9)`, return 2.0. The **floor** function, on the other hand, rounds down. It returns the largest integer less than or equal to the argument specified. Thus, both the calls, `floor(1.1)` and `floor(1.9)` return 1.0.

The **fabs** function returns absolute value of a floating-point number. Thus, both the calls, `fabs(1.2)` and `fabs(-1.2)` return 1.2. The C standard library provides three more functions to determine absolute values, namely, **abs**, **labs** and **cabs**. The **abs** and **labs** functions determine the absolute value of an **int** and **long int** argument. They are declared in **stdlib.h** header file. In Turbo C/C++, these functions are also declared in **math.h** header file. The **cabs** function, which is declared in **math.h** header file returns the absolute value of a complex number.

In the last chapter, we studied the modulus operator (%) which calculates the remainder after integer division. However, this operator cannot be used with floating-point numbers. Hence, the C language has provided the **fmod** function which calculates the remainder of the division of two floating point numbers specified as its arguments.<sup>~1</sup> To be more specific, the function call `fmod(x, y)` returns the remainder of  $x/y$ . Thus, the call `fmod(1.25, 0.3)` returns 0.05 and `fmod(-1.25, 0.3)` returns -0.05.

## 4.4 Advanced Concepts

This section discusses some advanced concepts related to the topics covered in this chapter. As mentioned in the previous chapter, a beginner may skip this section and return to it later after studying more important and useful concepts presented in subsequent chapters.

### 4.4.1 Streams

A **stream** is a source or destination of data associated with a disk or other peripheral such as keyboard, display, etc. The standard library supports two types of streams: *text* and *binary*. A text stream is a sequence of lines, each containing zero or more characters terminated by *newline* ('\n'). The streams associated with the keyboard and display are text streams. Binary streams are discussed in Chapter 14.

Before we perform I/O operations with a file or device, we should connect it to a stream. This is achieved by opening the stream. The connection is broken when we close the stream.

When program execution begins, three streams, namely, `stdin`, `stdout` and `stderr` are already open. The `stdin` is the standard input stream that is usually connected to the keyboard, whereas `stdout` and `stderr` are standard output and standard error streams that are usually connected to the display. When program execution is over, these streams are automatically closed.

The `scanf`, `getchar` and `gets` functions actually read data from the standard input stream (i.e., keyboard), whereas the `printf`, `putchar` and `puts` functions write the output to the standard output stream (i.e., display).

#### 4.4.2 Function Prototypes

A standard C header file contains the declarations or prototypes of functions of a particular category. A **function prototype** usually specifies the type of value returned by that function, the function name and a list specifying parameter types as

```
ret_type func_name ( type1, type2, ... );
```

where *func\_name* is the name of the function, *ret\_type* is the type of value returned by it and *type1*, *type2*, ... are the types of first, second, ... function parameters. Note that the parameter type list is comma separated and is enclosed within a pair of parentheses. It specifies the number of arguments required in function calls and the type of each argument. When a function is called, an argument of appropriate type must be specified for each of the parameter types specified in the function prototype.

We can either omit the parameter type list or use the `void` keyword in its place to indicate that the function does not have any parameters, i.e., it does not require any arguments. In addition, the `void` keyword may be used as the function return type to indicate that the function does not return any value. However, note that if the return type is omitted, the function is assumed to return a value of type `int`.

A function prototype may also contain the parameter names as shown below:

```
ret_type func_name ( type1 param1, type2 param2, ... );
```

where *param1* is the name of first parameter, *param2* is the name of the second parameter and so on. The inclusion of parameter names in function prototypes clarifies the order in which arguments are to be specified in a function call.

#### Example 4.10 Function prototypes

For example, the prototypes of the `sqrt` and `pow` standard library functions are given below.

```
double sqrt(double);
double pow(double, double);
```

The `sqrt` function returns the square root of the argument expression specified in its call. It accepts one argument of type `double` and returns a value of type `double`. On the other hand, the `pow` function which calculates  $x^y$  accepts two arguments of type `double` and returns a value of type `double`. This prototype can also be written using parameter names as

```
double pow(double x, double y);
```

Now consider the prototypes for the `getmaxx` and `setcolor` functions given in the `graphics.h` header file of Turbo C/C++.

```
int getmaxx(void);
void setcolor(int color);
```

Observe that the `getmaxx` function does not accept any argument and the `setcolor` function does not return any value.

#### 4.4.3 Formatted Output using the `printf` Function

The `printf` function allows the values of argument expressions to be printed in various formats. A large number of conversion characters are provided for printing expressions of different types. Also, the possibility of using several optional fields along with these conversion characters makes `printf` a very powerful and complicated function. In this section, we study only the commonly used formatting features of the `printf` function.

For a value being printed in a `printf` statement, we can specify the minimum field-width, precision, justification (left or right), etc. using the format given below.

%	Flag (-)	Minimum field width	.	Precision	Conversion character (d f c s, etc.)
---	----------	---------------------	---	-----------	--------------------------------------

When minimum field-width is not specified in a conversion specification, the corresponding argument is printed using as many character positions as necessary. However, if it is specified, the argument is printed in a field at least that wide, using additional spaces as necessary. If the field-width specified is more than what is required, the output values are right justified in the output field, unless the '-' flag is used which causes the values to be left justified. If the specified field-width is insufficient for the value being printed, additional character positions are used as necessary.

When printing floating point numbers, precision specifies the minimum number of digits to be printed after the decimal point. If a number being printed has fewer digits after the decimal point than specified, it is padded with zeros. However, if it has more digits after the decimal point, its value is rounded to the specified digit. Thus, the conversion specification `%8.4f` causes numbers 100.12 and 123.12345 to be printed as 100.1200 and 123.1235, respectively. Note that if we omit precision in the conversion specification, a default precision of six digits is used. In this section, we use the precision field only for floating point numbers.

### Example 4.11 Formatted output

#### a) Using minimum field-width and precision

Consider the `printf` statements given below that print values of different types by specifying minimum field-width for each data value and the precision for the floating values:

```
printf("%10d\n", 12);
printf("%10.7f %10.4f\n", 1.23456789, -1.2);
printf("%10s\n", "Apple");
printf("%10c\n", 'A');
```

The output of these statements is given below.

```
12
1.2345679      -1.2000
Apple
A
```

Observe that the values are printed right justified in field-width of 10. Also observe that the first floating number is rounded to the seventh decimal place, whereas the second one is printed using additional zeros.

#### b) Effect of specifying insufficient field-width

```
printf("%4s %4d %4f\n", "Monday", 12345, 1.2);
```

This statement uses a field-width of four for each constant to be printed. Note that this field-width is not sufficient for any of the constants, including 1.2, as the default precision for a floating point number is six digits. Thus, the values are printed using as many character positions as required. The output is given below.

```
Monday 12345 1.200000
```

#### c) Printing tabular output (Specifying field-widths and alignment)

This example illustrates how we can use the formatting features of `printf` function to print data properly aligned in a tabular manner.

```
printf("%-10s %6d %12.3f %4c\n", "Monday", 5, 2.1234, 'A');
printf("%-10s %6d %12.3f %4c\n", "Tuesday", 16, 17.5126, 'B');
printf("%-10s %6d %12.3f %4c\n", "Wednesday", 175, 120.25, 'C');
```

This code has three similar `printf` statements, each used to print four values using an identical

format. Note that the minimum field-widths specified in the conversion specifications are more than what is required. Thus, the values will be printed right justified unless the '-' flag is used.

As the strings are printed using "-10s" conversion specification, each string is printed left justified in a field-width of ten characters, whereas the values of other arguments are printed right justified in respective fields. The floating numbers are printed using three decimal places after the decimal point. The output is given below.

Monday	5	2.123	A
Tuesday	16	17.513	B
Wednesday	175	120.250	C

#### 4.4.4 Character Classification and Conversion

The C standard library provides several functions for character classification and conversion. These functions are summarized in Table 4.8. The declarations of these functions are provided in the `ctype.h` header file.

The standard library provides the **tolower** and **toupper** functions for character **case conversion**. These functions convert the case of a given character to lowercase and uppercase, respectively, and return this value. If the given character is not a letter, it is returned unchanged.

The character **classification** functions test whether a given character is in a particular category (such as letter, digit, whitespace, etc.) or not and return an integer value indicating true or false. These functions are of the form **isxxx** and a typical call to these functions takes the following form: **isxxx(ch)** where **ch** is the character being tested.

**Table 4.8** *Character classification and conversion functions in the C standard library*

Category	Function call	Explanation
Test letters	<code>islower(c)</code>	Returns true if $c$ is a lowercase letter
	<code>isupper(c)</code>	Returns true if $c$ is an uppercase letter
	<code>isalpha(c)</code>	Returns true if $c$ is a letter
	<code>isalnum(c)</code>	Returns true if $c$ is an alphanumeric character (letter or digit)
Test digits	<code>isdigit(c)</code>	Returns true if $c$ is a digit
	<code>isxdigit(c)</code>	Returns true if $c$ is a hexadecimal digit
Test space and special characters	<code>isspace(c)</code>	Returns true if $c$ is a whitespace character (space, tab, new line, etc.)
	<code>ispunct(c)</code>	Returns true if $c$ is a punctuation character, i. e., a printing character other than letter, digit or space
	<code>isprint(c)</code>	Returns true if $c$ is a printable character
	<code>isgraph(c)</code>	Returns true if $c$ is a printable character other than space
	<code>iscntrl(c)</code>	Returns true if $c$ is a control character
Case conversion	<code>tolower(c)</code>	Converts character $c$ to lowercase
	<code>toupper(c)</code>	Converts character $c$ to uppercase

The commonly used classification functions in the standard library include `isalnum`, `isalpha`, `isupper`, `islower`, `isdigit`, `isxdigit`, `isspace` and `ispunct`. In addition, three other functions, namely, `iscntrl`, `isgraph` and `isprint` are available. These functions actually accept an integer argument representing a character and return an integer value indicating true or false.

#### 4.4.5 Manipulating Strings

Several operations need to be performed on character strings, the most common being copy, concatenation, comparison, search and replace, case conversion and length determination. However, the C language does not provide any operators to perform operations on strings. Instead, it provides a large number of standard library functions to manipulate strings. The names of most of these functions start with "str" or "mem". The declarations of these functions are provided in the `string.h` header file. Thus, a program using any of these functions should include this header file as

```
#include <string.h>
```

Table 4.9 summarizes some commonly used string manipulation functions in the C language. Note that the `strcmp`, `strupr` and `strlwr` functions are not defined in the C standard library.

**Table 4.9** Commonly used string manipulation functions in the C language

Category	Function	Typical call	Description
Copy	<code>strcpy</code>	<code>strcpy(dest, src)</code>	Copies string <i>src</i> to <i>dest</i> ; return pointer to <i>dest</i>
Concatenation	<code>strcat</code>	<code>strcat(dest, src)</code>	Concatenates string <i>src</i> to <i>dest</i> ; return pointer to <i>dest</i>
Comparison	<code>strcmp</code>	<code>strcmp(s1, s2)</code>	Compares strings <i>s1</i> and <i>s2</i> (case-sensitive)
	<code>stricmp<sup>†</sup></code>	<code>stricmp(s1, s2)</code>	Compares strings <i>s1</i> and <i>s2</i> ignoring case
Search	<code>strchr</code>	<code>strchr(s, c)</code>	Finds first occurrence of character <i>c</i> in string <i>s</i>
	<code>strrchr</code>	<code>strrchr(s, c)</code>	Finds last occurrence of a character in a string
	<code>strstr</code>	<code>strstr(s1, s2)</code>	Searches for a substring <i>s2</i> in a string <i>s1</i>
Case conversion	<code>strlwr<sup>†</sup></code>	<code>strlwr(s)</code>	Converts string <i>s</i> to lowercase
	<code>strupr<sup>†</sup></code>	<code>strupr(s)</code>	Converts string <i>s</i> to uppercase
Length	<code>strlen</code>	<code>strlen(s)</code>	Determines length of string <i>s</i>

<sup>†</sup> `strcmp`, `strupr` and `strlwr` are not defined in the standard C library

In this section, we study the functions for string copy (`strcpy`), string concatenation (`strcat`) and string length (`strlen`). In addition, two case conversion functions (`strupr` and `strlwr`) are also discussed.

Note that the functions in the copy and concatenation categories (such as `strcpy` and `strcat`) write characters to target string which is basically a character array. It is the programmers' responsibility to ensure that the target array has enough room to accommodate new characters written to it. The C compiler does not give any error or warning if we try to write beyond the limits of the target array. In such situations, program behavior is likely to be unpredictable as the extra characters written to the target array may destroy some other important data.

## String Copy

String copy is one of the basic operations in manipulating strings. In this operation, all the characters in the source string, including the null terminator, are copied to the target string making it an exact replica of the source string.

As we already know, we cannot copy a string using an assignment operator, as in `str2 = str1`. Instead, the C standard library provides the `strcpy` and `strncpy` functions to copy a string. The `strcpy` function copies the entire string, whereas the `strncpy` function copies at most the first *n* characters. A typical call to the **strcpy function** takes the following form:

As we already know, we cannot copy a string using an assignment operator, as in `str2 = str1`. Instead, the C standard library provides the `strcpy` and `strncpy` functions to copy a string. The `strcpy` function copies the entire string, whereas the `strncpy` function copies at most the first *n* characters. A typical call to the **strncpy function** takes the following form:

**`strcpy (dest, src);`**

This function copies string `src` to `dest` including the null terminator and it returns a pointer to `dest`, which is often ignored, as in the above call. As mentioned before, the array `dest` must have enough character positions to accommodate string `src`. Observe that the order of arguments `dest` and `src` follows the conventions of assignment operation in which the variable being assigned is on the left hand side.

### Example 4.12 Copy strings

The code given below performs string copy using the `strcpy` function.

```
char str1[20] = "Orange";
char str2[20] = "Pineapple";
strcpy(str2, str1);
printf("str1: %s str2: %s\n", str1, str2);
```

The character arrays `str1` and `str2` can store up to 20 characters and are initialized with strings "Orange" and "Pineapple", respectively. The `strcpy` function copies entire string `str1` to `str2`, overwriting its contents. Seven characters are copied in this operation, including the null terminator. Thus, the output of this code is as follows:

```
str1: Orange str2: Orange
```

### String Concatenation

String concatenation is another basic string manipulation operation. In this, all the characters in a string, including the null terminator, are appended to the target string (i. e., copied at the end) replacing the null terminator in it. Thus, after the concatenation operation, the target string contains all the characters originally contained in it followed by those in the appended string. For example, if string `s1` contains "Pine" and string `s2` contains "apple", then appending string `s2` to `s1` modifies string `s1` to "Pineapple". Note that `s1` must have enough character positions to accommodate all the appended characters, including the null terminator.

We cannot use the `+=` operator to append a string to another string, as in `str2 += str1`. Instead, the C standard library provides the `strcat` and `strncat` functions to concatenate strings. The `strcat` function concatenates an entire string to the target string. The `strncat` function, on the other hand, appends at the most `n` characters.

A typical call to the **strcat function**, which is similar to the `strcpy` function call, is as follows:

```
strcat(dest, src);
```

This function appends string `src` to `dest` and returns a pointer to string `dest`, which is usually ignored.

### Example 4.13 Concatenate strings

The program segment given below accepts the user's first name and last name and concatenates them to construct a string containing his/her full name.

```
char fname[20], lname[20];
char full_name[40];

printf("Enter your first name: ");
gets(fname);
printf("Enter your last name: ");
gets(lname);

strcpy(full_name, fname);
strcat(full_name, " ");
strcat(full_name, lname);

printf("Your full name: %s\n", full_name);
```

The character arrays `fname`, `lname` and `full_name` are used to store the first name, last name and full name of a person. Note that the first and last names can each have at most 19 characters. Thus, 40 characters reserved for full name are adequate for the concatenated string ( $19 \times 2$  for names + 1 space + null terminator).

Initially, the `gets` functions read the user's first and last names from the keyboard. Then the `strcpy` function is used to copy the first name to `full_name` followed by two calls to the `strcat` function to append a space and last name to `full_name`. The output of this code segment is given below.

```
Enter your first name: Rajan
Enter your last name: Bichkar
Your full name: Rajan Bichkar
```

### String Length

The length of a string is defined as the number of characters in it excluding the null terminator. The **strlen function** returns the length of a specified string. A typical call to this function takes the following form:

**strlen(s)**

For example, the program segment given below determines the length of a string using the `strlen` function and prints it.

```
char str[] = "EVERYTHING is fair in LOVE and WAR";
```

```
printf("Length : %d\n", strlen(str));
```

### Case Conversion

Another commonly required string operation is that of converting the case of a given string. The C standard library does not provide any function for case conversion. However, some C implementations provide the **strlwr** and **strupr** functions. The **strlwr function** converts all characters in a given string to lowercase, whereas the **strupr function** converts all characters to uppercase. Typical calls to these functions take the following forms:

```
strlwr(s);  
strupr(s);
```

Note that these functions modify the string passed as an argument and return a pointer to the modified string. The program segment given below converts (and prints) a string to uppercase and lowercase representations.

```
char str[] = "EVERYTHING is fair in LOVE and WAR";  
  
printf("Uppercase: %s\n", strupr(str));  
printf("Lowercase: %s\n", strlwr(str));
```

The output of this program segment is shown below.

```
Uppercase: EVERYTHING IS FAIR IN LOVE AND WAR  
Lowercase: everything is fair in love and war
```

### 4.4.6 Utility Functions

The C standard library provides several utility functions. These include functions for integer arithmetic, string conversion, random number generation, searching and sorting, program control and communication with the environment. The declarations of these functions are provided in the **stdlib.h** standard header file. In this section, we discuss some commonly used functions in this group. These functions are summarized in Table 4.10.

**Table 4.10** Commonly used utility functions in C language (declared in the **stdlib.h** header file)

Category	Function	Typical call	Description
Integer arithmetic	abs	abs( <i>n</i> )	Returns absolute value of an integer <i>n</i>
	labs	labs( <i>n</i> )	Returns absolute value of a long integer <i>n</i>
String conversion	atoi	atoi( <i>s</i> )	Converts string <i>s</i> to integer and return it
	atof	atof( <i>s</i> )	Converts string <i>s</i> to double and return it
	atol	atol( <i>s</i> )	Converts string <i>s</i> to long integer and return it
Random number	rand	rand()	Returns a pseudo-random number (0 to RAND_MAX)
	srand	srand( <i>k</i> )	Reseeds random number generator using <i>k</i> as seed
Program control	exit	exit( <i>k</i> )	Terminates the program with exit status <i>k</i>
	abort	abort()	Abnormally terminates a program

## Integer Arithmetic

The **abs** and **labs** functions return the absolute value of a number of type **int** and **long int**, respectively. The prototypes of these functions are given below.

```
int abs(int);
long int labs(long int);
```

Recall that the declaration of the **fabs** function, used to determine the absolute value of a **double** number, is provided in the **math.h** header file. However, the declarations of the **abs** and **labs** functions are provided in the **stdlib.h** header file.

## Converting Strings to Numbers

The **atoi**, **atof** and **atol** utility functions are used to perform string to numeric conversion. As indicated by the last letter in function names, these functions convert the argument string to an integer, floating and long number. Note that the argument string should contain a number of appropriate type as text, optionally preceded by whitespace. For example, the **atof** function can successfully convert strings such as "1.23", "3.21E10", etc. The conversion stops when an inappropriate character is encountered in the input string. Thus, these functions will also be able to convert strings such as "-1.23ABC", "3.21E10ABC". If conversion is successful, these functions return the converted value, otherwise zero.

The **atoi** and **atol** functions convert the argument string to an integer number of type **int** and **long int**, respectively. The initial portion of the string should contain an integer number in the decimal format  $\pm ddd$ , in the range of data type being converted to (i. e., **int** or **long int**).

The **atof** function, on the other hand, converts the argument string to a floating number of type **double**. The initial portion of the string should contain a floating number in either decimal or scientific notation, in the range of type **double**.

## Random Number Generation

Random number generation is a very useful facility in many programming situations, particularly gaming and simulation. We can also use it to generate test data for programs, particularly when a large amount of data is required. This saves a lot of time on data entry.

The C standard library provides a function named **rand** to generate pseudo-random numbers. This function is declared in the **stdlib.h** header file. It does not require any argument and returns an integer number in the range 0 to **RAND\_MAX**, where **RAND\_MAX** is a constant defined in **stdlib.h** with value 32767.

We can manipulate the value returned by the **rand** function to generate integer or floating-point numbers in a desired range. The expression

```
(float) rand() / RAND_MAX
```

gives a random number of type **float** in the range [0.0, 1.0], i. e., 0 to 1.0, both inclusive. Note the use of typecast (**float**) to avoid integer division. To generate a random number in a desired range *m* to *n*, we can now use the following expression:

```
m + (m - n) * (float) rand() / RAND_MAX
```

The **rand** function actually generates pseudo-random data. Thus, each time the program is executed, it generates the same sequence of random numbers. This may not be acceptable in many situations. Hence, another function called **srand** is provided to reseed the random number generator. A typical call to this function takes the following form:

```
srand(seed)
```

where *seed* is an unsigned integer. This call reseeds the random number generator so that a different sequence of random numbers is generated on subsequent calls to **rand**.

## Program Termination Functions

The **exit** function can be used to terminate program execution and return a specified value as program status to the calling program, usually the operating system. A zero value indicates success. An example of the **exit** function indicating unsuccessful termination of a program is shown below.

```
exit(1);
```

The **abort** function, on the other hand, is used to abnormally terminate the program. This function displays a message "Abnormal Program termination" and exits the program. A typical call to this function takes the following form:

```
abort();
```

#### 4.4.7 Miscellaneous Functions

While using Turbo C/C++, the output of a program is displayed along with the previous output if any. We can use the **clrscr** function provided in the **conio.h** header file to clear the screen. This function does not take any argument nor does it return any value. Thus, a call to this function takes the following form:

```
clrscr();
```

### Exercises

1. Several **scanf** statements are given below. Identify errors, if any and rewrite incorrect statements correctly. Assume that the variables are declared as in the previous example.
  - a. **scanf &a;**
  - b. **scanf(&a, &b, &c)**
  - c. **scanf("&x, &y, &z");**
  - d. **scanf("%d", &a, &b, &c);**
  - e. **scanf("%f, %f, %f", @x, @y, @z);**
  - f. **scanf("Enter a char %c:", &ch1);**
  - g. **scanf{"%f %f", &(x+y), &(x-y)};**
  - h. **scan f("%5d%5d", &c, &d);**
  - i. **Scanf("%d", &ch1);**
  - j. **scanf("Hello world");**
  - k. **scanf("%f+i%f", &x, &y);**
  - l. **scanf("%d %d", &a, &a);**
  - m. **scanf("% % %", &x, &y, &z);**
  - n. **c = scanf("%c %c", &ch, &ch1);**
  - o. **scanf("%f", &fabs(x));**
  - p. **SCANF('%D %D', A, B);**
2. Several **printf** statements are given below. Identify errors, if any, and rewrite incorrect statements correctly. Assume that the variables used in these statements are declared as follows:  
**int a, b, c;**  
**float x, y;**

- ```
double z;
char chi, ch2;

a. print Hello world!;

b. printf('C programming is fun')

c. printf(a, b, c);

d. printf("%d\n %d\n %d, a,b,ch1");

e. printf("%f; %f", x, y, z);

f. printf("\tValue of a is #d\a" a);

g. printf("%f+%f %f-%f", x+y, x-y);

h. print f("%5d%5.2f%5c", a, x, ch);

i. printf("%f %f", &a, &b);

j. printf{"x:%3.5f; y:%2.3f;",y,x};

k. printf("%x %y %z", x, y, z);

l. c = printf("ch1: %c\n", (ch1));

m. Print("%s: %d %f %c", "Values of all variables: ", a,b,c,
       x,y,z, ch1,ch2,ch3);

n. printf("The values of variables a, b, c, are ", a, b, c);

o. printf("%d %d",sqrt(x),pow(x,5));

p. printf[%d %d, a, b];

3. Several program segments containing scanf and printf functions are given below. Determine their exact output. Indicate column numbers in the output. Note that the value to be entered for scanf functions are written as comments on same line.

a. int x = 50, y = 20;
   printf("x = \t", x);
   printf("y = \n", y);

b. printf("Apple");
   printf("Orange\t");
   printf("Guava\n");
   printf("Lemon");

c. printf("%d%c%f%s", 501, '+', 1.21, "SurPrise");
   printf("%5d %5c %20s %10.1f", 10, 65, "Hi, " "there!", 1.2);
   printf("%d %d", 'a', 'A' + 5);
```

- d. float x;  
`printf("Enter a number: ");`  
`scanf("%f", &x); /* 4 */`  
`printf("%f\t%f\t%f", x, x*x, sqrt(x));`
- l. Several statements containing standard library function calls are given below. Identify errors, if any, in these statements and rewrite incorrect statements correctly. Assume that the variables are declared as in question 1.
- a. `x = sqr(a*a + b*b);`
  - b. `c = power(a, b);`
  - c. `z = log10 x + loge y;`
  - d. `z = expx + expy`
  - e. `c = sin(x+35°); /* x in degrees */`
  - f. `d=tan(180/π*a); /* a in degrees */`
  - g. `z = sin-1(x * y + z);`
  - h. `getchar(&ch1);`
  - i. `ch2 = getche("%c");`
  - j. `puts(Hello, world);`
5. Write equivalent mathematical equations for the following expressions.
- a. `x = sqrt(a * a + b * b + c * c)`
  - b. `Y = pow(a, 5) + pow(b, 5) / pow(a + b, 5)`
  - c. `z = log10(floor(x+y)) + exp(ceil(x - y)) / exp(x + y)`
  - d. `z = (pow(sin(x + y), 4) - pow(cos(x - y), 4)) / log (x * x + y * y)`
  - e. `z = (exp(a * x) + exp(b * y)) / (a + b)`
5. Evaluate the expressions given below for two sets of variable values (i)  $a = 2, b = 3, c = 4$  and (ii)  $a = 3, b = 1, c = 1$
- a.  $(\text{pow}(a, 3) + \text{pow}(b, 3)) / (a + b)$
  - b.  $\text{floor}(\sqrt{2 * a + b * c}) / 3 + \text{ceil}(10.0 / (a+1) / 2)$
  - c.  $\text{ceil}(\sqrt{(\text{pow}(a, 3) + \text{pow}(b, 3)) / (a + b)})$
  - d.  $\text{pow}(\text{abs}(a - c) + \text{fabs}(b - c), 3)$

7. Write equivalent C expressions for the mathematical equations given below

a.  $c = ax^3 + by^6$

b.  $\sqrt{x^2 + y^2}$

c.  $\sqrt{\sin^2(x \log_e \sqrt{a+b})}$

d.  $c = \frac{a^{x+y} + b^{x-y}}{x - y}$

e.  $x = \sqrt[3]{\frac{a^3 + b^4}{a^4 - b^3}}$

f.  $\frac{e^{x+y} + e^{-(x+y)}}{\sqrt{x^2 + y^2}}$

3. Write the complete C program for the following problems:

a. Calculate compound interest on fixed deposit, compounded every six months.

b. A bank cashier has currency notes in eight denominations: 1000, 500, 100, 50, 20, 10, 5 and 1. Given an amount, determine how he/she should give it to a bank customer using the minimum number of available currency notes.

c. Convert the Cartesian coordinates of a point  $(x, y)$  to polar form  $(r, \theta)$  given as  $r = \sqrt{x^2 + y^2}$  and  $\theta = \tan^{-1}(y/x)$ .

d. Given the lengths of three sides of a triangle, determine its area.

## Exercises (Advanced Concepts)

1. Identify errors, if any, in the assignment statements given below and rewrite incorrect statements correctly. Assume that the variables are declared as in question 1.

a. `z = sqrt(-fabs(x));`

b. `c = log(-pow(a, b) + asin(x))`

c. `x = ceiling(1.1) + flour(-3.2);`

d. `a = is_upper(ch1);`

e. `ch2 = toupper(ch2);`

f. `a = rnd();`

2. Several function calls are given below. State the return values of these functions. Assume the variables `p`, `q` and `r` (all of type `char`) are initialized as follows: `p = 'A'`, `q = '!'` and `r = '5'`.

- a. `ispunct('+')`
  - b. `isalnum(r)`
  - c. `isalpha(p)`
  - d. `isspace('\t')`
  - e. `isxdigit(p)`
  - f. `iscntrl('\n')`
  - g. `islower(p)`
  - h. `isupper('T')`
  - i. `tolower('a')`
  - j. `toupper('r')`
  - k. `isprint(' ')`
  - l. `ispunct(q)`
3. State the value returned by the following function calls.
- a. `scanf("%d", &x);`
  - b. `printf("%d%d", 10, 20);`
  - c. `ceil(1/3)`
  - d. `sqrt(pow(4, 3))`
  - e. `ceil(sqrt(pow(3,3)))`
  - f. `strlen("Enter a number: \n")`
4. Write a complete C program for the following problems:
- a. Given the basic payment of an employee, calculate the dearness allowance (35% of basic payment), house rent allowance (20% of basic payment) and income tax (20% of total payment excluding H.R.A.). Also calculate the total and net salary. Note that each value must be rounded to the nearest integer.
  - b. A pack of chocolates contains  $c$  chocolates. Determine the minimum number of chocolate packs to be purchased if we have to give  $k$  chocolates to each of  $n$  persons.
  - c. Read names of four countries and prepare a single string by concatenating them. Separate the names by a comma and a space.
5. State the value returned by the following function calls.
- a. `scanf("%d", &x);`
  - b. `printf("%d%d", 10, 20);`

- c. `ceil(1/3)`
  - d. `fabs(-1.234)`
  - e. `ceil(sqrt(pow(3,3)))`
  - f. `strlen("Enter a number: \n")`
  - g. `sin(asin(x))`
  - h. `strcmp("Read", "Ready")`
5. Select the correct option for the following questions.
- a. Which of the following functions is used to test if a given character is alphanumeric?
    - i. `isalnum`
    - ii. `isalnum`
    - iii. `isalphanumeric`
    - iv. `isletterdigit`
  - b. In which of the following header files is the `rand` function declared?
    - i. `stdio.h`
    - ii. `ctype.h`
    - iii. `math.h`
    - iv. `stdlib.h`
  - c. Which of the following functions is provided in C for the concatenation of two strings?
    - i. `strconcat`
    - ii. `strcat`
    - iii. `strcon`
    - iv. `strconcatenate`
  - d. Which of the following is not a standard header file?
    - i. `conio.h`
    - ii. `ctype.h`
    - iii. `stdlib.h`
    - iv. `math.h`
7. Write the output of the following program segments. The data entered from the keyboard is shown in comments.

- a. 

```
int a = 100;
printf("%d", printf("%d", a));
```
- b. 

```
printf("%d", scanf("%d %d", &a, &b));/*10 20 */
```
- c. 

```
char a[20];
strcpy(a, "One");
strcat(a, "Zero");
strcpy(a, "Three");
printf("%d: %s", strlen(a), a);
```
- d. 

```
char c = 'A';
c = tolower(++c+3);
toupper(c++);
printf("%c", c);
```

3. Assume that several strings are initialized as shown below:

```
char name[] = "Pradeep", name2[] = "Leena";
char color1[] = "red", color2[] = "purple";
```

Using string manipulation functions, write the code to prepare the following strings:

"Pradeep likes red color.", "Leena has a purple car."

4. Write equivalent C expressions for the mathematical equations given below. Assume variables **a** and **b** of type **int**.

- a.  $c = \lfloor a/2 \rfloor + \lceil 3b \rceil$
  - b.  $c = \sqrt[3]{|a - b| - |b - 10|}$
  - c.  $c = \sqrt[5]{\frac{\lfloor a/2 \rfloor + \lceil b/3 \rceil}{\lceil ab \rceil - \lceil a/b \rceil}}$
- 

<sup>†</sup> The function declarations or prototypes are provided in header files and are discussed in Section 4.4.2.

# 5 Conditional Control

This chapter presents an important aspect of programming, the *conditional control* (or *selection*). It begins with *relational* and *equality* operators and explains how relational expressions, i. e., conditions are written and evaluated. Then it presents the `if` statement, which in its basic form, is used to select one of two alternative statements for execution. The *logical operators*, which are used to write Boolean expressions, are introduced next and their evaluation is discussed. The `switch` statement, a multi-way selection statement, is presented next. This is followed by *conditional expression operator*, which enables us to write conditional evaluations in a concise way. Finally, the advanced concepts related to the topics covered in this chapter are presented. A beginner may skip this section in the first reading. Numerous examples have been given throughout this chapter.

## 5.1 Relational and Equality Operators

The C language provides four relational and two equality operators for comparing the values of expressions. The **relational operators** are *less than* (`<`), *greater than* (`>`), *less than or equal to* (`<=`) and *greater than or equal to* (`>=`). The **equality operators** are *equal to* (`==`) and *not equal to* (`!=`). These operators are binary infix operators, i. e., they are used in the form  $a \text{ op } b$ , where  $a$  and  $b$  are operands (constants, variables or expressions) and  $\text{op}$  is a relational or an equality operator. Table 5.1 summarizes these operators.

Note that the token for equality operator is `==`. A beginner often makes the mistake of writing the *equal to* operator simply as `=`, which is an assignment operator. Thus, `a == b` means the values of variables `a` and `b` are compared for equality, whereas `a = b` means the value of variable `b` is assigned to variable `a`.

**Table 5.1** Relational and equality operators on the C language

| Category   | Operator | Meaning                  | Example | Associativity |
|------------|----------|--------------------------|---------|---------------|
| Relational | <        | Less than                | a < b   | Left to right |
|            | >        | Greater than             | a > b   |               |
|            | <=       | Less than or equal to    | a <= b  |               |
|            | >=       | Greater than or equal to | a >= b  |               |
| Equality   | ==       | Equal to                 | a == b  |               |
|            | !=       | Not equal to             | a != b  |               |

### 5.1.1 Relational and Equality Expressions

An expression containing relational operators is termed a **relational expression**, whereas an expression containing equality operators is termed an **equality expression**. A simple relational or equality expression takes the following form:

*expr1 op expr2*

where *expr1* and *expr2* are arithmetic expressions and *op* is a relational or equality operator. An expression may contain more than one relational or equality operator, as in

*expr1 op1 expr2 op2 expr3 ...*

and may also contain balanced parentheses. The value of a relational or equality expression is of type **int**. It is equal to 1 if the expression evaluates as true and zero otherwise. We can assign this value to a variable (usually of type **int**).

Note that an arithmetic expression can also be considered as a relational expression. An expression whose value is zero is considered as false, whereas an expression with a non-zero value is considered as true.

#### Example 5.1 Relational and equality expressions

Assume that the symbolic constants and variables are defined as follows

```
#define PI 3.1415927
int i = 1, j = 2;
double x = 5.0, y = 10.0;
char c = 'A', d = 'a';
```

- a) Several examples of simple relational and equality expressions are given below.

| Expression | Value     | Expression | Value     | Expression | Value     |
|------------|-----------|------------|-----------|------------|-----------|
| i < j      | 1 (true)  | x <= y     | 1 (true)  | c >= 'A'   | 1 (true)  |
| 5 >= i     | 1 (true)  | 1.5 >= PI  | 0 (false) | d > 'Z'    | 1 (true)  |
| j == -2    | 0 (false) | y != 10.0  | 0 (false) | c == d     | 0 (false) |

The expressions in the left, middle and right column contain operands of type `int`, `double` and `char`, respectively. The expressions on the first two lines contain relational operators, whereas those on the last line contain equality operators. The operands in these expressions are variables, constants or symbolic constants. The spaces between operators and operands are optional. Thus, the expression `i < j` can also be written as `i<j`. However, it is good programming practice to include spaces between operands and operators.

Note that the comparison of operands of type `char` is done considering their values in the machine's character set (ASCII assumed). The ASCII value of character '`'A'`' is 65. Thus, the value of expression `C >= 'A'` is 1 as this expression means `65 >= 65`, which is true.

**b)** It is not always necessary to know the exact ASCII values of characters, particularly when comparing letters and digits. Just remember that ASCII values of digits are less than the ASCII values of uppercase letters which in turn are less than those of lowercase letters. Also, the digits (0 ... 9), uppercase letters (A ... Z) and lowercase letters (a ... z) are assigned consecutive ASCII values.

It is also possible to compare operands of different arithmetic types. Several examples of such mixed-mode relational and equality expressions are given below along with their values.

| Expression | Value     | Expression | Value    | Expression | Value     |
|------------|-----------|------------|----------|------------|-----------|
| i <= x     | 1 (true)  | x > j      | 1 (true) | d <= y     | 0 (false) |
| j == '\n'  | 0 (false) | y != c     | 1 (true) | c == 65    | 1 (true)  |

As the value of the *newline* character ('`\n`') is 10 (ASCII), expression `j == '\n'` is false and has the value 0. Also, expression `c == 65` is true as the value of variable `c` is '`'A'`', i. e., 65.

Finally note that operator tokens must be written exactly as shown in Table 5.1. Consider the following relational expressions: `i < = j`, `x = = j`, `x => 5`, `y = 5`. The first three expressions are invalid as the operators are written incorrectly (note the spaces in operators `< =` and `= =`). However, the last expression, `y = 5` is a valid assignment expression. It assigns the value 5 to variable `y`. As the value of this expression is 5 (non-zero), it is considered as true and not false as you might think.

**c)** Several mathematical equations are given below along with equivalent C expressions.

| Mathematical expression            | C expression                   |
|------------------------------------|--------------------------------|
| $b^2 - 4ac = 0$                    | $b * b - 4 * a * c == 0$       |
| $n < (a+b)^2$                      | $n < (a + b) * (a + b)$        |
| $\frac{x+y}{a} \geq \frac{x-y}{b}$ | $(x + y) / a \geq (x - y) / b$ |
| $\sqrt{x^2 + y^2} > ab$            | $\sqrt{x * x + y * y} > a * b$ |

As arithmetic operators have higher precedence than the relational and equality operators, the operands of the latter (which are arithmetic expressions) need not be enclosed in parentheses. However, we can use such parentheses to improve the clarity of the expression, as in  $(b * b - 4 * a * c) == 0$ .

### 5.1.2 Evaluation of Relational and Equality Expressions

Table 5.2 summarizes the **precedence** and **associativity** rules of relational and equality operators along with arithmetic and assignment operators. The arithmetic operators (unary, multiplicative and additive) have higher precedence than relational operators, which in turn have higher precedence than equality operators. The assignment operators have the lowest precedence amongst these operators. Thus, while evaluating an expression, arithmetic operators are bound to operands first followed by relational, equality and assignment operators.

**Table 5.2** Precedence and associativity of arithmetic, relational, equality and assignment operators

| Operator Group | Operators        | Associativity |
|----------------|------------------|---------------|
| Unary          | + - ++ --        | Right to left |
| Multiplicative | * / %            |               |
| Additive       | + -              |               |
| Relational     | < > <= >=        | Left to right |
| Equality       | == !=            |               |
| Assignment     | = += -= *= /= %= | Right to left |

All relational operators ( $<$   $>$   $<=$   $>=$ ) have equal precedence. Also, both the equality operators ( $==$   $!=$ ) have equal precedence. Note that the relational and equality operators have left-to-right associativity. Thus, if an expression contains two or more relational operators, they are bound to operands from left to right. The same is true with equality operators.

If an expression contains balanced parentheses, operators within the parentheses are bound first to their operands in the order of their precedence. Also, if an expression contains nested parentheses, the operators in the innermost parentheses are bound first to their operands. When all operators in parenthesized sub-expressions are bound to their operands, the remaining operators in an expression are considered.

## Example 5.2 Evaluation of relational and equality expressions

Assume that variables are declared and initialized as follows:

```
float a = 1.0, b = 2.0, x = 3.0, y = 5.0;
```

Consider expression  $a + b < x * y$ , in which the multiplication operator has highest precedence, followed by the addition operator. These operations are first bound to their operands and then the values of  $a+b$  and  $x*y$  are compared. This is depicted in Fig. 5.1. The order in which operators are bound to operands is shown in Fig. 5.1a and expression evaluation is shown in Fig. 5.1b. The expression is correctly interpreted as  $(a + b) < xy$  and evaluates as true, i. e., 1 for the values given above. We may use parentheses in this expression to improve the readability, as in  $(a + b) < (x * y)$ .

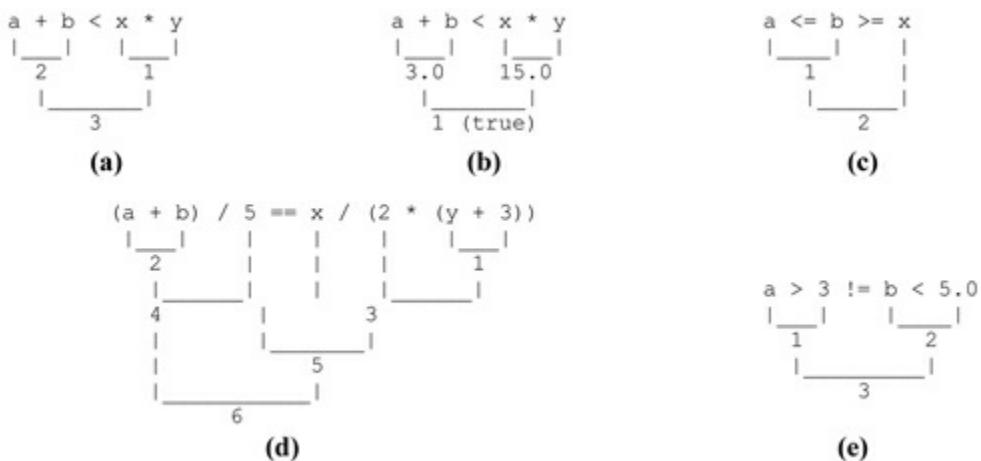


Fig. 5.1 Evaluation of relational expressions

Now consider the assignment statement given below in which the variable **test** is assumed to be of type **int**.

```
test = a + b < x * y;
```

As the assignment operator has least precedence in this statement, the variable **test** is assigned value of relational expression  $a + b < x * y$ , i. e., 1.

Next, consider the evaluation of expression  $a \leq b \geq x$ . Since the relational operators have left-to-right associativity, the  $\leq$  operator is bound first followed by the  $\geq$  operator as shown in Fig. 5.1c. Thus, the expression is equivalent to  $(a \leq b) \geq x$ . The expression  $a \leq x$  evaluates as true, i. e., 1. Now the given expression, which is equivalent to  $1 \geq x$ , evaluates as false, i. e., 0.

The evaluation of a more involved relational expression,  $(a + b) / 5 == x / (2 * (y + 3))$ , is considered next. The order in which the operands are bound to operators is shown in Fig. 5.1d.

It can be easily verified that for the values of the variables given above, this expression evaluates as false, i. e., 0.

Finally, expression  $a > 3 != b < 5.0$ , containing a relational as well as an equality operator is considered. Since the relational operators ( $<$  and  $>$ ) have higher precedence than the equality operator ( $!=$ ), they are first bound to their operands in the left-to-right order. Then the  $!=$  operator is bound to its operands, i. e., the expressions  $a > 3$  and  $b < 5.0$  as shown in Fig. 5.1e. It can be easily verified that the given expression evaluates as true, i. e., 1.

## 5.2 The **if** Statement

The C language provides the **if statement** for the conditional execution of parts of a program. The general form of this statement, which is called the **if-else statement**, is given below.

```
if (expr)
    statement1;
else
    statement2;
```

where *expr* is a valid C expression (such as arithmetic, relational, logical, etc.) and *statement1* and *statement2* are valid C statements. The logical expressions are covered in Section 5.3.

When this statement is executed, either *statement1* or *statement2* (but not both) will be executed depending on the value of *expr*. If the value of *expr* is true (i. e., non-zero), *statement1* is executed; otherwise *statement2* is executed. After the execution of either *statement1* or *statement2*, the next statement in the program (written after this *if-else* statement) is executed. The flowchart for *if-else* statement is given in Fig. 5.2a.

The statements *statement1* and *statement2* may be assignment, function call or block statements. They may even be control statements such as **if**, **switch**, **for**, **while**, **do ... while**, etc. In this section, we consider these statements to be either simple (such as assignment, function call, etc.) or block. If *statement1* and/or *statement2* are *if* statements, we get what is called *nested-if* statement. Such statements are considered in Section 7.1.

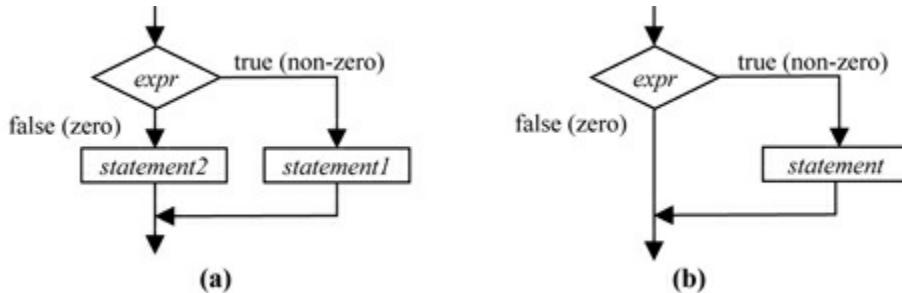
Observe that the statements contained within the *if-else* statement are indented, i. e., shifted right to improve readability. Also, note that these statements should be terminated with a semicolon. However, a semicolon should not be written after the parenthesized test expression and after the **else** keyword. Also, note that if *statement2* is a simple statement, it is often written immediately after the **else** clause to save space.

The **else** part in an *if-else* statement is optional. If it is omitted, the statement takes the following form:

```
if (expr)
```

*statement1*;

This form is called an *if* statement or **simple if statement**. If expression *expr* is true (i. e., non-zero), the contained *statement* is executed followed by the next statement in the program. However, if *expr* is false (i. e., zero), the contained *statement* is not executed and program execution continues with next statement in the program. The flowchart is given in Fig. 5.2b.



**Fig. 5.2** Flowcharts for *if* statements: (a) *if-else*, (b) *simple if*

### Example 5.3 *if-else* and *simple if* statements

Several examples of the *if-else* and *simple if* statements are given below.

#### a) Absolute value of a given number

```
if (n < 0)
    n = -n;
printf("%d", n);
```

These statements determine and print the absolute value of variable *n*. If *n* is negative, then *n < 0* evaluates as *true* and *n* is assigned the value of *-n*, which is then printed in the next statement assuming variable *n* to be of some integer type. However, if the value of *n* is positive, it remains unchanged and is printed. Note that the above *if* statement works with both integral and floating point types. Recall that we can also use the *abs* and *fabs* functions to determine the absolute value of an integer and floating point expression, respectively.

#### b) Print the larger of two numbers

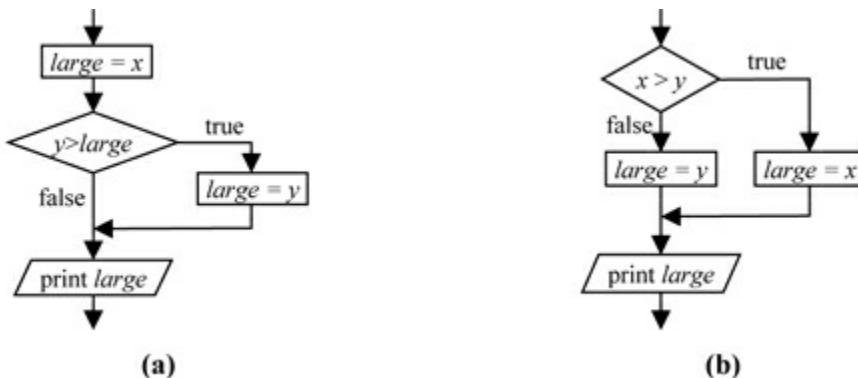
```
large = x;
if (y > large)
    large = y;
printf("Large: %f", large);
```

These statements determine and print the larger of the two variables, *x* and *y*, using a *simple if*

statement. The variables `x`, `y` and `large` are assumed to be of the same type, say `float`.

Initially, variable `x` is assumed to contain a larger value and is assigned to variable `large`. Then, a *simple if* statement compares the value of variable `y` with that of variable `large`. If  $y > \text{large}$  is true, variable `large` is assigned the value of variable `y`; otherwise its value remains unchanged. Thus, `large` contains the value of the larger of the two variables `x` and `y`. Finally, the value of `large` is printed using a `printf` statement. The flowchart for this program segment is given in Fig. 5.3a.

Note that the condition in the *if* statement can also be written equivalently as `large <= y`. However, as the assignment statement `large = y` need not be executed when the value of `large` is equal to `y`, the condition can simply be written as `large < y`.



**Fig. 5.3 Flowcharts to determine the larger of two numbers**

Alternatively, we can use an *if-else* statement to determine the larger of two numbers as follows:

```
if (x > y)
    large = x;
else large = y;
printf("Large: %f", large);
```

Here, the variables `x` and `y` are compared in an *if-else* statement. If condition `x > y` is true, variable `large` is assigned the value of `x`; otherwise it is assigned the value of `y`. The flowchart for this statement is given in Fig. 5.3b.

**c) Print whether an integer number is odd or even**

```
if (n % 2 == 0)
    printf("even");
else printf("odd");
```

This *if-else* statement prints whether an integer number `n` is odd or even. A number is even if it is

divisible by 2. Thus, if expression `n % 2 == 0` is true, number n is even; otherwise it is odd. Alternatively, we can write this `if` statement as follows:

```
if (n % 2 == 1)
    printf("odd");
else printf("even");
```

Note that as the arithmetic expression `n % 2` evaluates as 1 (i. e., true) when n is odd and 0 (i. e., false); otherwise, the equality test in this *if-else* statement is not required. Thus, we can write the test expression in simply as `n % 2`. However, such statements are somewhat difficult to understand and should be avoided, particularly by beginners.

d) Determine the examination result in a single subject

```
if (marks >= 40)
    printf("Pass");
else printf("Fail");
```

This *if-else* statement prints the result of an examination in a single subject, assuming minimum passing marks to be 40. The result is printed as `Pass` if the value of variable `marks` (of type `int`) is greater than or equal to 40; otherwise it is printed as `Fail`.

Alternatively, we can rewrite this `if` statement as

```
if (marks < 40)
    printf("Fail");
else printf("Pass");
```

Now consider the code given below which assigns the result to a string variable named `result` (of type `char[10]`).

```
if (marks >= 40)
    result = "Pass"; /* incorrect */
else result = "Fail"; /* incorrect */
```

This is obviously wrong as we cannot assign strings using the assignment operator. Instead, we have to use the `strcpy` standard library function as shown below.

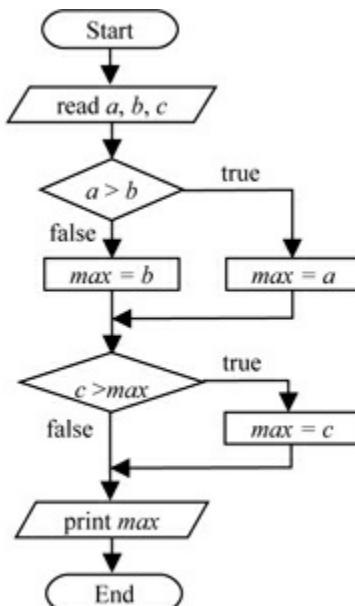
```
if (marks >= 40)
    strcpy(result, "Pass");
else strcpy(result, "Fail");
```

**Program 5.1** Maximum of three numbers

Write a program to determine the maximum of three integer numbers.

**Solution:** Let us use variables **a**, **b** and **c** (all of type **int**) to represent three integer numbers and variable **max** (also of type **int**) to represent the maximum of them.

A flowchart to determine the maximum of three numbers is given in Fig. 5.4. It first accepts the values of variables **a**, **b** and **c**. To determine the maximum number, the larger number of **a** and **b** is first assigned to **max** using the *if-else* statement, as explained in Example 5.3b. Then the larger of variables **max** and **c**, which is the maximum of the three given numbers, is determined using a simple *if* statement. Finally, the value of **max** is printed. The program is given below.



**Fig. 5.4 Flowchart to determine maximum of three numbers**

```
/* Determine maximum of three numbers */
#include <stdio.h>

int main()
{
    int a, b, c;
    int max; /* max of a, b and c */
    printf("Enter three numbers: ");
    scanf("%d %d %d", &a, &b, &c);

    /* determine maximum number */
    if (a > b)
        max = a;
```

```

else max = b;
if (c > max)
    max = c;

printf("Maximum = %d\n", max);

return 0;
}

```

The program output is given below.

```

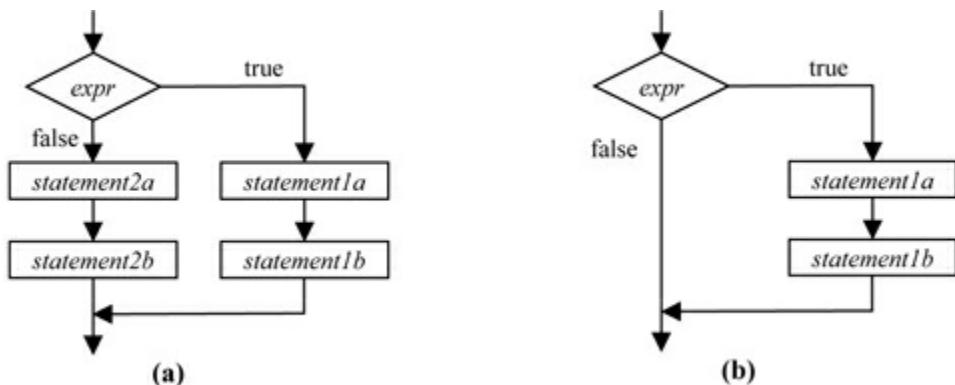
Enter three numbers: 10 15 5
Maximum = 15

```

### 5.2.1 Using Block Statements in the ***if*** Statement

Often we wish to perform more than one operation when the condition in an *if* statement evaluates as true and/or false as shown in Fig. 5.5. In such situations, we use block statements (also called compound statements) within the *if* statement.

When both alternative statements in the general format of an *if-else* statement (i. e., *statement1* and *statement2*) are block statements, the *if-else* statement may be written in one of the styles given below:



**Fig. 5.5** Flowcharts for *if* statements containing more than one statements in if and/or else blocks: (a) *if-else* statement, (b) simple *if* statement

```

if (expr)
{
    statement1a ;
    statement1b ;
    ...
}
else
{
    statement2a ;
    statement2b ;
    ...
}

```

If expression *expr* is true (i. e., non-zero), statements within the *if* block are executed in the given sequence; otherwise statements in the *else* block are executed. The flowchart for this statement is given in Fig. 5.5a. Note that the statements within the *if* and *else* blocks are indented to improve readability. Also, observe the style of using opening braces in a block. In first style, it is written on a line by itself, whereas in second style, it is written on the line containing the **if** and **else** keywords. Both styles are commonly used. The second style is used in this book mainly because it saves space.

If *statement1* or *statement2* (but not both) in the *if-else* statement is a block statement, we get two other forms of *if-else* statements as shown below.

```

if (expr) {
    statement1a ;
    statement1b ;
    ...
}
else
    statement2 ;

```

```

if (expr)
    statement1 ;
else {
    statement2a ;
    statement2b ;
    ...
}

```

Similarly, we can replace *statement1* in a *simple if* statement with a block statement as

```

if (expr) {
    statement1a ;
    statement1b ;
    ...
}

```

The flowchart for this *simple if* statement is given in Fig. 5.5b. Finally note that when the *if* block and *else* block contain a single statement, curly braces are not required. However, programmers often use such braces as shown below:

```

if (expr) {
    statement1 ;
}
else {

```

```
statement2;  
}
```

This form requires more lines, however, it simplifies the addition of one or more statements in *if* and/or *else* blocks and also prevents introduction of difficult to trace bugs when such statements are added without braces.

#### Example 5.4 Using block statements in *if* statements

##### a) Perform a division without causing a *division by zero* error

While performing a division operation, if the divisor is zero, the *division by zero* error occurs and program execution halts abruptly. The program segment given below calculates and prints the value of  $a/b$  without causing a *division by zero* error.

```
if (b == 0)  
    printf("Division by zero\n");  
else {  
    z = a / b;  
    printf("%f", z);  
}
```

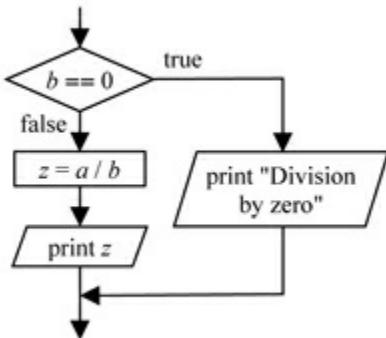
The *if-else* statement tests a value of variable *b* and if it is equal to 0, a message *Division by zero* is printed (and program execution continues); otherwise the value of  $a/b$  is calculated and printed. The flowchart for this statement is given in Fig. 5.6.

##### b) Calculate simple or compound interest on deposit

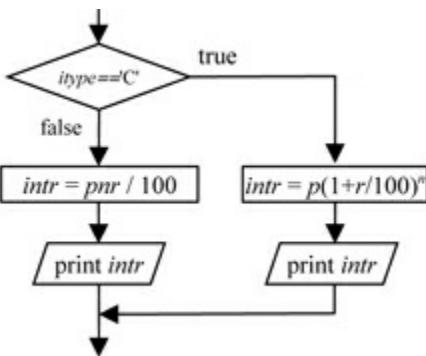
If the amount deposited in a bank (i. e., principal), number of years and interest rate are denoted as *p*, *n* and *r*, respectively, simple interest on deposit is calculated as  $pn\ r/100$  and compound interest (compounded annually) as  $p(1 + r/100)^n$ . The program segment given below calculates and prints either simple or compound interest on deposit.

```
scanf("%f %f %d %c", &p, &r, &n, &itype);  
if (itype == 'C') {  
    /* calculate compound interest - compounded annually */  
    intr = p * pow((1 + r / 100), n);  
    printf("Compound interest = %6.2f\n", intr);  
}  
else {  
    /* calculate simple interest */  
    intr = p * n * r / 100.0;  
    printf("Simple interest = %6.2f\n", intr);  
}
```

}



**Fig. 5.6** Flowchart to calculate value of  $a/b$



**Fig. 5.7** Flowchart to calculate simple and compound interest

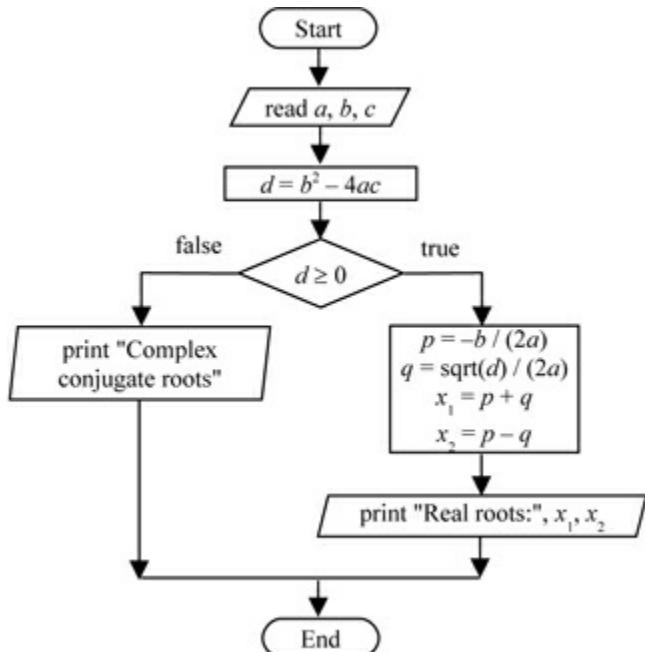
The `scanf` statement reads the amount deposited (`p` of type `float`), interest rate (`r` of type `float`), number of years (`n` of type `int`) and type of interest (`itype` of type `char`). Then an *if-else* statement is used to calculate and print interest. If the value of `itype` is 'C' (uppercase), compound interest is calculated and printed in the *if* block; otherwise, simple interest is calculated and printed in the *else* block. The flowchart is given in Fig. 5.7.

### Program 5.2 Real roots of the quadratic equation $ax^2 + bx + c = 0$

Write a program to accept the coefficients of a quadratic equation  $ax^2 + bx + c = 0$  and print its roots if they are real. However, if the roots are complex conjugate, print an appropriate message.

**Solution:** The roots of a quadratic equation are real if *discriminator*  $b^2 - 4ac$  is greater than or equal to zero; otherwise they are complex conjugate. A program to determine roots of above quadratic equation was developed in Program 4.6. It gives the correct solution for quadratic equation having real roots. However, if the roots are complex conjugate, i. e., if  $b^2 - 4ac < 0$ , the evaluation of expression `sqrt(b*b-4*a*c)` causes a *domain error* and the program halts abruptly.

In this example, we modify that program to print the appropriate error message when the roots are complex conjugate, thereby avoiding a *domain error*. The flowchart in Fig. 5.8 first accepts coefficients of a quadratic equation in variables **a**, **b** and **c** of type **float** and calculates the discriminator (**d**). Then an *if-else* construct is used to test the value of the discriminator. If it is greater than or equal to zero, real roots are calculated and printed; otherwise the message *Complex conjugate roots* is printed. The program is given below.



**Fig. 5.8** Flowchart to determine real roots of a quadratic equation

```

/* Determine real roots of a quadratic equation */
#include <stdio.h>
#include <math.h>
int main()
{
    float      a, b, c;          /* coefficients of quadratic equation */
    float      d;                /* discriminator */
    float      p, q;              /* intermediate variables */
    float      x1, x2;            /* real roots */

    printf("Enter coefficients of quadratic equation: ");
    scanf("%f %f %f", &a, &b, &c); /* read coefficients */

    d = b * b - 4.0 * a * c;
    if (d >= 0) {               /* roots are real */
        /* calculate intermediate variables */
        /* calculate real roots */
        /* print real roots */
    }
    else {
        /* print complex conjugate roots */
    }
}
  
```

```

    p = -b / (2.0 * a);
    q = sqrt(d) / (2.0 * a);
    x1 = p + q;
    x2 = p - q;
    printf("Real roots: x1 = %4.2f x2 = %4.2f\n", x1, x2);
}
else printf("Complex conjugate roots\n");

return 0;
}

```

The program is executed twice and the output given below.

```

Enter coefficients of quadratic equation: 1 -3 2
Real roots: x1 = 2.00 x2 = 1.00

```

```

Enter coefficients of quadratic equation: 1 2 3
Complex conjugate roots

```

Note that in the above program, variables `p`, `q`, `x1` and `x2` are used only within the `if` block. Thus, they can be declared in that block instead of the `main` function. Such *declarations at the point of use* reduce the number of variables declared at the beginning of the main function and make the program easier to understand. Moreover, these variables cannot be used outside the `if` block in which they are declared. This prevents errors caused by their misuse in other parts of the `main` function. We can also initialize these variables with appropriate expressions as shown below.

```

if (d >= 0) {
    float      p = -b /(2.0*a);
    float      q = sqrt(d)/(2.0 * a);
    float      x1 = p +q;
    float      x2 = p -q;
    printf("Roots are x1 = %4.2f x2 = %4.2f\n", x1, x2);
}
else printf("Complex conjugate roots\n");

```

However, remember that all declaration and initialization statements must be written before any executable statement (such as assignment, function call, etc.).

## 5.3 Logical Operators

The C language provides three **logical operators** that can be used to join relational and equality expressions and form complex **Boolean expressions**, i. e., expressions with operands having *true* or

*false* values. These operators include *logical AND* (`&&`), *logical OR* (`||`) and *logical NOT* (`!`). They are summarized in Table 5.3.

Note that the logical not (`!`) is a unary prefix operator. It is used, as in `! expr`, where `expr` is a relational expression. The other two operators are binary infix operators. They are used, as in `expr1 op expr2`, where `expr1` and `expr2` are relational expressions.

**Table 5.3 Logical operators in the C language**

| Operator                | Meaning     | Example                                           | Associativity |
|-------------------------|-------------|---------------------------------------------------|---------------|
| <code>!</code>          | Logical not | <code>! (a &lt; 0)</code>                         | Right to left |
| <code>&amp;&amp;</code> | Logical and | <code>(a &gt;= 0) &amp;&amp; (a &lt;= 100)</code> |               |
| <code>  </code>         | Logical or  | <code>(a &lt; 0)    (a &gt; 100)</code>           | Left to right |

### 5.3.1 Logical AND Operator

As the **logical AND** (`&&`) operator is a binary infix operator, it is used, as in

`expr1 && expr2`

where `expr1` and `expr2` are relational expressions. This expression evaluates as true, i. e., 1 if both `expr1` and `expr2` are true; otherwise, it evaluates as false. Consider the expression

`(a >= 0) && (a <= 100)`

This expression evaluates as true if both the relational expressions `(a >= 0)` and `(a <= 100)` are true, i. e., if the value of variable `a` is in the range 0 to 100 (both inclusive); otherwise, it evaluates as false, i. e., 0.

### 5.3.2 Logical OR Operator

The **logical or** (`||`) operator is also a binary infix operator. It is used, as in

`expr1 || expr2`

where `expr1` and `expr2` are relational expressions. This expression evaluates as true, i. e., 1 if either `expr1` or `expr2` is true (or both of them are true); otherwise, it evaluates as false. Consider the expression

`(a < 0) || (a > 100)`

This expression evaluates as true if the value of variable `a` is less than zero or greater than 100, i. e., outside the range 0 to 100; otherwise, it evaluates as false, i. e., 0.

### 5.3.3 Logical NOT Operator

The **logical not** (!) operator performs logical complement operation. This is a unary prefix operator and is used, as in

`! expr`

This expression evaluates as true if expression *expr* is false, and false otherwise. Consider the expression

`! (a < 0)`

If the value of variable *a* is less than zero, this expression evaluates as false; otherwise, it evaluates as true. Note that this expression is equivalent to the relational expression *a*  $\geq 0$ . Consider another expression given below.

`! (( a < 0) || ( a > 100))`

If the value of variable *a* is outside 0 to 100, the expression `(a < 0) || (a > 100)` evaluates as true and the given expression evaluates as false. On the other hand, if the value of *a* is in the range 0 ... 100, the above expression evaluates as true. Thus, the above expression is equivalent to the expression `(a >= 0) && (a <= 100)`.

#### 5.3.4 Evaluation of Boolean Expressions

Table 5.4 summarizes the **precedence** and **associativity** of logical operators along with the other operators including arithmetic, relational, equality and assignment operators. We know that unary operators have the highest precedence followed by arithmetic (multiplicative and additive), relational and equality operators.

As *logical not* (!) is a unary operator, it has highest precedence (same as other unary operators) and right-to-left associativity. The precedence of the logical operators `&&` and `||` is lower than the arithmetic, relational and equality operators but higher than the assignment operators. Note that the *logical and* (`&&`) operator has higher precedence than the *logical or* (`||`) operator. Also, note that `&&` and `||` operators have left-to-right associativity.

While evaluating Boolean expressions, the operators are bound to operands in the order of their precedence and associativity as shown in Table 5.4. Balanced parentheses can be used to alter the order in which the operands are bound to operands.

**Table 5.4** Precedence and associativity of arithmetic, relational, equality, logical, conditional evaluation, assignment and comma operators in the C language

| Operator group         | Operators          | Associativity |
|------------------------|--------------------|---------------|
| Unary                  | ! + - ++ -- (type) | Right to left |
| Multiplicative         | * / %              |               |
| Additive               | + -                |               |
| Relational             | < > <= >=          | Left to right |
| Equality               | == !=              |               |
| Logical and            | &&                 |               |
| Logical or             |                    |               |
| Conditional Expression | ? :                |               |
| Assignment             | = += -= *= /= %=   | Right to left |
| Comma                  | ,                  | Left to right |

We have seen earlier that the C language does not specify the order of evaluation of expressions. However, the logical and (`&&`) and logical not (`||`) operators are exception to this rule and are always evaluated left-to-right.

The evaluation of an expression of the form `expr1 && expr2` is done as follows: Initially expression `expr1` is evaluated. If it evaluates as false, the entire expression will evaluate as false irrespective of the value of expression `expr2`. Hence, expression `expr2` is not evaluated. This is sometimes referred to as *short-circuit evaluation*. However, if expression `expr1` evaluates as true, the value of the entire expression depends on the value of expression `expr2`. Hence, expression `expr2` is evaluated next and the value of the entire expression is determined. Note that in `expr1 && expr2`, `expr1` is guaranteed to evaluate before `expr2`.

The evaluation of an expression of the form `expr1 || expr2` is also done in a similar way: Initially expression `expr1` is evaluated. If it evaluates as true, the entire expression will evaluate as true irrespective of the value of expression `expr2`. Hence, expression `expr2` is not evaluated. However, if expression `expr1` evaluates as false, the value of the entire expression depends on the value of `expr2`. Hence, `expr2` is evaluated next and the value of the entire expression is determined. Note that in `expr1 || expr2`, `expr1` is guaranteed to evaluate before `expr2`.

### Example 5.5 Evaluation of Boolean expressions

Assume that variables `a`, `b` and `ch` are declared and initialized as shown below.

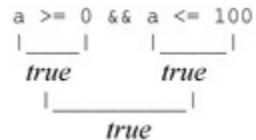
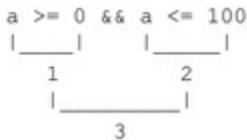
```
int a = 3, b = 5;
char ch = 'G';
```

Consider a simple Boolean expression given below.

```
a >= 0 && a <= 10
```

In this expression, the operators `>=` and `<=` have higher precedence than operator `&&` and are bound

to operands from left to right. Subsequently, the operator `&&` is bound to its operands, i.e., relational expressions `a >= 0` and `a <= 10`. During the evaluation of this expression, sub-expression `a >= 0` is evaluated first as true. Now the value of the given expression depends on sub-expression `a <= 10` and it is evaluated next. Since it evaluates as true, the value of the given expression is true. The order of operator binding and evaluation is shown below.

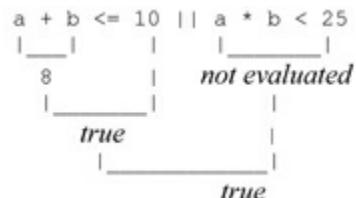
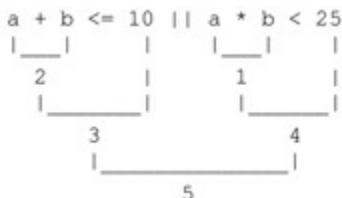


Note that we can make the given expression more readable by including redundant parentheses as `(a >= 0) && (a <= 100)`.

Now consider another Boolean expression given below.

`a + b <= 10 || a * b < 25`

This expression contains two relational expressions joined by the `||` operator. The operators in this expression are bound to operands in order shown below on the left-hand side. During evaluation, sub-expression `a + b <= 10` is evaluated first as true. Now the value of the given expression is true irrespective of the value of sub-expression `a * b < 25`. Hence, it is not evaluated.

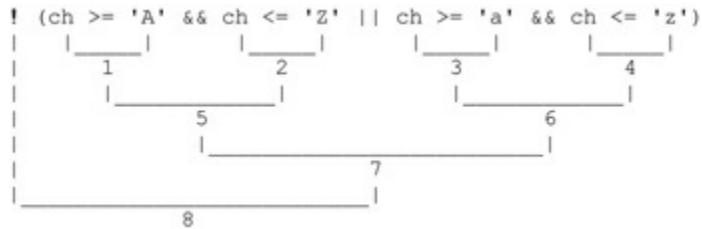


Note that the readability of the given expression can be improved by adding redundant parenthesis as `(a + b <= 10) || (a * b < 25)`.

Finally, consider a more involved Boolean expression given below.

`! (ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z')`

The order in which operands in this expression are bound to operands is shown below. Note that all the operators in the pair of parentheses are considered first followed by the `!` operator.



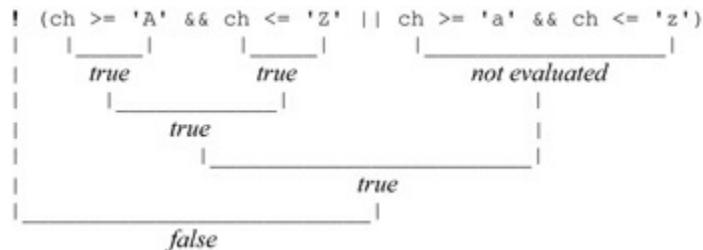
As the `&&` operator has higher precedence than the `||` operator, the given expression is equivalent to the following parenthesized expression:

```
! ( (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') )
```

Let us denote the left operand of the `||` operator (i. e., `ch >= 'A' && ch <= 'Z'`) as *op1* and the right operand as *op2*. Thus, the given expression is of the form `!(op1 || op2)`.

The evaluation of this expression proceeds as follows: First, operand *op1* is evaluated. If the left operand of the `&&` operator in *op1*, i. e., `ch >= 'A'` evaluates as false, the value of *op1* is false (right operand of the `&&` operator is not evaluated); otherwise, the right operand of the `&&` operator (i. e., `ch <= 'Z'`) is evaluated to determine the value of *op1*.

Next, if the value of expression *op1* is true, the value of the `||` operation is true (*op2* is not evaluated). Otherwise, the right hand side operand of the `||` operator, i. e., *op2* is evaluated to determine the outcome of the `||` operator. Finally, the logical not (`!`) operator is evaluated. The evaluation of the complete expression is shown below for a given value of `ch`, i. e., '`G`'.



Observe that *op1* evaluates as true if variable `ch` contains an uppercase letter and *op2* evaluates as true if `ch` contains a lowercase letter. Thus, the expression within the pair of parentheses evaluates as true if variable `ch` contains a letter (either uppercase or lowercase). The given expression thus, evaluates as false if variable `ch` contains a letter; otherwise, it evaluates as true. Thus, for the given value of variable `ch`, the expression evaluates as false.

### Example 5.6 Using logical operators

- a) Test validity of marks in a single subject

```
if (marks >= 0 && marks <= 100)
    printf("Valid marks");
else printf("Invalid marks");
```

This program segment uses an *if-else* statement to test whether the marks entered are valid or not and print the appropriate message. It is assumed that for **marks** to be valid, it should be in the range 0 to 100, i. e., greater than or equal to zero *and* less than or equal to 100.

We may rewrite the above code by using a test for invalid marks. The value of **marks** is invalid if it is outside the range 0 to 100, i. e., less than 0 *or* greater than 100. The **if-else** statement using such a test is given below.

```
if (marks < 0 || marks > 100)
    printf("Invalid marks");
else printf("Valid marks");
```

Note that there is also another way to express the validity of marks: it should not be invalid, i. e., it should not be outside the range 0 to 100. This logic is used in the **if** statement given below.

```
if (!(marks < 0 || marks > 100))
    printf("Valid marks");
else printf("Invalid marks");
```

### b) Determine whether a given year is leap or not

A year is leap if it is divisible by 4 but not by 100 or is divisible by 400. Thus, 1996, 2000 and 2004 are leap years but 1900, 2002 and 2100 are not. The program segment given below determines whether a given year (of type **int**) is *leap* or not.

```
if (year % 4 == 0 && year % 100 != 0 /* divisible by 4, not by 100 */
    || year % 400 == 0) /* or divisible by 400 */
    printf("%d is a leap year\n", year);
else printf("%d is not a leap year\n", year);
```

As we can assign the value of a Boolean expression to an integer variable, this program segment can also be written as follows, where **is\_leap** is assumed to be a variable of type **int**.

```
is_leap = (year % 4 == 0 && year % 100 != 0 || year % 400 == 0);
if (is_leap)
    printf("%d is a leap year\n", year);
else printf("%d is not a leap year\n", year);
```

### c) Result of a student in SSC examination

Consider that we wish to determine the result of a student in the SSC (Secondary School Certificate) examination, which has six subjects: four of 100 marks each and two of 150 marks each. For a student to pass the examination, he/she has to pass each subject, where the pass percentage is 35. Let us use variables `m1`, `m2`, `m3`, `m4`, `m5` and `m6` for marks in these subjects where the first four variables (`m1` to `m4`) represent marks in subjects having maximum 100 marks. The if statement given below determines the result of a student.

```
if (m1 >= 35 && m2 >= 35 && m3 >= 35 && m4 >= 35 && m5 >= 52 && m6 >= 52)
    printf("Pass\n");
else printf("Fail\n");
```

Alternatively, a student fails if he fails in any one of the subjects. Thus, the above `if` statement can be written as shown below.

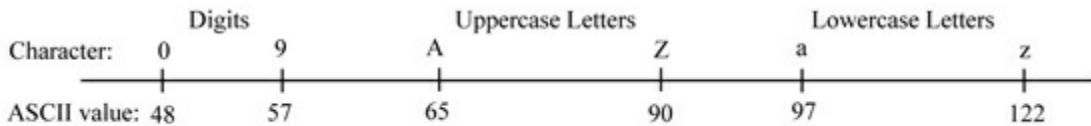
```
if (m1 < 35 || m2 < 35 || m3 < 35 || m4 < 35 || m5 < 52 || m6 < 52)
    printf("Fail\n");
else printf("Pass\n");
```

## 5.4 Character Test, Classification and Conversion

The characters in a machine's character set can be grouped in various categories such as letters (uppercase and/or lowercase), digits, alphanumeric characters, white space, punctuation, etc. The standard C library provides the `ctype.h` header file that includes several functions for character test, classification and conversion such as `isupper`, `islower`, `isalpha`, `isdigit`, `toupper`, `tolower`, etc. These functions are discussed in Section 4.4.4.

In this section, we study how we can perform such operations on characters by writing our own code instead of using library functions. The ASCII character set is assumed. In this character set, the codes for digits (0 to 9), uppercase letters (A to Z) and lowercase letters (a to z) are contiguous as shown in Fig. 5.9 along with the ASCII code values. However, this is not true in some character sets such as EBCDIC which is rarely used. Also, note that the ASCII codes for digits are smaller than the ASCII codes of uppercase letters which in turn are smaller than the codes for lowercase letters.

Note that while comparing character values, we should use the character constants ('A', 'Z', '0', '9', etc.) rather than the ASCII values. For example, to test whether the value of variable `ch` (of type `char`) is greater than or equal to uppercase 'A' (whose ASCII value is 65), we should use the expression `ch >= 'A'` rather than the expression `ch >= 65`. Direct use of ASCII values in the code makes it difficult to understand. Moreover, this code may not work correctly with machines using other character sets (such as EBCDIC), i. e., the program may not be portable.



**Fig. 5.9 Alphanumeric characters in the ASCII code**

**Fig. 5.9 Alphanumeric characters in the ASCII code**

As the codes for uppercase letters are contiguous, a given character is an uppercase letter if it is greater than or equal to 'A' and less than or equal to 'Z'. Similarly, a character is a lowercase letter if its value is in the range 'a' to 'Z' (both inclusive) and a character is a digit if its value is in the range '0' to '9' (both inclusive).

Also, observe that the difference between the ASCII values of any lowercase letter and the corresponding uppercase letter is 32. Thus, to convert a lowercase letter to its uppercase representation, we can subtract 32 from its value. Similarly, to convert an uppercase letter to its lowercase representation, we can add 32 to its value. Again, the use of literal 32 may lead to non-portable programs. Hence, we can add or subtract the difference in the uppercase and lowercase representation of any particular character, e.g., 'a' - 'A', 'b' - 'B', etc.

#### Example 5.7 Character test, classification and conversion

Several examples of character test, classification and conversion are given below. In these examples, the variable ch is assumed to be of type char.

##### a) Test for uppercase letter

A character is an uppercase letter if its value in the range 'A' to 'Z' (both inclusive). The program segment given below tests whether a given character is an uppercase letter or not.

```
if (ch >= 'A' && ch <= 'Z')
    printf("%c is uppercase letter\n", ch);
else printf("%c is not uppercase letter\n", ch);
```

This example uses an **if-else** statement to determine whether the character in variable ch is an uppercase letter or not. Note that if the condition in the **if** statement evaluates as false, we cannot assume the character to be a lowercase letter as it may be any other character such as a digit, punctuation symbol, space, etc.

We can rewrite the condition in the above **if** statement using the values of character constants 'A' and 'Z' as (ch >= 65 && ch <= 90). However, this should be avoided for the reasons mentioned above. Also, recall that we can use the **isupper** library function to perform the test for uppercase characters and rewrite the if statement as **if (isupper(ch))**.

### b) Test for whitespace

A character is a whitespace if it is one of the following characters: blank space (' '), newline ('\n'), horizontal tab ('\t'), carriage return ('\r'), form feed ('\f') or vertical tab ('\v'). The **if** statement given below tests whether the given character **ch** is equal to one of these characters using logical or (||) operator and prints an appropriate message.

```
if (ch == ' ' || /* ch is a blank space or */
    ch == '\t' || /* a horizontal tab or */
    ch == '\n' || /* a new line or */
    ch == '\r' || /* a carriage return or */
    ch == '\f' || /* a form feed or */
    ch == '\v') /* a vertical tab */
    printf("Whitespace\n");
else printf("Not whitespace\n");
```

Note that to test for a whitespace, it is often sufficient to check whether the given character is equal to one of three characters: space, horizontal tab and new line, e. g., when the characters are entered from the keyboard. Thus, the above **if** statement can be rewritten as shown below.

```
if (ch == ' ' || ch == '\t' || ch == '\n')
    printf("Whitespace\n");
else printf("Not whitespace\n");
```

### c) Test for alphanumeric character

An alphanumeric character is one that is either an uppercase or lowercase letter or a digit. The statement given below tests whether character **ch** is an alphanumeric character or not.

```
if (ch >= 'A' && ch <= 'Z' || /*ch is an uppercase letter or */
    ch >= 'a' && ch <= 'z' || /* a lowercase letter or */
    ch >= '0' && ch <= '9') /*a digit */
    printf("%c is alpha-numeric character\n", ch);
else printf("%c is not alpha-numeric character\n", ch);
```

### d) Character case conversion

The program segment given below converts an uppercase letter to lowercase.

```
if (ch >= 'A' && ch <= 'Z')
    ch += 32;
```

As the C language allows arithmetic operations on characters, we can use expression such as '**a**' -

'A', 'g' - 'G', etc. instead of literal 32. To convert a lowercase letter to uppercase, we should subtract either 32 or the difference 'a' - 'A' from it as shown below.

```
if (ch >= 'a' && ch <= 'z')
    ch -= 'a' - 'A'; /* ch -= 32; */
```

#### e) Test for a vowel

A given character is a vowel if it is one of the following characters: A, E, I, O and U. The **if-else** statement given below tests whether a given character is a vowel or not.

```
if (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U' ||
    ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u')
    printf("%c is a vowel\n", ch);
else printf("%c is not a vowel\n", ch);
```

Note that we have to compare variable **ch** with uppercase as well as lowercase letters. We can reduce the number of comparisons in the above code if we first convert the given character to uppercase (or lowercase) representation. Thus, the condition in the **if** statement involves comparison with uppercase (or lowercase) letters only. The code using this approach is given below.

```
/* convert ch to uppercase */
if (ch >= 'a' && ch <= 'z')
    ch -= 'a' - 'A';

/* test for vowel (uppercase only) */
if (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U')
    printf("%c is vowel\n", ch);
else printf("%c is not a vowel\n", ch);
```

## 5.5 The **Switch** Statement

The **switch statement** allows us to test an expression and depending on its value, execute a statement (or group of statements) amongst several alternatives. It is the most involved statement provided in C language. It uses four keywords, namely, **switch**, **case**, **default** and **break**. The last two keywords are optional and can be omitted. The general form of this statement is given below and its flowchart is shown in Fig. 5.10.

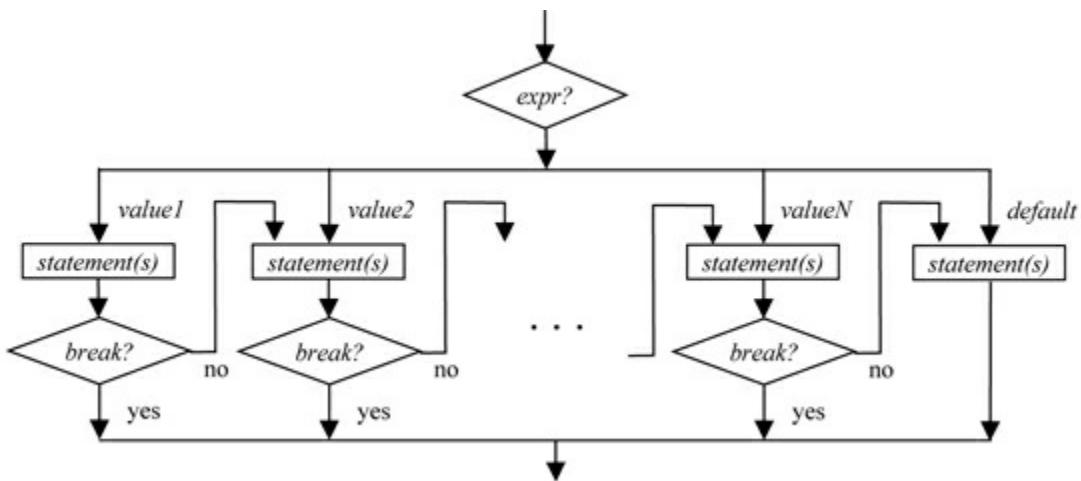
```
switch (expr)
{
    case value1:
        statement;
```

```

statement;
...
break;
case value2:
    statement;
    statement;
    ...
break;
...
default:
    statement;
    statement;
    ...
break;
}

```

The test expression *expr* is written within parentheses after the **switch** keyword. The value of this expression is used to select one of the alternative groups of statements (cases), written using the **case** and **default** keywords, for execution. Note that the test expression must be an *integral* expression. A beginner often makes the mistake of using either a floating or a string expression as a test expression in a **switch** statement.



**Fig. 5.10** Flowchart for the **switch** statement (The default group is assumed to be at the end of **switch** statement)

The body of the **switch** statement consists of one or more alternative groups of statements each preceded by a line of the form

**case** *value*:

Only one value can be written after each **case** keyword and it must be a *constant integral* expression. Thus, we can use integer and character constants as case values. However, floating and string constants are not permitted. Also, we cannot write non-constant expressions, numbers separated by commas or spaces (such as 1, 2, 3) and ranges (such as (0 ... 100, 'A' ... 'Z', etc.) as case values.

Each alternative statement group may contain one or more statements. We can include any valid statement in this group. Each alternative statement group is usually followed by an optional **break** statement. The statements in a particular group are selected for execution if the **case** value specified before that group equals the value of the test expression *expr*.

Note that the **switch** statement may also contain one group of alternative statements preceded by a line of the form

### **default:**

This statement group is optional. However, if specified, the statement(s) within this default group are executed if the value of *expr* is not found in any of the **case** clauses. Although we can write this group anywhere in a **switch** statement, it is usually written as the last group.

The execution of the **switch** statement proceeds as follows: Initially, expression *expr* is evaluated and its value is compared with the **case** values *value1*, *value2*, ... *valueN* in the order in which they are specified in the **switch** statement. If the value of expression *expr* matches a **case** value, the corresponding statement group is executed. If a **break** statement is encountered during this, the control leaves the **switch** statement; otherwise, the control falls through the next case and executes the statements in that case.

Note that if the value of *expr* does not match any of the **case** values specified in the **switch** statement, the statement group associated with the **default** case is executed if it is present; otherwise, the control leaves the **switch** statement without executing any statement.

Although individual **case** values and **default** group can be written in any order, it is a good practice to mention **case** values in some order and the **default** at the end of a **switch** statement. It is also a good programming practice to write a **break** statement after the last group of alternative statements. Although not really required, it prevents the introduction of a bug in the program when we add more cases at the end of the **switch** statement.

Finally, note that we can associate a group of statements with more than one **case** values by listing these **case** values before that statement group as shown below:

```
case value1:  
case value2:  
...  
case valueK:  
    statement;  
    statement;
```

```
...  
break;
```

These statements will be executed whenever *expr* evaluates to any one of the values *value1*, *value2*, ..., *valueK*. Note that this behavior of the **switch** statement is possible because in the absence of a **break** statement, the control falls through to the next case.

#### Example 5.8 Examples of the **switch** statement

##### a) Print name of color (Red, Green or Blue) that starts with a given character

The program segment given below accepts a character from the keyboard and prints the name of the corresponding color, e. g., if the user enters character R, it prints Red. However, it handles only three colors, namely, red, green and blue.

```
printf("Enter character (R/G/B): ");  
color = getchar();  
  
switch (color) {  
    case 'R':  
        printf("Red");  
        break;  
    case 'G':  
        printf("Green");  
        break;  
    case 'B':  
        printf("Blue");  
        break;  
}
```

The character entered from the keyboard is first accepted in variable **color** (of type **char**) and is then tested in the **switch** statement. The program includes three cases corresponding to the colors red, green and blue. Note that the case values are character constants 'R', 'G' and 'B'. Each **case** includes a **printf** statement to print the name of the color followed by a **break** statement. The output of program containing the above code is shown below.

```
Enter character (R/G/B): G  
Green
```

##### b) Print a given digit as a word

The program segment given below accepts an integer number in variable **digit** of type **int** from the keyboard and if it is a single digit number (0-9), it displays the word corresponding to it; otherwise, it

displays the message *Not a digit*. The **switch** statement has a **case** block for each digit (0-9) and a **default** block to handle all other numbers. Note that the code for each case has been written on a single line to save space.

```
printf("Enter a digit (0-9): ");
scanf("%d", &digit);
switch(digit) {
    case 0: printf("Zero"); break;
    case 1: printf("One"); break;
    case 2: printf("Two"); break;
    case 3: printf("Three"); break;
    case 4: printf("Four"); break;
    case 5: printf("Five"); break;
    case 6: printf("Six"); break;
    case 7: printf("Seven"); break;
    case 8: printf("Eight"); break;
    case 9: printf("Nine"); break;
    default: printf("Not a digit"); break;
}
printf("\n");
```

### c) Print remaining months in a year

```
switch(month) {
    default:
        case 1 : printf("Jan ");
        case 2 : printf("Feb ");
        case 3 : printf("Mar ");
        case 4 : printf("Apr ");
        case 5 : printf("May ");
        case 6 : printf("Jun ");
        case 7 : printf("Jul ");
        case 8 : printf("Aug ");
        case 9 : printf("Sep ");
        case 10: printf("Oct ");
        case 11: printf("Nov ");
        case 12: printf("Dec ");
```

}

This program segment prints the short names of all remaining months starting from a given month. This is achieved by omitting the **break** statement in all the cases. Also, note that the **default** case has been mentioned in the very beginning. Thus, if the user enters an invalid value for a month (any value other than 1 to 12), the program prints the names of all the months. The program containing this

code segment is executed twice and the output is given below:

Enter month number: 8

Aug Sep Oct Nov Dec

Enter month number: 15

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

### Program 5.3 Simple calculator

Write a program to accept an expression of the form  $a \text{ op } b$  from the keyboard and display its value on the screen. Assume that  $\text{op}$  may be one of the following operators:  $+$ ,  $-$ ,  $*$ ,  $\times$ ,  $X$  and  $/$  where letters  $x$  and  $X$  indicate multiplication operation.

**Solution:** Let us use variables  $a$  and  $b$  of type `float` to represent two operands, variable  $c$  of type `float` to represent the result and variable  $\text{op}$  of type `char` to represent the operator. In the program given below, an expression of the form  $a \text{ op } b$  is first accepted from the keyboard and a `switch` statement is used to evaluate the result. The operator `op` is used as test expression in the `switch` statement and four cases correspond to the given operators. In each case, the desired value is evaluated in variable  $c$ . This value is printed using a `printf` statement after the `switch` statement.

```
/* Simple calculator: evaluates expression of form a op b */
#include <stdio.h>

int main()
{
    float a, b, c; /* operands */
    char op; /* operator: + - * or / */
    printf("Enter expression of the form a op b:\n");
    scanf("%f %1c %f", &a, &op, &b);

    switch(op) {
        case '+':
            c = a + b;
            break;

        case '-':
            c = a - b;

        break;
        case '*':
        case 'x':
        case 'X':
            c = a * b;
    }
}
```

```

        break;

    case '/':
        c = a / b;
        break;
}
printf("c = %f\n", c);
return 0;
}

```

Observe that the case values are character constants: '+', '-', '\*', 'x', 'X' and '/'. Also, an operator is accepted from the keyboard using the `%1C` format in the `scanf` statement.

## 5.6 Conditional Expression Operator (? :)

The **conditional expression operator** (`? :`) is the only ternary operator in the C language. It takes three operands and is used to evaluate one of the two alternative expressions depending on the outcome of a test expression as shown below.

*test\_expr ? expr1 : expr2*

Here, *test\_expr* is the test expression and *expr1* and *expr2* are the alternative expressions, only one of which is evaluated, depending on the outcome of *test\_expr*. Observe how the question mark (?) and the colon (:) separate these expressions. If *test\_expr* evaluates as true (i. e., non-zero), the value of the entire expression is equal to the value of *expr1*; otherwise, it is equal to the value of *expr2*. Consider, for example, the expression `a < b ? a : b`. The value of this expression is either *a* or *b*, depending on whether the expression is true or false, respectively. Thus, the expression determines the smaller of the two numbers *a* and *b*.

The contained expressions may be of any such as arithmetic, relational, Boolean, assignment, etc. They may also be function calls or involved expressions containing function calls. We can also use a comma operator to write multiple expressions in place of sub-expressions. The conditional expression can be used just like any other expression. Thus, we can use it as a part of more involved expressions, assign its value to a variable or use it as an argument in a `printf` or other function call.

Observe in Table 5.4 that the conditional expression operator has right-to-left associativity and lower precedence than all other operators except the assignment and comma operators. Thus, while writing conditional expressions, the parentheses surrounding the contained expressions are often unnecessary unless we use assignment or comma operators in those expressions. However, to improve the readability, we can include complex expressions within parentheses as shown below.

`( test_expr ) ? ( expr1 ) : ( expr2 )`

The conditional expression is a powerful facility that allows us to write very concise code. It is very similar to the `if-else` statement and we can often replace it. However, there are some differences:

1. The **if - else** statement is more readable than the conditional expression but requires more space.
2. The *if* and *else* blocks can be any valid C statement including any control statement or a compound statement that may contain hundreds of statements. However, we cannot use any C statement or a block in a conditional expression. Moreover, to keep the code readable, expressions *expr1* and *expr2* in a conditional expression are usually simple.
3. The **else** clause in an **if** statement can be omitted. This is not possible in conditional expressions.

#### Example 5.9 Using the conditional expression operator

##### a) Larger of two numbers

```
large = a > b ? a : b;
```

This example illustrates how a conditional expression can be used to assign one of two alternative values to a variable depending on the outcome of some condition. As the conditional expression operator has higher precedence than the assignment operator, the above assignment statement is equivalent to

```
large = (a > b ? a : b);
```

The value of the conditional expression (*a* > *b* ? *a* : *b*) is equal to that of variable *a* if *a* is greater than *b*; otherwise, it is equal to the value of variable *b*. This value is assigned to variable *large*. Thus, the given statement assigns the larger of variables *a* and *b* to variable *large*. The same effect can be achieved using an **if - else** statement given below.

```
if (a > b)
    large = a;
else large = b;
```

We can rewrite the given conditional expression using the assignment expressions as alternative expressions, as shown below. Note that the parentheses are essential as the assignment operator has lower precedence than the conditional expression operator.

```
a > b ? (large = a) : (large = b);
```

We can also use a conditional expression as an argument in the **printf** statement, as shown below:

```
printf("Larger number: %d", a > b ? a : b);
```

We can also rewrite the above statement by calling the **printf** functions from within the conditional expression, as shown below.

```
a > b ? printf("Larger number: %d", a) : printf("Larger number: %d", b);
```

Recall that the `printf` (and `scanf`) function actually returns an integer value that is ignored in most calls. Thus, the `printf` function calls in the above statement are expressions as expected and not statements.

**b)** Determine the number of days in February

An `if-else` statement to test whether a given year is leap or not was given in Example 5.6b. The `if-else` statement given below uses the leap year test to determine the number of days in February.

```
if (yy % 4 == 0 && yy%100 != 0 || yy % 400 == 0) /* leap year test */
    days = 29;
else days = 28;
```

We can rewrite this code using a conditional expression as shown below.

```
(yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0) ? days=29 : days=28;
```

However, the code is concise and more readable when we first determine the desired value and then assign it to `days` as shown below:

```
days = ((yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0) ? 29 : 28);
```

We can verify, by applying the operator precedence rules, that the parentheses in the above statement are not essential but are useful as they improve the readability. The above statement can thus be written by removing the redundant parentheses as shown below.

```
days = (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0) ? 29 : 28;
days = yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0 ? 29 : 28;
```

Let us now use another approach to determine the number of days in February. We can use a simple `if` statement to determine `days` as follows:

```
days = 28;
if (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0)
    days++;
```

Here the value of `days` is first initialized to 28 and then incremented by 1 only if the given year `yy` is a leap year. This code can be written concisely using the conditional expression as

```
days = 28 + (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0 ? 1 : 0);
```

Here, depending on whether the given year is leap or not, we add either 1 or 0 to 28. Although the addition of 0 is meaningless, it cannot be avoided as both the alternative expressions must be specified in a conditional expression. Also, observe that the parentheses cannot be omitted.

Finally, let us see one more variation of this statement. We know that the value of a Boolean expression is 1 if it is true and 0 otherwise. This is exactly what is required in the above conditional expression. Thus, we can omit the conditional expression operator and simply add the value of Boolean expression to 28 as shown below.

```
days = 28 + (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0);
```

Observe that this statement is bit difficult to understand and that the parentheses surrounding the Boolean expression cannot be omitted.

#### c) Print whether a given number is odd or even

The program segment given below reads an integer number  $n$  and prints whether it is odd or even using a conditional expression.

```
scanf("%d", &n);
printf("Given number is ");
n % 2 == 0 ? printf("even") : printf("odd");
```

The code can also be written concisely as shown below:

```
scanf("%d", &n);
printf("Given number is %s", n % 2 == 0 ? "even" : "odd");
```

#### d) Determine simple or compound interest

In Example 5.4b, we have seen how to determine either simple or compound interest on a deposit using an **if-else** statement. The code is rewritten below using the conditional expression.

```
scanf("%f %f %d %c", &p, &r, &n, &itype);
intr = (itype == 'C') ? p * pow(1 + r / 100, n) : p * n * r / 100.0;
printf("interest = %.2f\n", intr);
```

## 5.7 Advanced Concepts

This section discusses some advanced concepts related to relational and logical operators and conditional control statements. A beginner may skip this section in the first reading. However, make sure to read this section as early as possible as it covers some of the common mistakes a beginner is likely to make. If you prefer to skip this section now, at least remember the following points while writing programs:

1. The equality operator must be written as `==` and not as `=` which is an assignment operator.
2. Avoid the use of `++`, `-` and `=` operators in Boolean expressions.

### 5.7.1 Common Mistakes in `if` Statements

#### Misplaced Semicolon after Test Expression

A beginner often writes a semicolon after the parenthesized test expression in `if` statements. First consider the `if-else` statement given below.

```
if (a > b) ;      /* Note the erroneous semicolon */
    max = a;
else max = b;
```

In this code, the value of the larger of `a` and `b` is to be assigned to variable `max`. However, because of the semicolon after the parenthesized test expression in the `if` statement, it is interpreted as shown below.

```
if (a > b)
;
max = a;
else max = b;
```

Note that the semicolon on the second line is actually a **null statement**. It is a valid C statement. Thus, a null statement is included in the `if` statement and the subsequent code is not a part of the `if` statement. Also, the original `if-else` statement has now become a simple `if` statement. This error is easy to identify as the compiler gives an error, such as `else without a previous if`, when it comes across the `else` keyword. However, this comes as a surprise to a novice as he sees a matching `if` for this `else` keyword.

Now consider the more serious situation in which such an erroneous semicolon is written after the test expression in a simple `if` statement, as shown below.

```
if (c > max);      /* Note the erroneous semicolon */
    max = c;
```

If the value of variable `c` is greater than that of `max`, it is to be assigned to variable `max`. However, because of the semicolon after the parenthesized test expression in the `if` statement, it is interpreted as shown below.

```
if (c > max)
;
max = c;
```

Thus, a null statement is included in the `if` statement and the assignment statement is not a part of it. When this statement is executed, nothing happens when the condition `c > max` is satisfied and the value of variable `c` is always assigned to variable `max`, giving wrong results. Note that the compiler does not give any error or warning in this case.

## Missing Braces

A beginner is also likely to forget the braces when multiple statements are to be written in either the `if` or the `else` part of an `if` statement. For example, consider the `if` statement given below that is supposed to calculate and print compound interest if the value of `itype` is 'C' .

```
if (itype == 'C')
    intr = p * pow((1 + r / 100), n);
    printf("Compound interest = %6.2f\n", intr);
```

Note that the last two statements are correctly indented but they are not included within braces as required. Thus, only one statement is actually included within the `if` statement as shown below.

```
if (itype == 'C')
    intr = p * pow((1 + r / 100), n);
printf("Compound interest = %6.2f\n", intr);
```

If the value of `itype` is 'C' , this code segment works as expected, i. e., it will calculate the interest and print it. However, if the value of `itype` is other than 'C' , the compound interest will not be calculated as expected but is printed.

The reader is encouraged to analyze what will happen if the braces in `if` and/or `else` clauses in the following `if-else` statement are omitted.

```
if (itype == 'C') {
    intr = p * pow((1 + r / 100), n);
    printf("Compound interest = %6.2f\n", intr);
}
else {
    intr = p * n * r / 100.0;
    printf("Simple interest = %6.2f\n", intr);
}
```

Remember that this error is more likely to happen when one or more statements are added to existing `if` statements that do not have braces.

## Inadvertent Use of = in Place of == Operator

A beginner often uses the assignment operator in place of the equality (==) operator. C compilers do not indicate any error for such inadvertent use of assignment operators in place of equality operators. This often leads to wrong results and difficult-to-trace *bugs*. However, C compilers may report an error in such situations due to the violation of some other rules, particularly the requirement of *lvalue* to the left-hand side of an assignment operator.

Consider the `if` statement given below:

```
if (a = b)
```

```
...
```

Here, an assignment operator has been used, probably by mistake, in place of an equality operator (`==`). Instead of comparing the values of variables `a` and `b`, now the value of variable `b` is assigned to variable `a` and if this value is non-zero, the expression `a = b` evaluates as true; otherwise, it evaluates as false. Thus, the given if statement is equivalent to the following code:

```
a = b;  
if (b != 0)  
...
```

Now consider the code given below.

```
a = 0;  
b = 0;  
if (a = b)  
    printf("Equal");  
else printf("Unequal");
```

As the values of variables `a` and `b` are equal, we might expect the string "Equal" to be printed. However, the output will be printed as "Unequal" as the value of expression `a = b` (note the use of the assignment operator instead of the equality operator) is 0, the value of variable `a` after the assignment operation.

### Assignment Operators in Test Expressions in `if` Statement

Sometimes, we may wish to use the assignment operator in relational and equality expressions primarily to make the code concise. Consider, for example, the code segment given below:

```
a = b + c;  
d = e * f;  
if (a < d)  
...
```

This code can be written concisely as

```
if ((a = b + c) < (d = e * f))  
...
```

Note that the assignment expressions have been enclosed in parentheses as the assignment operator has lower precedence than the other operators in the expression. During the evaluation of this test expression, the expressions within the parentheses will be evaluated first. Thus, variable `a` is assigned the value of expression `b + c`, which is the value of the first pair of parentheses and variable `d` is

assigned the value of expression `e * f`, which is the value of the second pair of parentheses. These values are then compared as desired. Although this code is concise, it is difficult to understand and beginners should avoid such constructs.

Experienced C programmers often use assignment operators in relational expressions, e. g., to combine data input and comparison operations. Consider that we have to read a character from the keyboard and then compare it with a character, say newline (`\n`) as

```
c = getchar();
if (c == '\n')
...
```

These statements are often combined in a single statement as shown below.

```
if ((c = getchar()) == '\n')
...
```

The `getchar` function first accepts a character from the keyboard and assigns it to variable `c`. This is the value of the assignment expression. This value is then compared with the newline character as desired. Note that in the absence of the parentheses surrounding the assignment expression, the meaning is completely changed:

```
if (c = getchar() == '\n')
...
```

Now the equality operator will be evaluated first as it has higher precedence than the assignment operator. Hence, the `getchar` function (first operand operand of `==` operator) will be executed first to accept a character which will then be compared with the newline character. Now the result of this comparison which is either true (1) or false (0) is assigned to variable `c`. Note that the compiler will not report any error in this situation as well, making the problem very difficult to trace. Hence, programmers should use this construct with care and beginners should completely avoid it.

### Errors in Comparison with Ranges and Multiple Values

We often have to test whether the value of a variable or an expression is in a specified range or not. Beginners often make simple mistakes because of their familiarity with the conventional mathematical notation. For example, the mathematical notation for validity of marks is often written as  $0 \leq \text{marks} \leq 100$ . The use of this notation in a program leads to wrong results and bugs that are very difficult to remove. Consider the `if` statement given below.

```
if (0 <= marks <= 100)
...
```

C compilers do not report any error in such situations. As the `<=` operator has left-to-right associativity, expression `0 <= marks` will be evaluated first as true (1) unless a negative value is entered. Next, the

result of this evaluation is compared with 100, giving a true value. Thus, the test expression will evaluate as false for negative value of `marks` as expected but not for invalid values of `marks` greater than 100.

Another common error is while comparing a value with more than one value. For example, the test for a vowel or whitespace is given in Example 5.7. Beginners often use a comma-separated list for such comparisons. Consider a test for a vowel given below in which `ch` is a variable of type `char`:

```
ch = getchar();
if (ch == 'A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u')
    ...
```

C compilers do not report any error in this case as well since the test expression is interpreted as a comma-separated list of expressions: the first being the equality test `ch == 'A'` and subsequent expressions being the character constants. As the comma operator guarantees left-to-right evaluation, the sub-expressions are evaluated from left-to-right. The value of the entire test expression is thus the value of the last expression in the list, i. e., '`u`'. It is *true* as the ASCII value of '`u`' is not zero. Thus, the test expression in the `if` statement will always evaluate as true irrespective of the value of variable `ch`.

Consider another test given below in which the characters are enclosed in parentheses.

```
ch = getchar();
if (ch == ('A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u'))
    ...
```

The compiler does not give any error in this case as well. Now, the comma-separated list is evaluated first (as it is enclosed in the parentheses) as '`u`'. Thus, the test expression in the `if` statement evaluates as true only if the variable `ch` equals '`u`'.

To avoid such difficult to trace bugs in programs, we must exercise caution while writing such expressions. Remember that we can compare a variable or an expression with only one value at a time and use logical AND/OR operators to compare multiple values.

## 5.7.2 Common Mistakes in Boolean Expressions

### Assignment Operators in Boolean Expressions

As mentioned earlier, C compilers do not indicate any error for inadvertent use of the assignment operator in place of the equality operator leading to wrong results and difficult-to-trace *bugs*. Consider that the `if` statement

```
if (a == 10 && b == 15)
```

contains the inadvertent use of the assignment operator in place of the equality operators as

```
if (a = 10 && b = 15)
```

Here the `&&` operator will be bound first to its operands as it has higher precedence. Next, the right-hand side assignment operator will be bound to its operands, `a = 10` and `b = 15`, causing an *lvalue* error.

However, if the assignment expressions are enclosed in parentheses as

```
if ((a = 10) && (b = 15))
```

the compiler will not give any error. Due to strict left-to-right evaluation of the `&&` operator, variable `a` will be first assigned value 10, destroying its earlier content. As the value of the left-hand side operand of the `&&` operator, i. e., `(a = 10)`, is now non-zero, the right-hand side operand is also evaluated assigning value 15 to variable `b`, destroying its earlier content as well. Finally, a logical AND operation is performed on the values of `a` and `b` to yield the true value. Observe that such expression will always evaluate as true except when variable `a` and/or `b` is assigned zero value. Moreover, the assignment of variable `b` depends on the value with which variable `a` is being compared (in the original expression). Thus, beginners should exercise utmost caution while using comparison operators in Boolean expressions.

### Increment and Decrement Operators in Boolean Expressions

The use of operators that cause side effects, namely, increment (`++`), decrement (`--`) and assignment (`=`) operators, in Boolean expressions may cause unpredictable results due to the peculiar *short-circuit* evaluation. Consider the expression given below that contains the logical AND (`&&`) operator:

```
a++ < 10 && ++b<5
```

Here, expression `a++ < 10` will be always evaluated due to strict left-to-right evaluation of the logical AND operator causing the increment of variable `a`. However, the evaluation of expression `++b < 5` depends on the outcome of evaluation of the expression `a++ < 10` and the value of variable `b` is incremented only if expression `a++ < 10` evaluates as true. Thus, the value of variable `b` depends on the value of variable `a`.

The expression given above can be written by interchanging the operands of the `&&` operator as

```
++b < 5 && a++ < 10
```

Although we expect this expression to be equivalent to the previous expressions, it is not as variable `b` is now always incremented but the increment of variable `a` depends on the value of variable `b`. Thus, two programs using above expressions will give different results depending on the values of `a` and `b`.

Now consider the expression containing the logical OR (`||`) operator given below.

```
a++<10 || ++b < 5
```

It can be easily verified that this expression also suffers from the problems discussed as the logical OR operator also uses strict left-to-right evaluation and short-circuit evaluation. Hence, we should avoid the use of the `++` and `--` operators in Boolean expressions.

### 5.7.3 Alternative Forms of Boolean Expressions

We can use **De Morgan's theorems** (usually studied in Digital Electronics) to write Boolean expressions in alternate forms. These theorems are as follows:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \text{ (Complement of sum equals product of complements)} \quad \dots (1)$$

$$\overline{A \cdot B} = \overline{A} + \overline{B} \text{ (Complement of product equals sum of complements)} \quad \dots (2)$$

Here, A and B are logical variables or expressions, `'.'` denotes the logical AND operation, `'+'` denotes the logical OR operation and the bar above a Boolean variable or expression denotes the logical NOT (i. e., complement).

We also know a property of complement operation that the complement of complement of a Boolean variable (or expression) is equal to the given Boolean variable (or expression) itself. Thus,  $\overline{\overline{A}} = A$ . By performing complement operation on both sides of De-Morgan's expressions given above, we get:

$$A + B = \overline{\overline{A} \cdot \overline{B}} \text{ (Sum equals complement of product of complements)} \quad \dots (3)$$

$$A \cdot B = \overline{\overline{A} + \overline{B}} \text{ (Product equals complement of sum of complements)} \quad \dots (4)$$

We can use these equations to obtain alternative forms for a given Boolean expression. Consider, for example, the test for an uppercase letter given as

```
ch >= 'A' && ch <= 'Z'
```

This expression is of the form  $A \cdot B$ . Using Eq. (4) given above, it can be written as complement of sum of complements of individual terms as shown below.

```
!(! (ch >= 'A') || !(ch <= 'Z'))
```

The inner complement operators can be eliminated to simplify the equation as

```
!(ch < 'A' || ch > 'Z')
```

This is a correct test for an uppercase letter since the expression within the parentheses will evaluate as false for an uppercase letter and true otherwise.

Consider now the leap year test given below.

```
y % 4 == 0 && y % 100 != 0 || y % 400 == 0
```

The expression is of the form  $A \cdot B + C$ . The steps in obtaining equivalent expressions using Eq. (3) and

(4) are shown below.

|                                      |   |                                       |                                                                     |
|--------------------------------------|---|---------------------------------------|---------------------------------------------------------------------|
| Given Eq.                            | : | $A \cdot B + C$                       | $y \% 4 == 0 \ \&\& \ y \% 100 != 0 \    \ y \% 400 == 0$           |
| Eq. (3)                              | : | $\overline{A} + \overline{B} + C$     | $!(! (y \% 4 == 0) \    \ !(y \% 100 != 0)) \    \ (y \% 400 == 0)$ |
| $a = \overline{A}, b = \overline{B}$ | : | $\overline{a+b} + C$                  | $!(y \% 4 != 0 \    \ y \% 100 == 0) \    \ (y \% 400 == 0)$        |
| Eq. (4)                              | : | $(\overline{a+b}) \cdot \overline{C}$ | $!((y \% 4 != 0 \    \ y \% 100 == 0) \ \&\& \ !(y \% 400 == 0))$   |
| $c = \overline{C}$                   | : | $(\overline{a+b}) \cdot c$            | $!((y \% 4 != 0 \    \ y \% 100 == 0) \ \&\& \ y \% 400 != 0)$      |

You can verify that the expressions given above are equivalent tests for a leap year.

## Exercises

- l. State with reasons, which of the following are invalid relational expressions. Assume that the variables are declared as shown below:

```
int a, b, c;  
float x, y, z;  
char ch1, ch2, str[30];
```

- a.  $a < ch1$
- b.  $x \leq \sqrt{y \cdot z}$
- c.  $a++ != 10 + 5$
- d.  $a+b <> c * (m+n)$
- e.  $ch1 \neq ch2$
- f.  $x = 0$
- g.  $(a^2 + b^2) < c^2$
- h.  $abc < x + y + z$
- i.  $x + y \approx z$
- j.  $z != \infty$
- k.  $x \leq \frac{1}{4}$
- l.  $x > 3a + 2b$
- m.  $++ch1 == 65$
- n.  $str == "Yes"$
- o.  $z >> x + y$

- p. `strlen(str) < 20`
- q. `0 <= x <= 1.0`
- r. `a <= b % 5`
- s. `x+a/b ~=~ y`
- t. `a => b`
- }. Evaluate the following relational expressions and also state whether they are true or false. Assume that the variables are initialized as shown below:  
`int m = 10, n = 20;`  
`float p = 1.5, q = -2.5;`  
`char c = 'A', d = 'a';`
- a. `m <= 10`
- b. `m / 3 > 3.14`
- c. `p + q == 1`
- d. `m+n % 12 < 10`
- e. `c - d == 32`
- f. `fabs(q) < p`
- g. `n = 10`
- h. `fabs(floor(q))>2.5`
- i. `++m == 10`
- j. `n++ > 20`
- k. `pow(2,m)>=1000`
- l. `sqrt(n+m / 2) == 5`
- m. `- p>q`
- n. `p > log10(m)`
- o. `m%6 / n%3 > 0`
- p. `floor(p) < m/n + 1`
- }. Write `if` statements for the following conditions:
- Test whether a given integer number is a perfect square or not.
  - Given the quantity and cost of three items, test whether we have sufficient money to buy them or not.

- c. Test whether a given character `ch` matches with `ch1`, `ch2` or `ch3`.
  - d. Given the lengths of the sides of a triangle, test whether it is an isosceles triangle or not.
  - e. Given a number and two ranges, determine whether the given number is included in any one of the ranges or not.
- f. Write C program segments using `if` statements to perform the following operations:
- a. Accept the marks of two students in physics, chemistry and maths and print the student having higher total marks.
  - b. Given the radius of a circle and sides of a rectangle, determine which of them has a larger area.
  - c. Given the coordinates of the left-top corner  $(x_1, y_1)$  and right-bottom corner  $(x_2, y_2)$  of a rectangle, determine whether it is a square or not.
  - d. Given the coordinates of the left-top and right-bottom corners of two rectangles, determine whether they overlap or not.
  - e. Given the coordinates of centers of two circles and their radii, determine whether they overlap or not.
  - f. Given the coordinates of three points  $(x_1, y_1)$ ,  $(x_2, y_2)$  and  $(x_3, y_3)$ , determine whether they are collinear or not.
- i. Several examples of `if` statements are given below. Identify the errors and rewrite the incorrect code segments correctly.
- a. 

```
if a > b
    printf("a is larger");
    else printf("b is smaller")
```
  - b. 

```
if sqrt(a*a + b*b) = c
    printf(Right angle triangle);
```
  - c. 

```
if (a % 2  0)
    printf("a is odd", a);
    else printf("a is even",a);
```
  - d. 

```
if (angle > 180°:
    angle -= 180;
    y = sin(angle);
```
  - e. 

```
int is leap = 0;
if (y % 4 > 0)
    is leap = 0;
otherwise is leap = 1;
```
  - f. 

```
scanf("%d", n);
if (n * n == sqrt(n))
```

- ```
    printf("%d is perfect square", d);
    else printf("%d is not perfect square", d)
```
- g. `d = b * b - 4 * a * c;`  
`If (d >= 0)`  
    `/* roots are real */`  
`else`  
    `/* roots are imaginary */`
- h. `if (ch1 <> ch2)`  
    `printf("The Characters ch1 and ch2 are`  
        `equal");`  
    `else printf("The characters ch1 and ch2 are`  
        `not equal");`
5. Several Boolean expressions are given below. State with reasons, which of them are invalid expressions. Assume that the variables are declared as shown in Problem 1.
- a. `!a`
  - b. `a >= 10 and a < 20`
  - c. `!a && !b`
  - d. `a & b`
  - e. `x < 1.5 && x > 2.0`
  - f. `((x ≤ 0) || (y ≤ 0) || (z ≤ 0))`
  - g. `!a OR b`
  - h. `b > 100 || b < 200`
  - i. `ch1 + (ch2 == 'A' || ch2 == 'B')`
  - j. `a||b + c`
  - k. `a > 10&&b > 20`
  - l. `str=="Ever" || str=="Never"`
  - m. `! p>q`
  - n. `!(ch1=a || ch2=b)`
  - o. `a++ > 10 && ++a > c`
7. Determine the values of the Boolean expressions given below and also state whether they are true or false. Assume that the variables are initialized as shown in Problem 3. Also, indicate the side effects of these expressions on the values of variables used.
- a. `!q`

- b. `m=5 && n >= 10`
- c. `!m && !n`
- d. `m && n`
- e. `n-m % 7>10 || p<1`
- f. `!(m + n >= 30 || ++c/n > 3.25)`
- g. `!p || q`
- h. `m++>=10 || n++<=30`
- i. `c + (d != 'a' || d == 'A')`
- j. `a||b + c`
- k. `p=10 && q=0`
- l. `(m||n) + (p&&q - c||d)`
- m. `! p>q`
- n. `!(d+32==c || d!='a')`
- o. `pow(++m,2)>100 || sqrt(pow(2,n--)) > 1000`
3. Several examples of the `if` statement are given below. Identify the errors and rewrite the incorrect code segments correctly.
- a. `if (a >= 0 || a <= 100)  
 printf("Valid marks");  
else  
 printf("Invalid marks");`
- b. `if (m = 0 && n = 0)  
printf("Error: Invalid matrix size.");  
exit();  
else printf("Enter the matrix:");`
- c. `getchar(ch);  
printf("%c is a ", ch);  
if (ch > 0 && ch < 9)  
 printf("digit");  
else printf("letter");`
- d. `if (day = 7 and month = 3)  
 printf("My birthday")  
else printf("Your birthday");`
4. Write program segments using the `switch` statement to perform the following operations:

- a. Given a character representing one of the colors in a rainbow (VIBGYOR), print the subsequent colors in the rainbow starting from the given color.
- b. Given a month number, print the name of that month.
- c. Consider the pairs of characters used in C programs: (), {}, [] and <>. Accept one of these characters and print the name of the corresponding pair of characters as *Parentheses*, *Curly braces*, *Square brackets* and *Angle brackets*.
- d. Write conditional expressions to perform the following operations:
  - a. Given the marks in a single subject, print the result (Pass/Fail).
  - b. Given two numbers, calculate their sum if both numbers are either odd or even; otherwise, calculate their difference.
  - c. Given the coordinates ( $xc$ ,  $yc$ ) of the center of a circle and its radius  $r$ , determine whether point  $(x, y)$  lies inside it or not.
  - d. Determine whether the sum of digits of a given three-digit number is odd or even.
- e. Rewrite the following statements correctly.
  - a. The C language provides several conditional control statements such as `for`, `while` and `do while`.
  - b. The `if` clause in an `if-else` statement is optional and can be omitted.
  - c. The `switch` statement is used to select one of two alternative groups of statements for execution.
  - d. The `switch` statement can be used to test only integer expressions.
  - e. It is possible to write every statement involving the use of conditional operators by using an `if-else` statement.

## Exercises (Advanced Concepts)

- f. Several examples of the `if` statement are given below. Identify the errors and rewrite the incorrect code segments correctly.
  - a. 

```
if (a >= 0 || a <= 100)
    printf("Valid marks");
else
    printf("Invalid marks")
```
  - b. 

```
if (m = 0 || n = 0)
printf("Error: Invalid matrix size.");
exit();
else printf("Enter the matrix:");
```

c. `getchar(ch);  
printf("%c is a ", ch);  
if (ch > 0 && ch < 9)  
 printf("digit");  
else printf("letter");`

d. `If (a = b)  
 printf("a and b are equal")  
else printf("a and b are equal"`

e. `if (day == 1 to 5)  
 printf("Working day")  
printf("Holiday");`

f. `if (month == 1, 3, 5, 7, 8, 10, 12)  
 days = 31;  
else days = 30;`

g. `if (0 ≤ marks ≤ 100)  
 printf("valid marks");  
else  
 printf("invalid marks")`

h. `/* calc area of a circle or sphere */  
area = if (shape == 'C' || 'c') /* circle*/  
 PI * r * r;  
else /* sphere */  
 4.0 / 3 * PI * pow (r, 3);`

?2. Determine the output of the following programs segments.

a. `a = 0;  
if (a = 0)  
 printf("Zero");  
else printf("Non-zero")`

b. `x = y = 0;  
if (x = 10 && y = 20)  
 printf("Condition is True\n");  
else printf("Condition is False\n");  
printf("x = %d y = %d\n, x, y);`

c. `a = 2, b = 1, c = 0;  
if (a, b, c)  
 printf("True");  
else printf("False");`

d. `a = 10, b = 20;`

```

if (a++ > 10 && b++ > 20)
    printf("Inside if");
printf("a = %d b = %d", a, b);

e. a = 5;
if (a % 2 == 0);
    printf("Even Number:");
printf("%d", a);

f. n = 9;
if(n % 2 == 0 || n % 3 == 0)
    printf("Divisible by 2 or");
    printf("Divisible by three");
printf("Not divisible by 2 or 3");

```

3. Several examples of **switch** statement are given below. Identify the errors and rewrite the incorrect code correctly.

```

a. getchar(color);
switch (color)
{
    case R || r:
        printf("Red");
    case G || g:
        printf("Green");
    case B || b:
        printf("Blue");
    case default:
        printf("Not RGB")
}

b. scanf("%d", &year);
Switch(year) {
    Case 1) y % 400 == 0:
        printf("Leap year");
    break;
    Case 2) y%4==0 && y%100!=0:
        printf("Leap year");
    break;
    else :
        printf("Not a leap year")
    break;
}

c. scanf("%d", &num);
switch(num % 10)

```

```
case 0, 2, 4, 6, 8:  
    printf("Even number\n");  
    break;  
case 1, 3, 5, 7, 9:  
    printf("Odd number\n");  
    break;  
default:  
    printf("Not odd or even");  
}
```

d. char ray\_type;

```
scan("%c", ray_type);  
switch(ray_type) {  
    case a: printf("Alpha rays");  
    case b: printf("Beta rays");  
    case y: printf("Gamma rays");  
    default:  
        printf("No such rays");  
}
```

e. Apply De Morgan's theorems to Boolean expressions given below to obtain other equivalent forms.

```
ch >= 'A' && ch <= 'Z'  
ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z'  
ch == ' ' || ch == '\t' || ch == '\n'
```

# 6      Looping Control

Procedural programming provides three main constructs, namely, sequence, selection and loops. So far we have studied the first two constructs. This chapter presents the third construct: loops.

Loops allow us to execute an activity repeatedly. The C language provides three loop statements, namely, **for**, **while** and **do ... while**. The **for** loop, which is the most commonly used loop statement, is presented first followed by the **while** loop. These loops are entry controlled. The **do ... while** loop, which is an exit-controlled loop, is discussed next. Numerous examples have been presented to illustrate the capabilities of these loops throughout this chapter. The examples in this chapter are simple and do not require any control statements within the loop body. Finally, several advanced concepts related to loops are presented. The nesting of control structures is covered in the next chapter.

## 6.1 The **for** Loop

The **for** loop is used for repetitive execution of a statement or a group of statements. It is a very powerful and flexible statement of the C language. It is generally used in situations where the number of iterations of the loop statement is known or can be determined in advance. It can also be used in situations where the number of iterations is not known or cannot be determined in advance. However, the **while** loop is more convenient in such situations. The general form of the **for** loop is given below.

```
for (initial_expr ; final_expr ; update_expr)  
    statement;
```

This loop executes the contained *statement* repeatedly as decided by three control expressions (*initial\_expr*, *final\_expr* and *update\_expr*). The *initial\_expr* is usually an assignment expression, whereas the *update\_expr* is an increment/decrement or compound assignment expression. The final expression is usually a relational expression but may also be a Boolean expression or any valid C expression. Observe that the control expressions are written within a pair of parentheses and are separated by semicolons.

The most commonly used form of the **for** loop uses a **loop control variable** (also called **loop variable**, **counter variable** or **index variable**). It is a variable of any arithmetic type such as integer, floating-point, character, etc. Most programmers prefer short names for loop variables, e. g., variables *i*, *j* and *k* are commonly used in situations where integer loop variables are required.

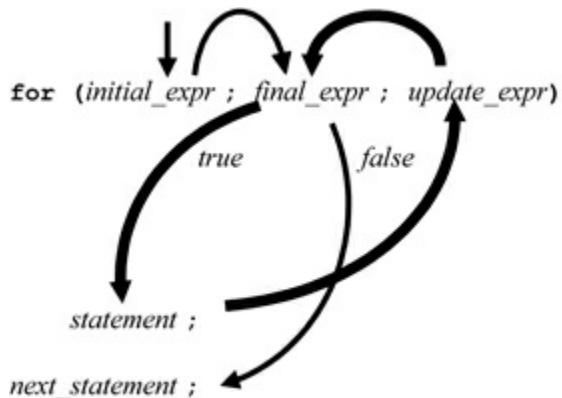
The loop variable assumes a sequence of values as decided by the three control expressions. For each value of the loop variable, the *statement* included in the **for** loop is executed once. The *initial\_expr* initializes the loop variable, the *update\_expr* updates (increment, decrement, etc.) its value and the *final\_expr* is a test expression that compares its value with the desired final value to decide whether the execution of the contained statement should be continued or not.

Consider the for loop given below used to print the string "Hello world\n" four times.

```
for (i = 1; i <= 4; i++)
    printf("Hello world\n");
```

In this **for** loop, variable *i* (assumed to be of type int) is used as a loop variable. The initial expression, *i = 1*, initializes the *i* to 1. The final expression, *i <= 4*, compares value of *i* with the desired final value, i. e., 4. The update expression, *i++*, increments the value of *i* by 1. Thus, the loop variable *i* assumes values 1, 2, 3 and 4. For each of these values, the **printf** statement prints the string "Hello world" followed by a newline. Thus, the output is as follows:

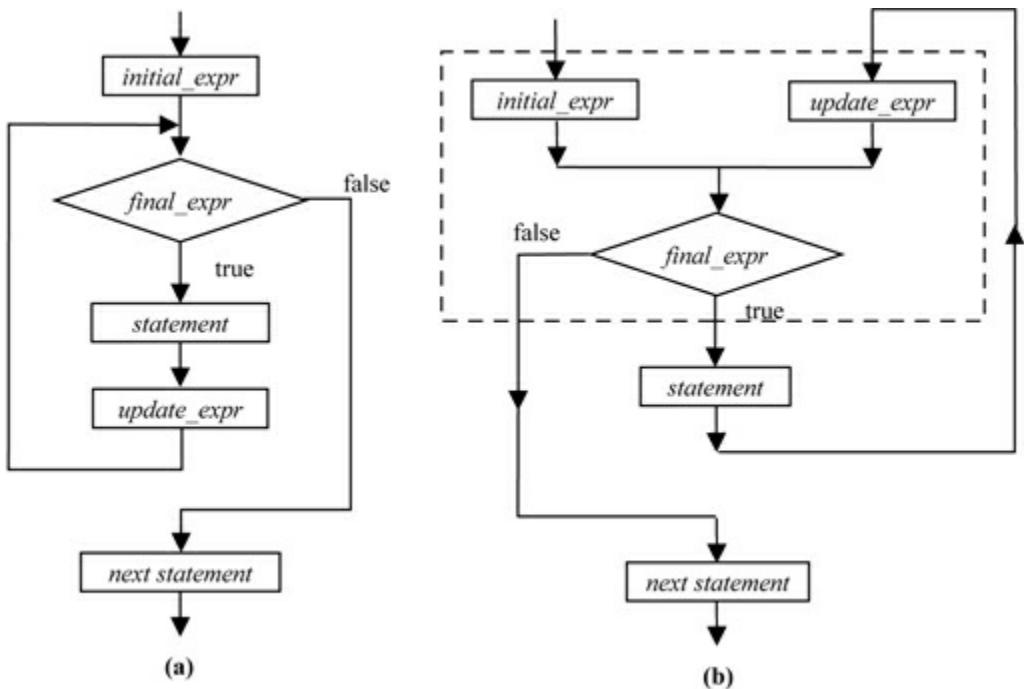
```
Hello world
Hello world
Hello world
Hello world
```



**Fig. 6.1** Control flow diagram of the **for** loop

To understand the execution of the *for* loop, refer control flow diagram shown in Fig. 6.1 and the flowcharts in Fig. 6.2.

The execution of the *for* loop proceeds as follows: Initially, expression *initial\_expr* is evaluated. It performs the initialization of the loop variable. Then *final\_expr* is evaluated. If it is *true*, the *statement* included in the loop is executed followed by evaluation of *update\_expr*.



**Fig. 6.2** Flowcharts of the **for** loop: (a) Commonly-used flowchart (b) Flowchart resembling C syntax

Then *final\_expr* is evaluated again. The execution of *statement* within the loop followed by the evaluation of *update\_expr* and *final\_expr* continues as long as *final\_expr* evaluates *true*; otherwise, the control leaves the **for** loop and continues with the execution of the next statement in the program. Note that the execution of the contained *statement* followed by the evaluation of update expression and final expression is commonly referred as an *iteration* of the loop. Also, the statement contained in the loop is commonly referred to as the *body* of the loop.

Although the values of loop variables used in the above **for** loop are quite natural, it is a common practice to use zero as the initial value for the loop variable. Moreover, as the array subscripts start from zero, the **for** loops used to process arrays also use zero as the initial value of loop variable. Hence, the style of using zero as the initial value of loop variables, wherever appropriate, is strongly recommended. Such a **for** loop to perform *n* iterations is given below.

```
for (i = 0; i < n; i++)
```

Note the use of the '<' operator in the final expression (*i < n*). Now the program segment to print **Hello world** five times can be rewritten as follows:

```
for (i = 0; i < 5; i++)
    printf("Hello world\n");
```

### Example 6.1 Simple for loops

a) Print a line of dashes

```
for (i = 0; i < 50; i++)
    printf("-");
printf("\n");
```

The **for** loop given above prints a line of fifty dashes followed by a newline. The **for** loop uses **i** as the loop variable whose initial and final values are 0 and 49, respectively (note the **<** operator in **i < 50**). The update expression increments the value of **i** by 1. Thus, the loop variable assumes values as 0, 1, 2, ..., 49. For each value of **i**, the **printf** statement contained within the **for** loop prints a dash. Finally, the **printf** statement after the **for** loop prints a newline character to move the cursor to the next line.

b) Print integer numbers in a given range

```
printf("Enter range (m, n): ");
scanf("%d %d", &m, &n);

for (num = m; num <= n; num++)
    printf("%d ", num);
```

This code segment first accepts a range of values in variables **m** and **n**, both of type **int**. The variable **num**, also of type **int**, is used as the loop variable. It assumes values from **m** to **n**. For each value of loop variable **num**, the **printf** statement within the **for** loop is executed. Thus, all the numbers from **m** to **n** are printed. Observe the space character after the **%d** format in the **printf** statement. This causes a space to be printed after each value of **num**. The output is given below.

```
Enter range (m, n): 11 20
11 12 13 14 15 16 17 18 19 20
```

c) Print integer numbers from 100 to 0 in steps of -10

```
for (j = 100; j >= 0; j -= 10)
    printf("%d ", j);
```

In this example, the initial value of loop variable **j** (of type **int**) is 100 and the update expression (**j -= 10**) reduces it by 10 after each iteration of the **for** loop. The **printf** statement is executed as long as the value of **j** is greater than or equal to zero. Thus, **j** assumes values as 100, 90, ..., 0. These values are printed using a **printf** statement as shown below.

```
100 90 80 70 60 50 40 30 20 10 0
```

d) Print a table of squares of numbers from 0.1 to 1.0 in steps of 0.2

```
for (x = 0.1; x <= 1.0; x += 0.2)
    printf("%5.1f %6.2f\n", x, x * x);
```

In this example, the initial and final values of loop variable `x` (of type `float`) are 0.1 and 1.0, respectively. The increment expression, `x += 0.2`, increases the value of `x` by 0.2. Thus, `x` assumes values 0.1, 0.3, 0.5, 0.7 and 0.9. Note that the final expression, `x <= 1.0`, evaluates as *false* for next value of `x`, i. e., 1.1. The `printf` statement, which is executed for each value of `x`, prints the value of `x` and `x * x` followed by a newline. The output is as shown below.

```
0.1  0.01
0.3  0.09
0.5  0.25
0.7  0.49
0.9  0.81
```

**e) Calculate the sum of integer numbers from 1 to  $n$**

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

sum = 0;
for (k = 1; k <= n; k++)
    sum += k;
printf("Sum = %d\n", sum);
```

This program segment calculates the sum of integer numbers from 1 to  $n$ . Initially, the value of `n` is read from the keyboard and variable `sum` is initialized to zero. Then a `for` loop is set up in which the loop variable `k` assumes values from 1 to  $n$ . For each value of `k`, the assignment statement included in the `for` loop is executed and the value of `k` is added to the previous value of `sum`. Thus, when the execution of the `for` loop is over, variable `sum` will contain the sum of numbers from 1 to  $n$ . Finally, the value of `sum` is printed on the screen. The program output is shown below:

```
Enter value of n: 10
Sum = 55
```

**f) Print all uppercase letters followed by all lowercase letters on the next line**

```
/* print uppercase letters */
for (ch = 'A'; ch <= 'Z'; ch++)
    putchar(ch);
putchar('\n');

/* print lowercase letters */
```

```

for (ch = 'a'; ch <= 'z'; ch++)
    putchar(ch);
putchar('\n');

```

In this example, we use two **for** loops in conjunction with the **putchar** library function to print all uppercase letters followed by all lowercase letters. The loop variable **ch** is assumed to be of type **char** or **int**. The first **for** loop prints the uppercase letters as loop variable **ch** assumes values as '**A**' , '**B**' , ... , '**Z**'. Then a **putchar** function prints a newline character to move the cursor to the next line on which the second **for** loop prints the lowercase letters. Note that the same loop variable is used for both **for** loops. The output is given below.

```

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

```

### Program 6.1 Determine the factorial of a given integer number

Write a program to determine the factorial of a given integer number.

**Solution:** The factorial of an integer number  $n$  is defined as the product of integer numbers from 1 to  $n$ , i. e.,

$$n! = \prod_{i=1}^n i = n \times (n-1) \times (n-2) \times \dots \times 1 .$$

Note that the factorial of 0 is 1, i. e.,  $0!=1$ . The technique used here to determine this product is similar to that used in Example 6.1e to determine the sum of integer numbers from 1 to  $n$ . First initialize a variable **fact** to 1 and then multiply it by each number from  $n$  to 2 using a **for** loop:

```

fact = 1;
for(i = n; i > 1; i--)
    fact *= i;

```

Observe that this code segment correctly calculates the factorial of 0 and 1 although the statement within the **for** loop is never executed for these values.

The program given below declares variables **n** and **fact** of type **int** to represent the given number and its factorial, respectively. It first accepts the value of variable **n** and then determines its factorial and prints it.

```

/* Factorial of a given number */
#include <stdio.h>

int main()
{
    int n, fact;      /* given number and its factorial */
    int i;

```

```

printf("Enter a small integer number :");
scanf("%d", &n);

/* calculate factorial of n */
fact = 1;
for(i = n; i > 1; i--)
    fact *= i;

printf("%d! = %d\n", n, fact);
return 0;
}

```

The program is executed twice (in Turbo C/C++) and the output is given below:

```

Enter a small integer number : 5
5! = 120

```

```

Enter a small integer number : 8
8! = -25216

```

Observe that the program prints wrong output when  $n = 8$  because the value of  $8!$  ( $40320$ ) is outside the range of the `int` data type in Turbo C/C++ ( $-32768$  to  $32767$ ). Thus, the program will give wrong output for  $n \geq 8$ . To overcome this problem, we can declare variable `fact` of type `long int` that provides a much larger range and use the `%ld` conversion specification in the `printf` statement to print the value of `fact`. Alternatively, we can use IDE supporting 32-bit `int` such as Code::Blocks or Dev-C++. The program can now print the correct factorial of numbers up to 13 (remember that factorial values grow very fast).

### 6.1.1 Using Compound Statements in the `for` Loop

As illustrated in Example 6.1, the statement contained in a `for` loop may be a simple statement such as an assignment or a function call. It may also be a block statement as shown below:

```

for ( initial_expr ; final_expr ; update_expr )
{
    statement ;
    statement ;
    ...
}

```

In each iteration of this loop, the statements included within the block are executed in the given sequence. Note that the statements within the block are indented to improve the readability. In this book, we use a slightly different style for the opening curly brace of the block as shown below:

```
for (initial_expr; final_expr; update_expr) {  
    statement;  
    statement;  
    ...  
}
```

A common error, particularly among beginners, is the omission of braces when the **for** loop contains more than one statement for repetitive execution as shown below.

```
for (initial_expr; final_expr; update_expr)  
    statement1;  
    statement2;
```

One might think that this **for** loop repeatedly executes both statements (*statement1* and *statement2*). However, in the absence of braces surrounding these statements, only *statement1* is included within the loop. Thus, *statement2* will be executed only once after the execution of the **for** loop. Such mistakes usually occur when we add one or more statements to a **for** loop containing a single statement. To avoid such mistakes, programmers often use a block statement even if there only one statement within the **for** loop as shown below.

```
for (initial_expr; final_expr; update_expr) {  
    statement;  
}
```

### Program 6.2 Sum and average of *n* given numbers

Write a program to read *n* numbers from the keyboard and determine their sum and average.

**Solution:** A program segment to calculate a sum of integers from 1 to *n* is given in Example 6.1e. In this example, similar technique is used to determine the sum of *n* numbers entered from the keyboard. Thus, variable **sum** is initialized to zero and a **for** loop is set up to process *n* numbers. In each iteration of this loop, a number is read from the keyboard in variable **x** of type **float** and added to variable **sum** as shown below.

```
sum = 0.0;  
for(i = 0; i < n; i++) {  
    scanf("%f", &x);  
    sum += x;  
}
```

Note that the body of the loop is a block containing two statements. A complete program to determine the sum and average of *n* given numbers is given below. Initially, the value of *n* is read from the keyboard. Then *n* numbers are accepted from the keyboard and their sum is determined as explained above. Finally, the average of the given numbers (**avg**) is calculated after the **for** loop and the values of

sum and avg are printed.

```
/* Sum and average of n given numbers */
#include <stdio.h>
int main()
{
    float x;          /* number(s) entered */
    float sum, avg;  /* sum and average */
    int n, i;

    printf("How many numbers? ");
    scanf("%d", &n);

    /* read numbers and calculate their sum */
    printf("Enter %d numbers: ", n);
    sum = 0.0;
    for(i = 0; i < n; i++) {
        scanf("%f", &x);
        sum += x;
    }
    avg = sum / n;
    printf("Sum = %f  Average = %f\n", sum, avg);
    return 0;
}
```

The program output is given below.

```
How many numbers? 5
Enter 5 numbers: 1.1 2.2 3.3 4.4 5.5
Sum = 16.500000      Average = 3.300000
```

### Program 6.3 Print the Fibonacci series

Write a program to print the first  $n$  terms in the Fibonacci series.

**Solution:** Let us denote the  $i$  th term in the Fibonacci series as  $F_i$ , where  $i$  is assumed to take values starting from 0. Thus, the first four terms in the Fibonacci series are denoted as  $F_0$ ,  $F_1$ ,  $F_2$  and  $F_3$ . The first two terms are given as  $F_0=0$  and  $F_1=1$ . Each subsequent term is obtained by adding two previous terms in the series. Thus, the first 10 terms in the series are given as 0, 1, 1, 2, 3, 5, 8, 13, 21 and 34. The equation for term  $F_i$ ,  $i \geq 2$ , can be expressed in the form of a recurrence relationship as  $F_i = F_{i-1} + F_{i-2}$ .

$i$	0	1	2	3	4
$F_i$	0	1			
	prev2	prev1	term		

(a)

$i$	0	1	2	3	4
$F_i$	0	1	1		
	prev2	prev1	term		

(b)

**Fig. 6.3**

Let us use the variable `term` to represent term  $F_i$  in the Fibonacci series and variables `prev1` and `prev2` to represent previous terms ( $F_{i-1}$  and  $F_{i-2}$ ) as shown in Fig. 6.3a for  $i = 2$ . Now term  $F_2$  can be obtained as

```
term = prev1 + prev2;
```

To determine the next term in the series, first move `prev1` and `prev2` to next position as shown in Fig. 6.3b using the following statements:

```
prev2 = prev1;
prev1 = term;
```

Now term  $F_3$  can be calculated using the same statement given above (`term = prev1 + prev2`).

We can determine the subsequent terms in the series by repeating the above procedure. A program to determine and print the first  $n$  terms in the Fibonacci series is given below.

```
/* Print Fibonacci series */
#include <stdio.h>

int main()
{
    int prev1, prev2; /* previous terms in the series */
    int i, n;          /* n: terms to be printed */

    printf("How many terms? ");
    scanf("%d", &n);

    /* Generate and print Fibonacci series */
    printf("\nFibonacci series: ");
    prev2 = 0;           /* first two terms*/
    prev1 = 1;
```

```

printf("%d %d ", prev2, prev1);

for(i = 2; i < n; i++) {
    int term = prev1 + prev2;           /* next term */
    printf("%d ", term);
    prev2 = prev1;
    prev1 = term;
}
return 0;
}

```

Initially, the program accepts the value of `n` from the keyboard. It then initializes the first two terms in the series (`prev1` and `prev2`) to 0 and 1, respectively, and prints them. A `for` loop is then used to print the remaining terms in the series with the values of `i` from 2 to `n-1`. The statements within the `for` loop calculate next term ( $F_i$ ) in variable `term`, print it and then assign new values to variables `prev1` and `prev2`, which are used in the next iteration to determine the next `term` in the series. Observe that the variable `term` is declared inside the `for` loop as it is not required outside the loop. The program output is given below.

```

How many terms? 10
Fibonacci series: 0 1 1 2 3 5 8 13 21 34

```

#### Program 6.4 Evaluation of $e^x$ by series approximation

Write a program to evaluate  $e^x$  using the first  $m$  terms in the series:

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

**Solution:** Let us denote the terms in this series as  $T_i$  where  $i$  takes values starting from 0. Thus,  $T_0 = 1$ ,  $T_1 = x$ ,  $T_2 = x^2/2!$  and so on. A straight-forward computation of term  $T_i$  (i. e.,  $x^i/i!$ ) involves evaluation of  $x^i$  and  $i!$ , each requiring  $i-1$  multiplications. Thus, a direct computation of this series is compute intensive. In this program, an efficient approach for computation of such a series is presented. Observe that there is a fixed relation between term  $T_i$  and previous term  $T_{i-1}$ , which can be expressed as their ratio as:

$$\frac{T_i}{T_{i-1}} = \frac{x^i/i!}{x^{i-1}/(i-1)!} = \frac{x}{i}$$

Term  $T_i$  can thus be obtained from term  $T_{i-1}$  as  $T_i = \frac{x}{i} T_{i-1}$ . Thus, knowing the first term (1), we can calculate each subsequent term in the series using just one multiplication and one division operation. The program to determine the value of  $e^x$  using the first  $m$  terms in the series is given below. Besides the variables `x` (of type `float`) and `m` (of type `int`), it uses the variables `term` and `sum`, both of type

`float`, to represent a term in the series and sum of terms, respectively. The program first accepts values of `x` and `m` and initializes variables `term` and `sum` to 1.0. Then a `for` loop is used to evaluate subsequent terms in the series and add them to `sum`. Finally, the value of `sum`, i. e., the value of  $e^x$  is printed along with the value obtained by the standard library function `exp`.

```
/* Evaluation of  $e^x$  by series approximation:  
   uses m terms (user specified) in the series */  
#include <stdio.h>  
#include <math.h>  
  
int main()  
{  
    float x, term, sum;  
    int m, i;  
    printf("Enter x and number of terms to be used: ");  
    scanf("%f %d", &x, &m);  
  
    /* evaluate  $e^x$  */  
    term = sum = 1.0; /* first term and initial sum */  
    for (i = 1; i < m; i++) {  
        term *= x / i; /* next term */  
        sum += term;  
    }  
  
    printf("e^x (using %2d terms in the series): %f\n", m, sum);  
    printf("e^x using library function: %f\n", exp(x));  
  
    return 0;  
}
```

The output of the program is given below.

```
Enter x and number of terms to be used: 2.5 20  
e^x (using 20 terms in the series): 12.182494  
e^x using library function: 12.182494
```

## 6.2 The `while` Loop

The `for` loop, presented in the previous section, is particularly useful when the number of iterations is known or can be determined in advance. The `while` loop is particularly useful when the number of iterations is not known or cannot be determined in advance. The general syntax of the `while` loop is as

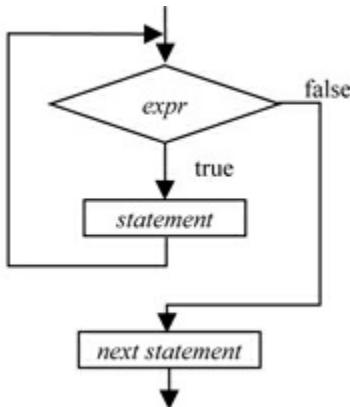
follows:

```
while ( expr )
    statement;
```

where *expr* is the loop control or test expression that may be any valid C expression such as arithmetic, relational or logical and the *statement* is the body of the loop that is to be executed repeatedly. Note that the body of a **while** loop will usually be a block statement as shown below.

```
while ( expr ) {
    statement;
    statement;
    ...
}
```

The **while** loop is an **entry-controlled** or **top-tested loop** as the loop control or test appears in the beginning of the loop. The flowchart of the while Fig. 6.4



**Fig. 6.4** Flowchart for the *while* loop

The execution of the **while** loop proceeds as follows: Initially, test expression *expr* is evaluated. If it evaluates as true (i. e., non-zero), the body of the loop is executed and the control is transferred to the beginning of the loop where expression *expr* is evaluated again. However, if the test expression evaluates as false (i. e., zero), the control leaves the loop. Thus, the body of the loop is executed *while* (as long as) test expression *expr* evaluates as true (non-zero). Note that if the expression *expr* evaluates as false (i. e., zero) when the loop is entered, the body of the loop is not executed at all.

#### Example 6.2 Simple **while** loops (unknown number of iterations)

- Money payment – add numbers until desired sum is obtained

Consider that we have to add the given numbers until the desired sum is obtained (i. e., as long as the sum is less than a specified value). This is required in many situations. For example, to pay a shopkeeper for the goods purchased, we can continue to give one currency note at a time as long as the amount being given is less than the desired amount. Such a sum can be calculated using a `while` loop as shown below.

```
printf("Enter payment to be made: ");
scanf("%d", &payment);

printf("Enter currency note values:\n");
sum = 0;
while (sum < payment) {
    scanf("%d", &value);
    sum += value;
}

printf("\nAmount payable: %d Amount paid: %d\n", payment, sum);
```

Initially, the value of the amount to be paid is accepted from the keyboard in variable `payment` (of type `int`). Then variable `sum` is initialized to zero and a `while` loop is setup to execute its body as long as the value of `sum` is less than `payment`. The statements within the body of the loop accept a number (value of next currency note) from the keyboard and add it to `sum`. The program containing this code segment is executed once and the output is given below.

```
Enter payment to be made: 125
Enter currency note values:
100
20
10
```

```
Amount payable: 125 Amount paid: 130
```

Observe that we have pressed the *Enter* key after each value of currency note. As a result, the program accepts only the required data values and displays the `sum` of entered numbers when `sum` exceeds `payment`. If we enter the currency note values separated by spaces we might sometimes get an output that looks wrong, similar to one shown below.

```
Enter payment to be made: 125
Enter currency note values:
100 20 10 10 10
```

```
Amount payable: 125 Amount paid: 130
```

However, this output is correct. Actually, the last two numbers were not processed by the program as the desired sum was obtained after the third number. This can be easily verified by printing the

numbers read from the keyboard within the `while` loop as shown below (only the `while` loop is shown to save space):

```
while (sum < payment) {  
    scanf("%d", &value);  
    printf("%d ", value);  
    sum += value;  
}
```

The output of the modified program is given below. Observe that after we press the *Enter* key, the program first prints the three values processed by it followed by the result.

```
Enter payment to be made: 125  
Enter currency note values:  
100 20 10 10 10  
100 20 10
```

Amount payable: 125 Amount paid: 130

Note that the code to calculate the sum can be written concisely using the `for` loop as shown below.

```
printf("Enter currency note values:\n");  
for (sum = 0; sum < payment; sum += value)  
    scanf("%d", &value);
```

#### b) Add numbers until a negative or zero is encountered

The program segment given below accepts numbers from the keyboard until we enter a zero or a negative number and calculates their sum excluding the last number.

```
printf("Enter positive numbers (0 or -ve number to stop):\n");  
sum = 0;  
scanf("%d", &num);  
while (num > 0) {  
    sum += num;  
    scanf("%d", &num);  
}  
printf("Sum = %d ", sum);
```

The variables `num` and `sum` (both of type `int`) are used to represent a number entered from keyboard and the sum of the numbers, respectively. Initially, variable `sum` is initialized to zero and the first number is read in variable `num`. A `while` loop is then set up in which we add a number to `sum` and accept next number. The body of the loop is executed as long as the entered number is positive. The output obtained by executing the program containing the above code is given below.

Enter positive numbers (0 or -ve number to stop):

10 15 20 50 0

Sum = 95

Note the use of a `scanf` statement to read a number before the `while` loop. Another `scanf` statement is required within the `while` loop to read subsequent numbers. Compare this `while` loop with that in the previous example (money payment) which requires only one `scanf` statement.

### c) Evaluation of $e^x$ correct up to three decimal places using series approximation

In Program 6.4, we evaluated the value of  $e^x$  using the first  $m$  terms in its series approximation. Although it calculate the desired value, we cannot tell beforehand the number of terms required for the evaluation of this series correct up to a specified decimal position. Thus, we have to either perform more computation than required or be satisfied with an inaccurate result.

The modified code to evaluate this series correct up to three decimal places is given below. It uses a `while` loop to determine the next term and add it to sum as long as its value is greater than 0.0001.

```
term = sum = 1.0;          /* first term and initial sum */
i = 1;
while (term > 0.0001) {
    term *= x / i;        /* next term */
    sum += term;
    i++;
}
```

We can generalize this code to evaluate the series correct up to a user-specified number of digits (say  $d$ ) by substituting the value 0.0001 by `pow(10, -(d+1))`, i. e.,  $10^{-(d+1)}$  and including the `math.h` header file in the program.

### Program 6.5 Sum of digits of an integer number

Write a program to determine the sum of digits of a given non-negative integer number.

**Solution:** In Example 3.7, we determined the sum of digits of a non-negative integer number by determining the least significant digit and then removing it from given number. In this example, we generalize this technique to determine the sum of digits of any non-negative integer number. To separate the digits of given number `num`, we can use a `while` loop as shown below:

```
while (num > 0) {
    digit = num % 10;
    num /= 10;
}
```

In each iteration of this loop, the least significant digit of number `num` is separated in variable `digit` and then removed from the given number. The statements within this loop are executed as long as the value of `num` is greater than zero, i. e., as long as the number contains one or more digits.

Note that the value of `digit` will be modified in subsequent iterations. Thus, we should immediately add it to variable `sum` used to store the sum of digits. The variable `sum` should be initialized to zero before the loop. The program segment given below determines the sum of digits of the number `num`:

```
sum = 0;
while (num > 0) {
    digit = num % 10;
    sum += digit;
    num /= 10;
}
```

Note that this program works correctly even if the given number is zero. The complete program given below accepts a number from the keyboard, determines the sum of its digits and prints the sum.

```
#include <stdio.h>

int main()
{
    int num, digit, sum;

    printf("Enter integer number: ");
    scanf("%d", &num);

    sum = 0;
    while (num > 0) {
        digit = num % 10; /* add LS digit to sum */
        sum += digit;
        num /= 10; /* remove LS digit from num */
    }
    printf("Digit sum: %d\n", sum);
    return 0;
}
```

The program output is given below.

```
Enter integer number: 12345
Digit sum: 15
```

Recall that in Turbo C/C++, the range of integer numbers is -32768 to 32767. Thus, this program will

give the correct result only for numbers from 0 to 32767. For example, the sum of digits of numbers 75000 and 32768 is printed as 23 and 0, respectively. This is because when interpreted as `int`, the values of these numbers are actually 9464 and -32768. This can be verified by including a `printf` statement after the `scanf` statement.

The code to determine the sum of digits can also be written in a concise way using a `for` loop instead of the `while` loop and eliminating the variable `digit` as shown below.

```
for(sum = 0; num > 0; num /= 10)
    sum += num % 10;
```

### Program 6.6 GCD using Euclid's algorithm

Write a program to determine the greatest common divisor (GCD) of two integer numbers.

**Solution:** Let us use variables `m` and `n` to represent two integer numbers and variable `r` to represent the remainder of their division, i. e.,  $r = m \% n$ . Euclid's algorithm to determine the GCD of two numbers `m` and `n` is given below and its action is illustrated for `m = 50` and `n = 35`.

```
while(n > 0) {
    int r = m % n;
    m = n;
    n = r;
}
```

Iteration	m	n	$r = m \% n$
1	50	35	15
2	35	15	5
3	15	5	0
4	5	0	(GCD)

In each iteration of this loop, we determine the remainder ( $r = m \% n$ ) and assign current values of variables `n` and `r` to variables `m` and `n`, respectively. Execution is continued as long as the value of divisor `n` is greater than zero. When the value of `n` becomes zero, the value of variable `m` is the GCD of the given numbers as indicated above. The complete program is given below.

```
/* GCD of two numbers using Euclid's algorithm */
#include <stdio.h>

int main()
{
    int m, n;          /* given numbers */
```

```

printf("Enter two integer numbers: ");
scanf("%d %d", &m, &n);

while(n > 0) {
    int r = m % n;
    m = n;
    n = r;
}
printf("GCD = %d\n", m);
return 0;
}

```

This program works correctly even when the value of *m* is smaller than that of *n*. In this case, the first iteration of the loop causes the values of *m* and *n* to be exchanged to have *m* > *n* as required.

### 6.2.1 The **while** Loop for Known Number of Iterations

The examples presented so far illustrate the use of the **while** loop in situations having unknown number of iterations. The **while** loop can also be used in situations where number of iterations is known or can be determined in advance, although a **for** loop is more suitable in such situations. The **for** loop

```
for (initial_expr; final_expr; update_expr)
    statement;
```

can be written using an equivalent **while** loop as shown below.

```
initial_expr;
while (final_expr) {
    statement;
    incr_expr;
}
```

For example, consider the **for** loop given below to print numbers from 0 to 9:

```
for(i = 0; i < 10; i++)
    printf("%d", i);
```

This **for** loops can be written using an equivalent **while** loop as shown below:

```
i = 0;
while (i < 10) {
```

```
    printf("%d", i);
    i++;
}
```

Since every **for** loop can be written using an equivalent **while** loop and vice-versa, one of these loops is obviously redundant. However, C programmers use both the loops in their programs. As already mentioned, if the number of iterations is known or can be determined in advance, the **for** loop is preferred; otherwise, the **while** loop is preferred.

### Example 6.3 Simple **while** loops (known number of iterations)

#### a) Factorial of an integer number

In Program 6.1, we determined the factorial of a given integer number  $n$  using a **for** loop as shown below.

```
fact = 1;
for(i = n; i > 1; i--)
    fact *= i;
```

This can be rewritten using an equivalent **while** loop as follows:

```
fact = 1;
i = n;
while (i > 1) {
    fact *= i;
    i--;
}
```

#### b) Calculate the sum of $n$ numbers entered from the keyboard

A program segment to calculate the sum of  $n$  numbers using a **for** loop is given below.

```
sum = 0.0;
for(i = 0; i < n; i++) {
    scanf("%f", &x);
    sum += x;
}
```

This can be written using a **while** loop as shown below.

```
sum = 0.0;
i = 0;
```

```

while(i < n) {
    scanf("%f", &x);
    sum += x;
    i++;
}

```

### 6.2.2 Text Input Using a Loop and the `getchar` Function

So far we have seen two approaches to read text from the keyboard. In first approach, we use the `%s` format in the `printf` statement to read a single word terminated with a white-space character. Another approach is to use the `gets` library function which allows us to read a line of text, that may contain whitespace characters, terminated by a newline character.

Now let us understand how to read text terminated by any specific character (such as '.', '?', etc.) and display it on the screen. For this, we setup a `while` loop to read one character at a time (using the `getchar` function) and display it (using the `putchar` function) until the desired character is entered. The program segment given below reads text terminated by a period and displays it on the screen.

Now let us understand how to read text terminated by any specific character (such as '.', '?', etc.) and display it on the screen. For this, we setup a `while` loop to read one character at a time (using the `getchar` function) and display it (using the `putchar` function) until the desired character is entered. The program segment given below reads text terminated by a period and displays it on the screen.

```

printf("Enter text followed by a period and Enter key:\n");
ch = getchar();
while (ch != '.') {
    putchar(ch);
    ch = getchar();
}

```

Initially, the `getchar` function reads a character from the keyboard in variable `ch` of type `char`. Then a `while` loop is set up. As long as the value of `ch` is not equal to period ('.'), the statements within this loop display current contents of `ch` and read another character in it from the keyboard. Thus, the code given above accepts a sequence of characters followed by a period and displays this text on the screen.

If you read this code carefully, you will probably bet that it will not work as expected. There is of course a valid reason for that. Every character typed from the keyboard is displayed on the screen as well. Also, the `putchar` function displays the character as soon as it is read from the keyboard. Thus, we expect the output of this code to be completely mixed up with the text entered from the keyboard. For example, if we enter the text `Good Morning` followed by a period (and *Enter* key), we expect the text displayed on the screen as

Enter text followed by a period and Enter key:  
GGoooodd MMoorrnniinngg.

Only one period is expected as the `while` loop terminates when it is entered. However, you will be surprised to know that this is not the actual output. In fact, we get a neat display as shown below:

Good Morning.  
Good Morning

To understand this output, recall that most console I/O functions including `getchar` and `putchar` perform buffered I/O. The text typed from the keyboard is stored in a buffer (portion of memory) and sent to the program only when the *Enter* key is pressed. Thus, the text typed from the keyboard is first displayed on the screen and there is no possibility of the output of `putchar` function getting mixed with it.

When the *Enter* key is pressed, the text typed is forwarded to program for processing. Now characters in input text are read one by one using the `getchar` function and printed on screen using `putchar`. Note that the output of `putchar` is also not directly written to the screen. Instead, it is written to another buffer. The contents of this buffer are sent to the display device when one of the following condition is satisfied: (a) a newline character is written to buffer, (b) the output buffer is full, (c) the output operation is over and (d) output operation is followed by an input operation.

If we include some text after the period, it is not processed by the `while` loop, as shown in the output given below (the first line is input text and the second is output as before):

C is powerful. C is efficient.  
C is powerful

Finally, note that the input may span several lines, as illustrated in the output given below. Note that the output line is printed as soon as we press the *Enter* key after each input line.

C is a very powerful and  
C is a very powerful and  
efficient programming language.  
efficient programming language

## 6.3 The `do ... while` Loop

The `while` loop discussed in last section is particularly suitable when the number of iterations is not known or can not be determined in advanced. In this section, another loop that is useful in similar situations, the `do ... while` loop is discussed. However, the difference is that the `do ... while` is a **exit-controlled** or **bottom-tested loop** as opposed to the `while` loop which is entry-controlled or top-tested. Thus, the statement(s) within a `do ... while` loop will be executed at least once, whereas the statement(s) within a `while` loop may not be executed at all. The general syntax of the `do ... while` loop is given below.

```

do
    statement ;
while ( expr );

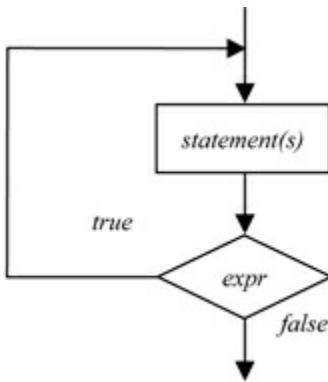
```

where *expr* is the loop control expression that may be any valid C expression such as arithmetic, relational or logical and *statement* is the loop body that is to be executed repeatedly. The body of the **do ... while** loop may comprise a compound or a block statement as shown below.

```

do {
    statement ;
    statement ;
    ...
} while ( expr );

```



**Fig. 6.5 Flowchart for *do ... while* loop**

The flowchart for the **do ... while** loop is given in Fig. 6.5. When the loop is entered, the statement(s) within the loop are executed and then expression *expr* is evaluated. If the value of *expr* is true (non-zero), the body of the loop is executed again; otherwise, control leaves the loop. Thus, the body of the loop is executed until expression *expr* evaluates as true (non-zero). Note that the statements within the **do ... while** loop will be executed at least once.

Although the **for** loop is more suitable in situations where the number of iterations are known or can be determined in advance, we can also use the **do ... while** loop in such situations. This can be done as follows:

```

initial_expr ;
do {
    statement ;
    update_expr ;
} while (final_expr);

```

However, remember that this is not equivalent to the **for** loop given below as the **do ... while** loop is bottom-tested, whereas the **for** loop is top-tested.

```
for (initial_expr; final_expr; update_expr)
    statement;
```

#### **Example 6.4** Simple do ... while loops

##### a) Sum of digits of an integer number

A program to determine the sum of digits of a given non-negative integer number using a **while** loop is presented in Program 6.5. The program segment given below does the same thing using a **do ... while** loop.

```
sum = 0;
do {
    sum += num % 10; /* add LS digit to digit_sum */
    num /= 10;         /* remove LS digit from num */
} while (num > 0);
```

The only problem with this implementation is that the statements within the loop will be executed at least once giving incorrect results for negative numbers. To ensure that the given number **num** is non-negative, we can declare it of type **unsigned int**.

##### b) Accept data from the keyboard until correct data is entered

As a good programming practice, we display a message to prompt the user before accepting data from the keyboard. This enables the user to enter the required data correctly. However, the user may still enter incorrect data. Such data may cause the programs to print incorrect results. One way to handle this problem is to print an error message and stop the program execution when incorrect data is encountered. However, the user will usually expect the program to discard incorrect data and allow him/her to reenter correct data. We can implement code to achieve this using a **do ... while** loop. Consider the program segment given below.

```
do {
    printf("Enter marks (0..100): ");
    scanf("%d", &marks);
} while (marks < 0 || marks > 100);
```

This loop prints a message and accepts marks from the keyboard as long as the marks entered are incorrect (either less than zero or greater than 100). The control leaves the loop only when the user enters valid marks (in the range 0 to 100). The output is shown below, where the user has entered two

incorrect values (120 and -20) before entering the correct value (90).

```
Enter marks (0..100): 120
Enter marks (0..100): -20
Enter marks (0..100): 90
```

#### c) Determine sum and average of several numbers entered from the keyboard

In Program 6.2, we used a **for** loop to determine the sum of several numbers. The program segment given below does the same thing using a **do ... while** loop and then determines the average of the given numbers.

```
printf("How many numbers? ");
scanf("%d", &n);

sum = 0.0;
i = 0;
do {
    scanf("%f", &x);
    sum += x;
    i++;
} while (i < n);

avg = sum / n; /* sum is assumed to be of type float */
```

#### d) Reverse digits in an integer number

The program segment given below obtains a number by reversing the digits in the given number.

```
scanf("%d", &num);
rev_num = 0;
do {
    digit = num % 10;
    rev_num = rev_num * 10 + digit;
    num /= 10;
} while(num > 0);

printf("reverse number = %d", rev_num);
```

This program segment first accepts a number **num** from the keyboard and initializes variable **rev\_num** to zero. Then it uses a **do ... while** loop to separate the digits in **num** and construct number **rev\_num** in reverse order of digits, one digit at a time. Observe carefully the statement used to obtain the reversed number:

```
rev_num = rev_num * 10 + digit;
```

In this statement, the current value of `rev_num` is first multiplied by 10 and the value of `digit` is added to it. The steps in the formation of the reverse number are illustrated below for `num` = 1234.

Loop iteration	num (in loop beginning)	digit	rev_num	num (at the end of loop)
			0	
1	1234	4	4	123
2	123	3	43	12
3	12	2	432	1
4	1	1	4321	0

The code to determine the reverse number can also be written using the `while` loop. The code given below avoids use of variable `digit`. The I/O statements are omitted to save space.

```
rev_num = 0;
while (num > 0) {
    rev_num = rev_num * 10 + num % 10;
    num /= 10;
}
```

This code can be made more concise using the `for` loop as shown below.

```
for (rev_num = 0; num > 0; num /= 10)
    rev_num = rev_num * 10 + num % 10;
```

## 6.4 Advanced Concepts

In this section, some advanced concepts related to loops are presented that allow us to write concise and powerful code. A beginner may skip this section and return to it later after studying more important and useful concepts presented in subsequent chapters.

### 6.4.1 Variations in `for` Loops

C is a very powerful language. It permits several variations in the basic `for` loop syntax. These include

1. Omission of control expressions
2. Setting up an endless loop
3. Use of multiple control expressions (comma-separated list)
4. Use of involved or complex control expressions

5. Use of a variable as final expression
6. Use of a null statement as loop body
7. Modification of loop variable within the loop body.

These variations are discussed below.

### Omitting Control Expressions

Recall that a **for** loop has three control expressions, namely, *initial* expression, *final* expression and *update* expression. The C language allows us to omit any of these control expressions. However, the semicolons separating these expressions cannot be omitted. For example, if the initial expression is omitted, the **for** loop can be written as

```
for ( ;final_expr; update_expr)
    statement;
```

#### Example 6.5 Factorial of an integer number: Omitting control expressions in the **for** loop

Consider the code given below to determine the factorial of a given number.

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

fact = 1;
for ( ; n > 1; n--)      /* initial expression is omitted */
    fact *= n;

printf("Factorial = %d", fact);
```

In this example, a **for** loop is used to determine the factorial of a given number, *n*. The variable *n* itself is used as the loop variable. As the value of variable *n* will be initialized when the **scanf** statement is executed, the initial expression in the **for** loop is not required and is omitted. Note that we have avoided a separate loop variable, reducing the memory requirement of this program. However, the original value of variable *n* is not available after the loop is executed. This approach is particularly suitable in a function that calculates and returns the value of factorial of a given number.

We can also move the update expression (*n--*) in above **for** loop inside the body of the loop as shown below:

```
fact = 1;
for ( ; n > 1; ) {      /* initial and final expressions are omitted */
    fact *= n;
    n--;
}
```

Of course, a **while** loop is more readable when both the initial and final expressions are omitted in a **for** loop:

```
fact = 1;  
while (n > 1) {  
    fact *= n;  
    n--;  
}
```

Note that the above **for** loop (as well as **while** loop) can be written in a concise way by combining the statements in the body of the loop using a decrement operator as shown below:

```
for ( ; n > 1; )  
    fact *= n--;
```

```
while (n > 1)  
    fact *= n--;
```

### Endless Loop

If we omit all the control expressions in the **for** loop, we get an **endless loop**:

```
for (;;)  
    statement;
```

As the name indicates, an endless loop allows us to execute a statement or group of statements repeatedly forever. Such loops are required in many programming situations. For example, the operating system in a computer is a program that executes forever, unless of course we shut down the machine. Many game programs are also written using an endless loop. We can use either the **break** statement or the **exit** standard library function to stop execution of such a loop. This concept is covered in next chapter.

Now consider the **for** loop given below where the loop variable **i** is assumed to be of some integral type (such as **char**, **int**, etc.).

```
for (i = 1; i > 0; i++)  
    printf("%d ", i);
```

This surely appears to be an infinite loop as the value of loop variable **i** goes on continuously increasing in the update expression and the final expression (**i > 0**) will always evaluate as true.

However, you will be surprised to know that this is not an endless loop! This is due the limited range of integral types. For example, while using 2-byte **int** type in Turbo C/C++, the maximum value of the loop variable is 32767. When this value is incremented, the loop variable assumes value as -32768, for which the final expression evaluates as false and the loop terminates.

### Using Multiple Control Expressions

The C language allows us to write very powerful and concise `for` loops in which each control expression may be a comma-separated list of expressions. This feature is usually used in the initial and final expressions of the `for` loop as shown below:

```
for( initial_expr1, initial_expr2, ... ; final_expr ; update_expr1, update_expr2, ... )  
    statement;
```

The execution of this loop is very similar to that of the simple `for` loop. Initially, all initialization expressions are evaluated in the given sequence. Then the final expression is evaluated. If it evaluates as *true*, the *statement* included in the `for` loop is executed. After this, all the update expressions are evaluated in the given sequence and the final expression is evaluated again. This will continue as long as the final expression evaluates as true. The control leaves the loop when the final expression evaluates as false.

Finally, note that the control expressions in `for` loops need not be simple expressions. We can use more involved control expressions that may contain various operators and function calls.

#### Example 6.6 Using multiple control expressions in the `for` loop

a) The `for` loop given below determines the sum of *n* numbers entered from the keyboard.

```
for(i = 0, sum = 0.0; i < n; i++) {  
    scanf("%f", &num);  
    sum += num;  
}
```

Observe that the initialization part consists of two expressions (*i* = 0 and *sum* = 0.0) separated by a comma operator. The variable *sum* can also be updated in the update expression of the `for` loop to make the code concise (but sacrificing readability), as shown below.

```
for(i = 0, sum = 0.0; i < n; i++, sum += num)  
    scanf("%f", &num);
```

b) Now consider that we wish to print letters A to Z in the first column and letters Z to A in the second column, as shown below:

A Z  
B Y  
C X  
:  
Z A

In this example, we have to print 26 rows. Thus, we setup a `for` loop to perform 26 iterations. Let us use variables `ch1` and `ch2` (of type `char`) to store two characters printed on each line. We have to

initialize these variables to 'A' and 'Z', respectively, before the `for` loop. Also, the values of these variables should be updated before we proceed to the next iteration. Thus, we can use a simple `for` loop to write the code as shown below.

```
ch1 = 'A';
ch2 = 'Z';
for(i = 0; i < 26; i++) {
    printf("%c %c\n", ch1, ch2);
    ch1++;
    ch2--;
}
```

However, we can rewrite this code very concisely using multiple expressions in the initial and final expressions of the `for` loop as shown below:

```
for(i = 0, ch1 = 'A', ch2 = 'Z'; i < 26; i++, ch1++, ch2--)
    printf("%c %c\n", ch1, ch2);
```

Note that we can further simplify this code by eliminating the loop variable `i` and using either `ch1` or `ch2` in the final expression to control the number of iterations as shown below.

```
for(ch1 = 'A', ch2 = 'Z'; ch1 <= 'Z'; ch1++, ch2--)
    printf("%c %c\n", ch1, ch2);
```

## Using Involved Control Expressions

The C language permits the use of involved or complex expressions as control expressions in a `for` loop. This enables us to write concise code. For example, consider that we wish to print odd numbers in a given range  $m$  to  $n$ . One simple approach presented in the next chapter uses an `if` statement within the `for` loop. In this approach, the loop variable of the `for` loop takes consecutive values from  $m$  to  $n$  and the `if` statement used within the loop tests each value and prints it if it is an odd number.

Another efficient approach is to set up the `for` loop so that the loop variable takes only odd values in the range  $m$  to  $n$ . This can be achieved by initializing the loop variable to either  $m$  or  $m+1$  depending on whether  $m$  is odd or even and then increment the values of the loop variable by 2. The code given below uses a conditional expression to initialize the value of the loop variable `num`.

```
printf("Odd numbers in the range %d to %d:\n", m, n);
for (num = (m % 2 == 1 ? m : m + 1); num <= n; num += 2)
    printf("%d ", num);
```

Observe how the redundant parentheses surrounding the conditional operator in the initial expression greatly improve the readability of this code.

## Using a Variable as Final Expression

We may often have to execute the statements within the loop as long as the value of the loop variable is non-zero. In such situations, we can simply use the loop variable as a final expression. Consider, for example, the **for** loop given below.

```
for (i = 10; i; i--)
    printf("%d ", i);
```

The loop variable **i** assumes decreasing integer values starting from 10. The **printf** statement is executed as long as the value of variable **i** is true, i. e., non-zero. Thus, the loop prints integer numbers from 10 to 1.

Consider another **for** loop given below that determines the factorial of an integer number **n** entered from the keyboard.

```
scanf("%d", &n);
for (fact = 1; n; n--)
    fact *= n;
```

## Null Statement as Loop Body

A **null statement** is an empty statement. It consists of only the terminating semicolon as shown below:

```
;
```

This statement does nothing. It is used as a placeholder where a statement is expected as per C syntax but no operation is to be performed as per implementation. A **for** loop containing a null statement as its body is usually written as shown below:

```
for (initial_expr; final_expr; update_expr)
    ;
```

Observe that a null statement is written on a line by itself to clearly indicate its presence and the fact that this **for** loop has an empty body. Such a loop is particularly useful when we wish to locate a particular position in an array or string.

Beginners often make a mistake of writing a semicolon at the end of the **for** loop:

```
for (initial_expr; final_expr; update_expr); /* Oops! */
    statement;
```

This leads to one of the very difficult to trace bug as C compilers do not report an error in such situations. This code is interpreted as shown below:

```
for (initial_expr; final_expr; update_expr)
```

```
;  
statement;
```

Thus, the **for** loop is interpreted to have a null statement as its body and the *statement* which was intended to be the body of the **for** loop is now outside it. Execution of this code causes the null statement to be repeatedly executed. In simple loops, usually nothing happens as a result of this execution except the update of the loop variable. After the **for** loop is executed, the *statement* will be executed only once.

### Example 6.7 Null statement as loop body

a) Consider the **for** loop given below:

```
for(i = 1; i <= 10; i++);  
    printf("%d", i);
```

What is the output of this loop? If you think that it will print integer numbers from 1 to 10, have a closer look at the code.

Did you notice the semicolon after the line containing the **for** loop. As just explained, this code is equivalent to the following code:

```
for(i = 1; i <= 10; i++)  
;  
printf("%d", i);
```

Thus, the **for** loop executes the null statement 10 times. After the execution of the loop, the value of the loop variable *i* is 11 which is printed by the **printf** statement.

b) Now consider the code given below to determine the sum of integer numbers from 1 to *n*.

```
sum = 0;  
for(i = 1; i <= n; i++)  
    sum += i;
```

As the **for** loop allows multiple expressions to be written as initial, final and update expressions, we can rewrite this code concisely (of course, sacrificing readability) as follows:

```
for(i = 1, sum = 0; i <= n; i++, sum += i)  
;
```

c) Consider that we wish to print a given string in reverse order. Thus, the string "Hello" should be printed as olleH. Recall that a string is a null-terminated sequence of characters stored in an array of type **char**, with the index of the first character being 0. Also, a character at index *i* in string **str** is

accessed as `str[i]`.

To print a string in reverse order, we have to first locate the position of the null terminator. This can be done using a `for` loop that includes a null statement as shown below.

```
for(i = 0, str[i] != '\0; i++)
;
```

Here, the value of loop variable `i` is incremented as long as `str[i]`, i. e., character at position `i` is not equal to null ('`\0`'). Thus, when the execution of this loop is over, the loop variable `i` will contain the position of null terminator. This loop can be made more concise by omitting comparison with null character as shown below.

```
for(i = 0; str[i]; i++)
;
```

Once the null character is located, we can print the string in reverse order using another `for` loop as shown below.

```
for(i--; i >= 0; i--)
putchar(str[i]);
```

Observe that the initial expression in the `for` loop decrements the value of `i` so that it points to the character before the null terminator.

### Modifying the `for` Loop Variable within Loop Body

The loop variable in a `for` loop is initialized in the initial expression and updated in the update expression. The C language allows the values of the `for` loop variables to be modified in places other than the update expression. However, such modifications lead to code that is very difficult to understand and may also lead to incorrect number of loop iterations. Hence, programmers should not modify loop variable in places other than update expression. One common possibility for introduction of such errors in our programs is when we replace a `while` loop with a `for` loop and forget to delete the statement that updates the loops variable.

#### Example 6.8 Modifying the `for` loop variable within the loop body

a) Consider the `for` loop given below.

```
for (i = 1; i < 10; i++) {
    printf("%d ", i);
    i++;
}
```

From the control expressions, observe that the loop variable **i** takes values as 1, 2, ... 9.

However, note that besides the update expression, the loop variable **i** is also incremented after it is printed inside the loop. Thus, the loop variable assumes odd values from 1 to 9 which are printed as shown below:

1 3 5 7 9

Now consider another **for** loop given below.

```
for (i = 1; i < 10; i++)
    printf("%d ", ++i);
```

In this loop, the value of loop variable **i** is first incremented in the **printf** statement and then printed. The value of **i** is again incremented in the update expression. Thus, the loop prints even values from 2 to 8 as shown below:

2 4 6 8

b) Consider the **for** loop given below:

```
for (i = 1; i < 10; i++)
    printf("%d ", i--);
```

This **for** loop is set up such that the loop variable takes values from 1 to 9. However, note that during the execution of this loop, the value of **i** is decremented in the **printf** statement and then incremented in the update expression. Thus, its value is always 1 in the test expression and the loop is an endless loop. It continuously prints value 1.

c) Carefully study the following code segment to determine the sum of **n** numbers entered from the keyboard and comment on what will happen when this code is executed.

```
scanf("%d", &n);
sum = 0;
for (num = 0; num < n; num++) {
    scanf("%d", &num);
    sum += num;
}
printf("sum = %d\n", sum);
```

As seen from the loop control expressions, the loop has been set up to process **n** numbers. However, observe that the **scanf** statement within the loop also modifies the value of loop variable **num**. Thus, the execution of the loop depends on the values entered from the keyboard. The execution of the loop continues as long as the user enters values less than equal to **n-2** (and not **n-1**) as the value of **num** is subsequently incremented in the update expression.

d) Now consider an obscure program segment given below to calculate the factorial of an integer

number n.

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

fact = 1;
for (i = n; i > 0; i++) {
    fact *= i;
    i -= 2;
}
printf("%d! = %d ", n, fact);
```

The control expressions in the **for** loop might mislead you. However, observe that the loop variable **i** is decremented by 2 in the loop body and incremented by 1 in the loop update expression. Thus, it is effectively decremented by 1 and assumes values as **n**, **n-1**, **n-2**, ... **1**. Thus, the program segment correctly calculates the factorial of a given integer number.

### 6.4.2 Variations in the **while** and **do ... while** Loops

Some of the concepts discussed in the last section are also applicable to the **while** and **do ... while** loops. These include endless loop, use of null statement as loop body, modifications to the values of the loop variable (in test expression) and use of complex expression as well as simple variable as control expression.

#### Endless Loop

To set up an **endless loop** using the **while** and **do ... while** loops, we use a test expression that always evaluates as true. Although any non-zero value can be used as test expression to set up such a loop, the convention is to use integer value 1 as shown below:

```
while (1)
    statement;
do
    statement;
while (1);
```

Note that it is an error to omit the test expression in a **while** and **do ... while** loops. We can use an endless loop to handle situations involving unknown number of iterations. The exit from such a loop is usually provided using a **break** statement. The **break** statement is discussed in the next chapter.

#### Null Statement as Loop Body

As in the case of the **for** loop, the C language permits a **null statement** to be used as the body of **while** and **do ... while** loops. For example, we can locate the null terminator of string **str** using

the `while` loop given below.

```
i = 0;  
while (str[i] != '\0')  
i++;
```

If we perform the decrement operation in the test expression itself, we get a `while` loop with a null statement as its body as shown below:

```
i = 0;  
while(str[i++] != '\0')  
;
```

We have studied the problem caused by a semicolon after the closing parenthesis in the `for` loop. A similar problem occurs when a semicolon is written after the closing parenthesis in the `while` loop as shown below:

```
while ( test_expr ); /* Oops! */  
      statement;
```

This code is equivalent to the following:

```
while ( test expr )  
;  
      statement;
```

Thus, the `while` loop repeatedly executes the null statement and the contained statement is executed only once after the execution of the loop. It is easy to understand that such a problem does not arise in a `do ... while` loop.

### Modifying Loop Variable in Test Expression

Consider the `while` loop given below. Do you think this code will correctly calculate the factorial of a given integer number?

```
scanf ("%d", &n);  
fact = i = 1;  
while (i++ < n)  
    fact *= i;  
printf("n! = %d\n", fact);
```

Note that the loop variable `i` is updated in the test expression using the postfix increment operator. Thus, the loop body is executed for the values of loop variable from 1 to `n`-1 (in the test expression). The corresponding values of the loop variable are from 2 to `n` within the body of the loop. Thus, the

loop correctly evaluates the product of integers from 2 to n as desired. However, observe that the code is very difficult to understand. We should of course avoid this coding style.

To understand the difficulties caused by such coding style, analyze another code segment given below to determine the factorial of a given number for its correctness.

```
scanf ("%d", &n);
fact = 1;
while (n--)
    fact *= n;

printf("Factorial = %d\n", fact);
```

### Using a Variable as Test Expression

We may often have to execute the statements within the loop as long as the value of the loop variable is non-zero. In such situations, we can simply use the loop variable as a test expression. For example, consider the solution of sum of digits of a non-negative integer number (Program 6.5) in which the execution of the `while` loop is continued as long as the value of `num` is non-zero. We can rewrite this code by replacing test expression `num > 0` with `num` as shown below.

```
sum = 0;
while (num) /* means num != 0 */
{
    digit = num % 10;
    sum += digit;
    num /= 10;
}
```

### Using Involved Test Expressions

The test expressions in loops presented so far are simple relational expressions. We can write more involved test expressions that use other operators such as arithmetic, logical, bitwise, assignment, conditional expression, etc. This also applies to the `for` loop.

#### Example 6.9 Using involved control expressions in loops

##### a) Read a line of text and display it

Consider the code given below to read a line of text from the keyboard and display it.

```
ch = getchar();
while (ch != '\n') {
    putchar(ch);
    ch = getchar();
```

```
}
```

Observe that the `getchar` statement appears twice and that the value of variable `ch` read by these statements is tested in the test expression. Thus, we can write this code concisely if we combine the `getchar` function call with the test expression, as shown below.

```
while ((ch = getchar()) != '\n')
    putchar(ch);
```

Note that as the assignment operator (`=`) has a lower precedence than the inequality (`!=`) operator, the assignment expression `ch = getchar()` is enclosed in a pair of parentheses. Thus, the test expression in the loop first reads a character from the keyboard and assigns it to variable `ch` and then compares it with the newline character. If the character read (and assigned to `ch`) is not equal to newline, it is printed on the screen. Otherwise, control leaves the loop.

#### b) Add numbers until a negative or zero number is encountered

Consider the code given below (from Example 6.2b) in which a `while` loop is used to add numbers entered from the keyboard as long as they are positive.

```
printf("Enter positive numbers (0 or -ve number to stop):\n");
sum = 0;
scanf("%d", &num);
while (num > 0) {
    sum += num;
    scanf("%d", &num);
}

printf("Sum = %d ", sum);
```

This code uses two `scanf` statements similar to the use of `getchar` statements in the previous example. Note that due to the basic difference in the use of `getchar` and `scanf`, we cannot use the technique employed in the previous example to write this code concisely. We can, however, use a comma operator to write the `scanf` function call and the test `num > 0`, as shown below to achieve the same effect:

```
printf("Enter positive numbers (0 or -ve number to stop):\n");
sum = 0;
while (scanf("%d", &num), num > 0)
    sum += num;

printf("Sum = %d ", sum);
```

Recall that comma separated expressions are evaluated strictly in the left-to-right order. Thus, the `scanf` function is called first to read the value of `num` which is then compared with zero. The value of the comma-separated test expression in the `while` loop is the value of the rightmost expression, i. e., `num > 0` as desired. Thus, the `while` loop reads a number and adds it to sum as long as the entered number is greater than zero. However, note that this style of programming should be avoided as may make programs difficult to read.

## Exercises

1. Determine the values of control variables and number of iterations in the following `for` loops.
  - a. `for (i = 0; i < 10; i += 2)`
  - b. `for (i = 10; i >= -10; i = i - 1)`
  - c. `for (x=1.0; x >= 0.5; x-= 0.1)`
  - d. `for (j = 1; j < 1000; j *= 2)`
  - e. `for (k = 0; k < 50; k = k + 5)`
  - f. `for (i = 10000; i >= 1; i /= 10)`
  - g. `for (i = 'A'; i <= 'f'; i++)`
  - h. `for (ch = 'A'; ch != 'F'; ++ch)`
2. Determine the output of the following program segments:
  - a. `for (j = 0; j < 10; k++)`  
`printf("%d ", j);`
  - b. `for (i = 5; i <= 10; i += 1)`  
`printf("%d^2 = %d\n", i, i*i);`
  - c. `int num = 1234;`  
`for (x = 1; num > 0; num /= 10)`  
`x *= num % 10;`  
`printf("x = %d", x);`
  - d. `sum = 0;`  
`for (k = i = 1; k < 10; k += 3)`  
`sum += i;`  
`printf("Sum = ", sum);`
  - e. `s = 0;`  
`for (i = '0'; i < '5'; i++)`  
`s += i;`  
`printf("%d", s);`

```
f. for (k = 0; ++k < 10; k++)
    printf("%d ", ++k);
printf("%d ", ++k);
```

3. Write equivalent **while** loops for the following **for** loops:

- ```
for (ch = 'a'; ch <= 'z'; ++ch)
    putchar(ch);
```
- ```
for (x = 2.0; x >= 1.0; x -= 0.2)
printf("x = %5.2f\n", x);
```
- ```
sum = 0;
for (i = 0; i < 10; i++) {
    scanf("%d", &a);
    sum = a * a;
}
printf("Sum: %d", sum);
```
- ```
for(ij = 1024; ij > 0; ij /= 2) {
    printf("ij = %d", ij);
    printf("ln(ij)= %d", log(ij));
    printf("\n");
}
```

4. Identify errors in the program segments containing **for** loops given below. Also rewrite these program segments correctly.

- ```
// print numbers from 1 to 10
for (i = 0; i < 10, i = i - 1)
    printf("i = %d\n", i);
```
- ```
/* print characters from A to Z */
char ch;
for(ch = A; ch < Z; ch++)
    putchar('ch');
```
- ```
/* print a line of dash */
int i;

for(i = 0, i < n, i++)
print("-");
```
- ```
/* det sum of numbers from 1 to n */
for(num =1; num >= n; ++num)
    scanf("%d", num);
    sum = num + sum;
printf("\nsum: %d", sum);
```

```

}

e. /* Factorial of a number */
int n, fact;

scanf("%d", &n);
for (i = n; i >= 0; i++)
    fact *= i;
printf("Factorial = %d", fact);
}

f. /* Sum of squares of diffs of consecutive numbers */
double num1, num2;
int i, n;

scanf("%d", &num1);
for (i = 0; i < n; i++) {
    scanf("%f", &num2);
    sum += (num1 - num2)²;
    printf("Sum = %f", sum);
}

```

5. Identify the purpose of the following program segments:

- `for(c = 'A'; c <= 'Z'; c++)
 printf("%d ", c);`
- `scanf ("%d", &num);
for (f = 1; num >= 2; num--)
 f *= num;`
- `int n, m;
scanf("%d", &n);
for (m = 0; n > 0; n /= 10)
 m = m * 10 + n % 10;
printf("%d", m);`
- `int num = 1234;
for (x = 1; num > 0; num /= 10)
 x *= num % 10;
printf("x = %d", x);`

6. Several program segments using `while` loops are given below. Identify the errors in these program segments and rewrite them correctly.

- `/* det sum of 10 numbers */
int num, i, sum;`

```
while (i < 10) {
    scanf("%d", &num);
    sum += num;
}
printf ("Sum = ", sum);

b. /* capital letters in reverse order */
char x = Z;

while (x >= A) {
    printf(x);
    x--;
}

c. /* sum of digits of a no. */
int num;

digit = 1;
while (digit < = 5)
    digit = num / 10;
    digit_sum += digit;
    digit++;

d. /* Generate 1 digit random numbers
until their sum exceeds 100 */
int num, sum;

printf("Generated numbers: ");
while(sum >= 100)  {
    num = random() % 10;
    sum = sum + num;
}
printf("Their sum: %d", sum);

e. /* add nos till 0 is entered */
int num, sum;

sum = 0;
while(num != 0) {
{    scanf("%d", &num);
    sum += num;
    printf("Sum = %d", sum);
}

f. /* Determine factorial of number */
```

```

int n, fact;
scanf("%d %d", &n, &fact);
fact = 0;
while (n >= 0) {
    fact = fact * n;
    n--;
}
printf("n! = \n", n!);

```

7. Identify errors in the program segments given below and rewrite them correctly.

- /\* print numbers from 1 to n \*/

```

i = 1;
do
    printf("%d ", i);
    i++;
while i < n;

```
- /\* Print Fibonacci series: n terms \*/

```

int p = 0, q = 1;
printf("0 1");
do {
    sum = p + q;
    p = sum;
    sum = q;
}
printf("%d", sum);

```
- /\*test if a num is divisible by 3\*/
/\* first det sum of digits \*/
sum = 0;
do {
 sum += num % 10;
 num /= 10;
} while (num > 0);

/\* test digit sum \*/
if (sum % 3 != 0)
 printf("Not");
printf("Divisible by 3");

8. Write **for** loops for the following situations:

- Calculate the sum of numbers from 1 to  $n$ , where  $n$  is entered from the keyboard.
- Print uppercase and lowercase letters, as in AaBbCc ... Zz.
- Calculate the sum of squares of  $n$  given numbers entered from the keyboard.

- d. Reverse a given integer number.
- e. Calculate the sum of squares of differences of consecutive numbers entered from the keyboard, e.g., for numbers 1, 4, 5 and 3, the program should calculate  $(4 - 1)^2 + (5 - 4)^2 + (3 - 5)^2$
- f. Accept n numbers from the keyboard and calculate their mean and standard deviation.
- g. Write complete programs for the following problems using suitable loop.
  - a. Print alternate uppercase letters starting with letter A.
  - b. Print a table of values of  $\sin(x)$ ,  $\cos(x)$  and  $\tan(x)$  for values of  $x$  from  $0^\circ$  to  $90^\circ$  in steps of  $5^\circ$ .
  - c. Test whether the sum of digits of a given number is odd or even.
  - d. Test whether a given number is a palindrome or not, i. e., if it is equal to its reverse number or not.
  - e. Evaluate the value of  $\sin(x)$  using the first  $n$  terms in its series expansion.

## Exercises (Advanced Concepts)

1. Write equivalent **while** loops for the following **for** loops:

```

a. for (i = 10; i; i--)
    printf("%d ", i);

b. for ( ; ; )
    printf("Good bye\n");

c. prod = 1;
    for (i = num; i > 0; --i)
        prod *= i;
    printf("%d ", prod);

d. i = 10;
    for ( ; i > 0; i--)
        printf(" %d", i);
  
```

2. Determine the output of the following program segments:

```

a. for(z = 10; z < 100; z++)
    printf("%d ", z);

b. for (m = 10; m; --m)
    printf("%d ", m);

c. for(i = 0 ,j=10;
    i+j <= 20;
    i+=3, j--)
    printf("%d ", i*j);
  
```

d. `for (a=5, b=5; a && b; a--, b++)`  
{  
    `printf("%d ", a * b);`  
}  
  
e. `sum = 0;`  
    `for (i = 0; i < 10; i++)`  
        `scanf("%d", &num);`  
        `sum += num;`  
    `printf("sum = %d", sum);`  
  
f. `int n;`  
    `for (n = 10000; n < 30000;`  
        `n -= 10000)`  
    `printf("%d ", n);`  
  
g. `int i = 0;`  
    `do`  
        `printf("Hello %d ", i);`  
    `while(i++ < 5);`  
  
h. `int i = 5;`  
    `while(i--)`  
        `printf("%d ", i);`

3. Identify errors in the program segments containing `for` loops given below. Also rewrite these program segments correctly.

a. `x = 10;`  
    `for (x >= 0; x -= 0.1)`  
        `printf("%f ", sqrt(x));`  
  
b. `for (i,j = 0; i < 5; i++, j--)`  
        `printf("Hello");`  
  
c. `scanf ("%d", &n);`  
    `for (sum = 0; n > 0; n /= 10)`  
        `digit = n % 10;`  
        `sum += digit;`  
  
d. `/* print sequence 1,2,4,...,1024 */`  
    `char i, j;`  
    `for (i=0, j=1; i<=10; i++, j*=2)`  
        `printf("%d,", j);`  
  
e. `/* count digits in a number */`  
    `int num, digits = 0, i;`  
    `scanf("%d", &num);`

```

for ( ; num > 0; num /= 10)
    digit++;

printf("Sum: %d", &sum);

f. for()
{
    printf("Repeat, repeat,
           and repeat ...
           Till you succced!");
}

g. /* Sum and avg of given numbers */
int count, sum;

sum = count = 0;
for(    count ≤ n; count+1) {
    scanf(&num);
    sum = sum + num;
    avg = sum / n;
    printf(sum and avergae);
}

h. /* print multiplication table
for a given number */
int i = 1;
scanf("%d", &n);
for( ; i < 10; i = i + 1) {
    printf("n x i = ", n x i);
    n++;
}

```

i. Identify errors in the program segments given below and rewrite them correctly.

```

a. while () {
    printf("Endless loop");
}

b. i = 5;
while(i < 10) do
    printf(%d , i);

c. /* Table of 2^n, for n=1 to 10*/
int k = 1;
do printf("%d %d", power(2, k));
while(++k <= 10);

```

d. /\* Print products of pairs: 10 x 20,  
11 x 19, 12 x 17, 13 x 16 and 14 x 15  
Int i = 10, j = 20;  
do printf("%d", ++i x --j);  
while (i > j);

# 7 Nested Control Structures

The C language allows the control structures to be nested, i. e., written within one another. In this chapter, we study how various control structures can be nested. First we present the *nested if statements* (`if` statements within `if` statement) followed by nesting of `if` statements within loops. Then nested loops and nested control structures involving the `switch` statement are presented. Finally, the loop interruption statements, namely, `break` and `continue`, are discussed.

In the *Advanced Concepts* section, a discussion on how the statements containing logical AND (`&&`) and logical OR (`||`) operators can be written equivalently using nested `if` and `if-else-if` statement, respectively, is presented. Then the nesting of conditional expression operators is discussed.

## 7.1 Nested `if` Statements

The C language allows ***nested if statements*** in which the `if` block and/or `else` block of an `if` statement contains `if` or `if-else` statements. The inner `if` statement(s) may in turn contain other `if` statements and so on.

### 7.1.1 Two-Level Nested `if` Statements

Consider the general form of the `if-else` statement given below.

```
if (expr)
    statement1;
else statement2;
```

The general form of a **two-level *nested if statement*** obtained by replacing both `statement1` and `statement2` in the above form with the `if-else` statement itself is given below.

```

if (expr1)
    if (expr2)
        statement1 ;
    else statement2 ;
else
    if (expr3)
        statement3 ;
    else statement4 ;
}

if (expr1) {
    if (expr2)
        statement1 ;
    else statement2 ;
}
else {
    if (expr3)
        statement3 ;
    else statement4 ;
}

```

Observe how the inner **if-else** statements are indented (shifted to the right) to improve readability. Also note that since the *if* block and *else* block of the outer **if-else** statement comprise a single **if-else** statement, they need not be included within braces. However, we can surround inner **if-else** statements with braces to improve readability as illustrated in the right-hand side form. As explained earlier, the use of such braces also prevents errors when we add one or more statements to these *if* and *else* blocks.

The flowchart for this general two-level *nested if* statement is given in Fig. 1.9a. Observe that only one of the four alternative statements (*statement1*, *statement2*, *statement3*, *statement4*) will be executed depending on the values of the test expressions *expr1*, *expr2* and *expr3*. Also note that these alternative statements can be any valid C statement such as a simple statement (assignment, function call, etc.), control statement or compound statement.

The execution of this statement is as follows: Initially, *expr1* is evaluated. If it is *true* (non-zero), *expr2* is evaluated and depending on its value, either *statement1* or *statement2* is executed, i. e., if *expr2* is true, *statement1* is executed; otherwise, *statement2* is executed. After execution of either *statement1* or *statement2*, the control leaves *nested if* statement. Note that *expr3* is not evaluated in this case.

On the other hand, if *expr1* evaluates as *false*, *expr3* is evaluated and depending on its value, either *statement3* or *statement4* is executed. After execution of either statement, control leaves the *nested if* statement. Note that *expr2* is not evaluated in this case.

### Example 7.1 Determine the maximum of three numbers

A program to determine the maximum of three numbers, *a*, *b* and *c*, is given in Program 5.1 in which we first determined the larger of two numbers (*a* and *b*) which was then compared with the third number, *c*. Here, another technique is presented, the algorithm for which is given below.

```

if (a > b)
    max = larger of a and c
else max = larger of b and c

```

Thus, if *a* is greater than *b*, the larger of *a* and *c* is the maximum number; otherwise, the maximum number is the larger of *b* and *c*. If we use an **if-else** statement to determine the larger of two numbers in the above algorithm, we get a two-level *nested if* statement to determine maximum of three

numbers as shown below.

```
/* determine maximum of 3 numbers */
if (a > b) {
    if (a > c)
        max = a;
    else max = c;
}
else {
    if (b > c)
        max = b;
    else max = c;
}
```

### Other Forms of Two-Level Nested **if** Statements

Consider that either the *if* block or the *else* block in the general form of the **if-else** statement is replaced with an **if-else** statement, but not both. Now we have two other forms of two-level *nested if* statement, as shown below.

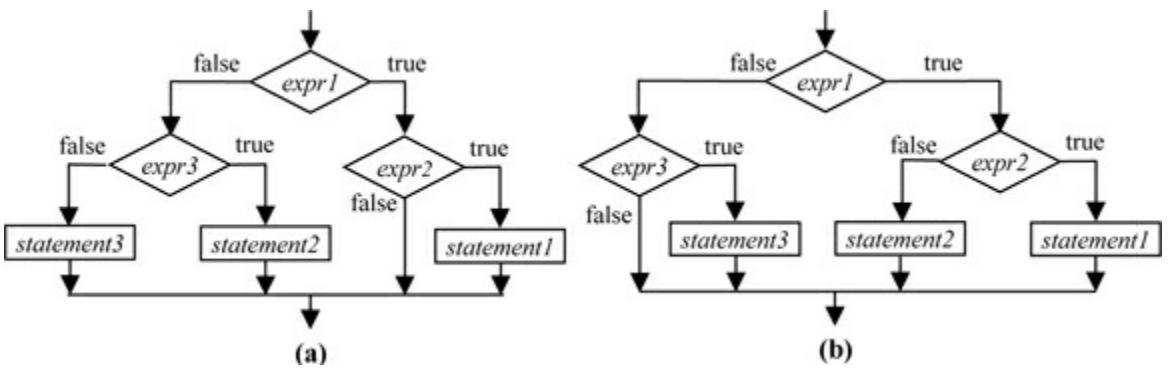
<pre>if (expr1)     if (expr2)         statement1 ;     else statement2 ; else     statement3 ;</pre>	<pre>if (expr1)     statement1 ; else     if (expr2)         statement2 ;     else statement3 ;</pre>
---	---

The flowcharts for these statements are given in Fig. 1.9b and Fig. 1.9c, respectively. The second form which contains an **if-else** statement only in the **else** part is called as **if-else-if** statement and is explained shortly in detail.

As the *else* block in an **if-else** statement is optional, we can omit the *else* blocks in the inner **if** statements to obtain several additional forms of the two-level *nested if* statement. However, while writing such statements we need to remember the following **rule of nested if statements**: *An else clause is associated with the nearest if statement that is not already associated with an else clause.*

If we omit the **else** clause in the inner **if** statements of a general two-level *nested if* statement, we get two other forms given below and illustrated in Fig. 7.1.

<pre>if (expr1) {     if (expr2)         statement1 ; } else     if (expr3)         statement2 ;     else statement3 ;</pre>	<pre>if (expr1)     if (expr2)         statement1 ;     else statement2 ; else     if (expr3)         statement3 ;</pre>
--	--



**Fig. 7.1** Flowcharts for nested if statements with else clause omitted for inner if statement

Note that braces are required around the inner **if** statement in the first format for correct interpretation that the **else**-clause of the inner **if** statement is omitted, as illustrated in Fig. 7.1a. In the absence of these braces, the **else** clause of the outer **if** statement becomes incorrectly associated with the inner **if** statement. Also note that we do not require such braces in the second case.

Two other forms of two-level *nested if* statements are given below. In the first form, the **else** clause of the outer **if** statement is omitted. Whereas, in the second form, the **else** clause of the outer and inner **if** statements are omitted.

```
if (expr1)
    if (expr2)
        statement1 ;
    else statement2 ;
```

```
if (expr1)
    if (expr2)
        statement1 ;
```

### Example 7.2 Character test – letter, vowel, consonant

The program segment given below uses a *nested if* statement to determine whether a given character is a letter or not. In addition, if the given character is a letter, it tests whether it is a vowel or consonant.

```
ch = getchar();
if (ch >= 'A' && ch <= 'Z' || ch >='a' && ch <= 'z') {
    /* ch is a letter, test for vowel */
    if (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U'
        || ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u')
        printf("%c is vowel\n", ch);
    else printf("%c is consonant\n", ch);
}
else printf("%c is not a letter\n", ch);
```

The outer **if** statement first tests whether the given character **ch** is a letter or not. If **ch** is a letter (either uppercase or lowercase), the condition

```
(ch >= 'A' && ch <= 'Z' || ch >='a' && ch <= 'z')
```

evaluates as true and inner **if-else** statement determines and prints whether **ch** is a vowel or a consonant; otherwise, the **else** block prints that the given character is not a letter.

As explained in Example 5.7e, we can simplify the test for a vowel by first converting the lowercase letters to uppercase as shown below.

```
ch = getchar();
if (ch == 'A' && ch <= 'Z' || ch >='a' && ch <= 'z') {
    /* ch is a letter, convert it to uppercase */
    if (ch >= 'a' && ch <= 'z')
        ch -= 'a' - 'A';

    /* test for vowel */
    if (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch=='U')
        printf("%c is vowel\n", ch);
    else printf("%c is consonant\n", ch);
}
else printf("%c is not a letter\n", ch);
```

Observe that the **if** statement for the conversion of a lowercase letter **ch** to uppercase can be more conveniently written before the outer **if** statement. This further simplifies the code and improves readability. The code given below uses the standard library functions, **toupper** for conversion of a character to uppercase and **isletter** to test for a letter.

```
ch = getchar();
ch = toupper(ch); /* if ch is lowercase, convert ch to uppercase */

if (isletter(ch)) {
    if (ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch=='U')
        printf("%c is vowel\n", ch);
    else printf("%c is consonant\n", ch);
}
else printf("%c is not a letter\n", ch);
```

### 7.1.2 Higher Level Nested **if** Statements

The C language allows *nested if* statements of much higher depth. If we replace one or more alternative *statements* in a two-level *nested if* statement with another **if-else** statement, we get a three-level *nested if* statement. The general form of this statement is obtained by replacing each alternative *statement* in a two-level *nested if* statement with an **if-else** statement. Further, if we replace any of the alternative statements in this general three-level *nested if* statement, we get a four-level *nested if* statement and so on.

### 7.1.3 if-else-if Statement

If we substitute the *else* block in an **if-else** statement with another **if-else** statement, we obtain a two-level **if-else-if statement**, as shown below.

```
if (expr1)
    statement1;
else
    if (expr2)
        statement2;
    else statement3;
```

```
if (expr1)
    statement1;
else if (expr2)
    statement2;
else statement3;
```

This statement is usually written as shown on the right-hand side in which the inner **if-else** statement is not indented. The flowchart for this statement is given in Fig. 1.9c. We can further substitute *statement3* in the above statement with an **if-else** statement to obtain a three-level nested **if-else-if** statement given below and illustrated in Fig. 1.11.

```
if (expr1)
    statement1;
else if (expr2)
    statement2;
else if (expr3)
    statement3;
else statement4;
```

The execution of this statement is as follows: Initially, *expr1* is evaluated and if its value is true (non-zero), *statement1* is executed; otherwise, *expr2* is evaluated. If *expr2* evaluates as true, *statement2* is executed; otherwise, *expr3* is evaluated and depending on its value either *statement3* or *statement4* is executed. Thus, only one of the alternative statements (*statement1* to *statement4*) is executed depending on the values of *expr1*, *expr2* and *expr3*. The control is then transferred to the statement following the **if-else-if** chain.

We can continue to substitute the statement in the last *else* block of an **if-else-if** statement with an **if-else** statement to obtain a higher-level **if-else-if** statement. Finally, note that the alternative statements included within an **if-else-if** chain can be any valid C statement such as simple, compound or control.

#### Example 7.3 Using if-else-if statements

##### a) Examination result in a single subject with data validation

The program segment given below accepts **marks** in a single subject and uses a *nested if* statement to determine the validity of **marks** and the result if the value of **marks** is valid.

```

scanf("%d", &marks);
if (marks >= 0) {
    if (marks <= 100) {
        if (marks >= 35)
            printf("Result: Pass\n");
        else printf("Result: Fail\n");
    }
    else printf("Error: Marks can't exceed 100\n");
}
else printf("Error: Marks can't be negative\n");

```

This code can be written in a more readable form using an **if-else-if** statement as

```

scanf("%d", &marks);

if (marks < 0)
    printf("Error: Marks can't be negative\n");
else if (marks > 100)
    printf("Error: Marks can't exceed 100\n");
else if (marks >= 35)
    printf("Result: Pass\n");
else printf("Result: Fail\n");

```

We can simplify this code by printing a single error message if the value of **marks** is invalid:

```

scanf("%d", &marks);

if (marks < 0 || marks > 100)
    printf("Error: Invalid marks\n");
else if (marks >= 35)
    printf("Result: Pass\n");
else printf("Result: Fail\n");

```

### b) Character classification – upper- or lowercase letter, digit, whitespace or other character

The program segment given below classifies a given character into one of the following categories: uppercase letter, lowercase letter, digit, white space and other character.

```

ch = getchar();

if (ch >= 'A' && ch <= 'Z')
    printf("Uppercase Letter\n");
else if (ch >= 'a' && ch <= 'z')
    printf("Lowercase Letter\n");

```

```

else if (ch >= '0' && ch <= '9')
    printf("Digit\n");
else if (ch == ' ' || ch == '\t' || ch == '\n' ||
         ch == '\v' || ch == '\r' || ch == '\f')
    printf("White space\n");
else printf("Other character\n");

```

Note that **ch** is a white space if it is one of the following characters: blank space, horizontal tab, newline, vertical tab, carriage return and form feed.

### c) Number of days in a given month

The program segment given below uses an **if-else-if** statement to determine the number of days in a given month (**mm**).

```

/* determine days in a given month */
if (mm == 1 || mm == 3 || mm == 5 || mm == 7 || mm == 8 ||
    mm == 10 || mm == 12)
    mdays = 31;
else if (mm == 4 || mm == 6 || mm == 9 || mm == 11)
    mdays = 30;
else if (mm == 2) {      /* check if year yy is leap */
    if (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0)
        mdays = 29;
    else mdays = 28;
}
else {
    printf("Invalid month.");
    mdays = 0;
}
printf("Days: %d\n", mdays);

```

The **if-else-if** statement has four alternative statements to handle the following cases: months having 31 days, months having 30 days, February having either 29 or 28 days depending on whether the year is leap or not and an invalid value of month.

The code for first two alternatives is straightforward and simply assigns the appropriate value (either 31 or 30) to **mdays**. The **if** statement for February assigns value 29 to variable **mdays** if **yy** is a leap year and 28 otherwise. The statements in the **else** clause print a message *Invalid month* and assign zero value to **mdays**. The value of **mdays** is printed after the **if-else-if** statement.

If we know that the value of **mm** is valid (1 to 12), we can greatly simplify this code by eliminating the test for months having 31 days, as shown below. Note the use of the conditional expression to simplify the code further.

```

/* determine days in a month, assume month is valid */
if (mm == 2)
    mdays = (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0)? 29: 28;
else if (mm == 4 || mm == 6 || mm == 9 || mm == 11)
    mdays = 30;
else mdays = 31;

printf("Days: %d\n", mdays);

```

### Testing Values in a Continuous Range

We can systematically test the value of a variable (or an expression) in a continuous range. Consider that we have to print the result and class obtained by a student in a single subject using the rules given below.

$75 \leq marks \leq 100$	Passed in first class with distinction
$60 \leq marks \leq 74$	Passed in first class
$50 \leq marks \leq 59$	Passed in second class
$35 \leq marks \leq 49$	Passed
$0 \leq marks \leq 34$	Failed

For simplicity, assume that the variable `marks` contains valid marks (0 to 100) obtained by a student in a single subject. We can determine the result in a straight forward way using several simple `if` statements, as shown below.

```

if (marks >= 75 && marks <= 100)
    printf("Passed in first class with distinction\n");
if (marks >= 60 && marks <= 74)
    printf("Passed in first class \n");
if (marks >= 50 && marks <= 59)
    printf("Passed in second class\n");
if (marks >= 35 && marks <= 49)
    printf("Passed\n");
if (marks >= 0 && marks <= 34)
    printf("Failed\n");

```

Each `if` statement tests whether the value of `marks` is within the desired range as specified in the problem statement. Note that this code works correctly even if we rewrite the `if` statements in any order. However, it requires more time for execution as at least one condition in each `if` statement is always evaluated. We can overcome this problem by converting these statements into an `if-else-if` chain by writing the `else` keyword before each `if` statement, except the first, and then removing the last `if` statement as shown below.

```

if (marks >= 75 && marks <= 100)
    printf("Passed in first class with distinction\n");
else if (marks >= 60 && marks <= 74)
    printf("Passed in first class \n");
else if (marks >= 50 && marks <= 59)
    printf("Passed in second class\n");
else if (marks >= 35 && marks <= 49)
    printf("Passed\n");
else printf("Failed\n");

```

To simplify this code further, note that initially the condition in the first **if** statement, i. e., **marks>= 75 && marks <= 100**, is evaluated. As we have assumed that the value of **marks** is valid (0 to 100), the condition **marks <= 100** is redundant.

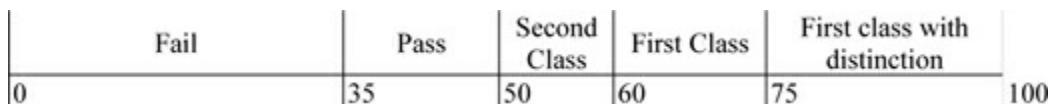
If the value of **marks** is less than 75, the control moves to the **else-block** to evaluate the condition **marks >= 60 && marks <= 74**. Since the value of **marks** is less than 75, as already verified in the first **if** statement, the second part of this condition (**marks <= 74**) is also redundant. By extending this logic, we can simplify the subsequent conditions as well. The simplified **if-else-if** statement is given below.

```

if (marks >= 75)
    printf("Passed in first class with distinction\n");
else if (marks >= 60)
    printf("Passed in first class \n");
else if (marks >= 50)
    printf("Passed in second class\n");
else if (marks >= 35)
    printf("Passed\n");
else printf("Failed\n");

```

Now correlate this statement with the desired sub-ranges depicted below and you will realize how easy it is to write such a statement.



We can also rewrite the above statement by considering the sub-ranges from left to right as

```

if (marks < 35)
    printf("Failed\n");
else if (marks < 50)
    printf("Passed\n");
else if (marks < 60)

```

```

    printf("Passed in second class\n");
else if (marks < 75)
    printf("Passed in first class\n");
else printf("Passed in first class with distinction\n");

```

Finally note that if the user enters invalid `marks` (e. g., -75, 137, etc.), the `if-else-if` statements given above will print incorrect result. This problem can also be easily overcome by adding a data validation statement in the beginning of the `if-else-if` chain as

```

if (marks < 0 || marks > 100)
    printf("Invalid marks\n");
else if (marks >= 75)
    printf("Passed in first class with distinction\n");
else if (marks >= 60)
    printf("Passed in first class \n");
else if (marks >= 50)
    printf("Passed in second class\n");
else if (marks >= 35)
    printf("Passed\n");
else printf("Failed\n");

```

### Program 7.1 Complete solution of a quadratic equation $ax^2 + bx + c = 0$

Write a program for the complete solution of a quadratic equation  $ax^2 + bx + c = 0$ .

**Solution:** The solution for the evaluation of the real roots of a quadratic equation was presented in Program 5.2. Now consider three distinct cases depending on the value of the discriminant  $d = b^2 - 4ac$  as shown below.

Discriminant	Nature of roots	Equations for roots
$b^2 - 4ac = 0$	Real and equal	$x_1 = x_2 = \frac{-b}{2a}$
$b^2 - 4ac > 0$	Real and unequal	$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ and $x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$
$b^2 - 4ac < 0$	Imaginary (Complex conjugate)	$x_1 = x_r + jx_i$ and $x_2 = x_r - jx_i$ where $x_r = \frac{-b}{2a}$ and $x_i = \frac{\sqrt{-d}}{2a}$ , where $d = b^2 - 4ac$

The last case,  $b^2 - 4ac < 0$ , requires explanation. We know that the standard library function `sqrt` requires a non-negative argument; otherwise, it gives a *domain error*. Thus, we can't directly evaluate the roots when  $b^2 - 4ac < 0$ . Hence, we need to evaluate the real part  $x_r$  and imaginary part  $x_i$  separately.

Let us use variables `a`, `b` and `c` of type `float` to represent the coefficients of the quadratic equation

and variable **d** of type **float** to represent the discriminant. The program given below first accepts the coefficients and calculates the discriminant. It then uses an **if-else-if** statement to determine and print the roots. Observe that the variables representing the roots are declared within the blocks in which they are used.

```
/* Complete solution of a quadratic equation ax^2 + bx + c = 0.  
Determines real as well as complex conjugate roots */  
#include <stdio.h>  
#include <math.h>  
  
int main()  
{  
    float a, b, c; /* coefficients of quadratic equation */  
    float d; /* discriminant */  
  
    /* read coefficients */  
    printf("Enter coefficients of quadratic equation: ");  
    scanf("%f %f %f", &a, &b, &c);  
  
    d = b * b - 4 * a * c;  
  
    if (d == 0) { /* real equal roots */  
        float x = -b / (2 * a);  
        printf("Roots are real and equal\n");  
        printf("    x = %4.2f\n", x);  
    }  
    else if (d >= 0) { /* real unequal roots */  
        float p = -b / (2 * a);  
        float q = sqrt(d) / (2 * a);  
        float x1 = p + q;  
        float x2 = p - q;  
        printf("Roots are real and unequal\n");  
        printf("    x1 = %4.2f x2 = %4.2f\n", x1, x2);  
    }  
    else { /* complex conjugate roots */  
        float xr = -b / (2 * a); /* real part */  
        float xi = sqrt(-d) / (2 * a); /* imaginary part */  
        printf("Roots are complex conjugate\n");  
        printf("    real part = %4.2f\n", xr);  
        printf("    imaginary part = %4.2f\n", xi);  
    }  
  
    return 0;
```

```
}
```

The program is executed twice and the output is given below.

```
Enter coefficients of quadratic equation: 1 3 2
Roots are real and unequal
x1 = -1.00      x2 = -2.00

Enter coefficients of quadratic equation: 1 2 3
Roots are complex conjugate
real part = -1.00
imaginary part = 1.41
```

### Program 7.2 Date validity check

Write a program to determine whether a given date is valid or not.

**Solution:** Let us use variables `dd`, `mm` and `yy` (all of type `int`) to represent the day, month and year in a given date. The given date is valid only if year (`yy`) is non-zero, month (`mm`) is in the range 1 to 12 and day of month (`dd`) is in the range 1 to `mdays`, the number of days in a given month `mm`. An algorithm to determine the validity of a given date is given below.

```
valid = 0;
if (yy != 0) {
    if (mm ≥ 1 && mm ≤ 12) {
        determine mdays, the number of days in month mm
        if (dd≥1 && dd≤mdays)
            valid = 1;
    }
}
```

The algorithm is very straight forward. Initially, a flag `valid` is set to zero (false) indicating that the given date is invalid. Then a *nested if* statement sets this flag to 1 (true) if all the required conditions are satisfied. Note that we have to determine `mdays`, the number of days in a given month, only if the value of `mm` is valid. Thus, we can use the simplified logic presented in Example 7.3c.

The program given below first accepts a date from the keyboard, determines whether it is valid or not using the algorithm given above and then prints an appropriate message.

```
/* Date validity program */
#include <stdio.h>

int main()
{
    int dd, mm, yy;      /* given date */
```

```

int valid;           /* flag to indicate date validity */

printf("Enter date as dd/mm/yyyy: ");
scanf("%d/%d/%d", &dd, &mm, &yy);

/* determine validity of given date */
valid = 0;
if (yy != 0) {      /* check year */
    if (mm >= 1 && mm <= 12) {        /* check month */
        /* determine number of days in given month */
        int mdays;
        if (mm == 2)
            mdays = (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0) ?
                29 : 28;
        else if (mm == 4 || mm == 6 || mm == 9 || mm == 11)
            mdays = 30;
        else mdays = 31;
        if (dd >= 1 && dd <= mdays)    /* check date dd */
            valid = 1;
    }
}
printf("Date is %s\n", valid ? "valid" : "invalid");
return 0;
}

```

Observe that the `%d` format specifiers in the `scanf` format string are separated by '/' characters as "`%d/%d/%d`". Such non-format characters in the format string of the `scanf` statement must be entered by the user at the indicated position. Thus, the user must enter the date in the format `dd/mm/yyyy`. The program is executed twice and the output is given below.

```

Enter date (dd mm yyyy): 29/2/2004
Date is valid

```

```

Enter date (dd mm yyyy): 29/2/2005
Date is invalid

```

An alternative algorithm to determine the validity of a given date is given below. It is based on the logic that a date is invalid if either year is invalid (zero), month is invalid (outside the range 1 to 12) or date is invalid (outside range 1 to `mdays`).

```

valid = 1
if(yy == 0 )
    valid = 0

```

```

else if (mm<1 || mm>12)
    valid = 0
else {
    determine mdays, the number of days in month mm
    if (dd<1 || dd>mdays)
        valid = 0
}

```

## Comparison of **switch** and **if-else-if** Statements

Both the **switch** and **if-else-if** statements enable us to select one of several alternative statements for execution. However, they differ in several aspects:

1. The **switch** statement restricts the selection of alternative statements for execution based on the value of a single control expression. The **if-else-if** statement, on the other hand, is very flexible and permits selection of alternative statements based on values of arbitrary expressions.
2. In the **switch** statement, each desired value of the control expression must be mentioned explicitly using a **case** keyword. However, in addition to the individual values, the **if-else-if** statement allows a range of values to be specified in test expressions.
3. The control expression in a **switch** statement must be an integral expression, whereas the **if-else-if** statement allows us to test integral and floating-point expressions as well as character strings (using the library functions provided in the `<string.h>`).
4. The syntax of the **switch** statement is much cleaner compared to the **if-else-if** statement. Thus, programs using **switch** statements are usually more readable.
5. The **switch** statement has the ability to execute more than one alternative statements by omitting **break** statements. This is not possible in **if-else-if** statement.

## 7.2 Using **if** Statements within Loops

### 7.2.1 Using **if** Statements within a **for** Loop

Consider the general format of the **for** statement given below:

```
for (initial_expr ; final_expr ; update_expr)
    statement ;
```

The *statement* included within a **for** loop may be a simple or compound statement. Thus, it can also be an **if-else** (or simple **if**) statement as shown below.

```
for (initial_expr ; final_expr ; update_expr) {
    if (expr)
        statement1 ;
    else statement2;
}
```

As **if-else** is single statement, the braces surrounding it are not necessary. However, it is a good practice to use them to improve program readability and to avoid errors when additional statements are introduced later within the body of the **for** loop. Note that the statements within the *if* and *else* blocks may be either simple or compound. Moreover, the included **if** statement may be a *nested if* statement. Finally note that the **for** loop may contain more than one **if** statement along with other statements.

#### Example 7.4 Using **if** statements within a **for** loop

##### a) Print a comma-separated list of numbers

Consider the code given below to print a comma-separated list of numbers from 1 to 10:

```
for (i = 1; i <= 10; i++) {
    printf("%d", i);
    if (i<10)
        printf(", ");
}
printf("\n");
```

The **if** statement within the body of the **for** loop is used to print a comma after each value of the loop variable except the last one. The output of a program containing this code is shown below.

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

We can write the above code concisely using the conditional expression as shown below:

```
for (i = 1; i <= 10; i++)
    printf("%d%s", i, i<10 ? ", " : "\n");
```

##### b) Print odd numbers in a given range m to n

```
for (num = m; num <= n; num++) {
    if (num % 2 == 1)
        printf("%d ", num);
}
```

In this example, a **for** loop is set up with values of loop variable **num** from **m** to **n**. The **if** statement is

executed for each value of `num` and if `num` is an odd number (`num % 2` equals 1), it is printed using the `printf` statement. The output of this code is given below for `m = 20` and `n = 40`.

Odd numbers in the range 20 to 40:  
21 23 25 27 29 31 33 35 37 39

Note that as the `for` loop contains a single `if` statement, we can omit the braces (not a good practice, though). Also, the condition in the `if` statement can be simplified as `num % 2`.

An efficient approach to print odd numbers in the range `m` to `n` is given below.

```
if (m % 2 == 0)
    m++;
for (num = m; num <= n; num += 2)
    printf("%d ", num);
```

The `if` statement adjusts the value of `m` to the first odd number to be printed. Then a `for` loop is set up such that the loop variable `num` assumes only odd values in the range `m` to `n` (observe the increment expression `num += 2`). Each value of `num` is printed in the body of the `for` loop.

Consider an even more concise solution presented in Section 7.4.1 in which a conditional expression is used to initialize the value of variable `num` to the first odd number to be printed.

```
for (num = (m % 2 == 1 ? m : m + 1); num <= n; num += 2)
    printf("%d ", num);
```

c) Determine the maximum and minimum from a given list of  $n$  numbers

The program segment given below determines the maximum of  $n$  given numbers.

```
printf("Enter %d numbers: ", n);
scanf("%f", &num);      /* read first number */
max = num;              /* assume first number to be maximum */
for (i = 1; i < n; i++) {
    scanf("%f", &num);      /* read next number */

    if (num > max)        /* if new maximum found, */
        max = num;          /* update maximum */
}
printf("Maximum: %f\n", max);
```

Initially, the first number in the list is read and it is assigned to variable `max`. Then a `for` loop is set up to process the remaining numbers in the list and determine the maximum. In each iteration of `for` loop, the next number is accepted from the keyboard and is compared with the current value of `max`. If

the entered number (`num`) is greater than `max`, it is assigned to `max` as the new maximum. After the execution of `for` loop is over, variable `max` contains the maximum of the given numbers. Note that to determine the minimum from a given list of  $n$  numbers, we should replace `max` with `min` and modify the condition in the `if` statement as `num < min`.

Now consider a similar program segment that determines both the minimum and the maximum from  $n$  given numbers. Observe that it uses an `if-else` statement to update the values of variables `min` and `max` within the loop.

```
printf("Enter %d numbers:\n", n);
scanf("%f", &num);           /* read first number */

min = max = num;           /* assume it to be minimum and maximum */
for (i = 1; i < n; i++)
    scanf("%f", &num);     /* read next number */
    if (num < min)
        min = num;         /* update minimum */
    else if (num > max)
        max = num;         /* update maximum */
}
printf("Maximum: %f Minimum: %f\n", max, min);
```

#### d) Print three-digit Armstrong numbers

An Armstrong number has an interesting property that the sum of cubes of its digits is equal to the number itself (e. g.,  $153 = 1^3 + 5^3 + 3^3$ ). The program segment given below prints all the three-digit Armstrong numbers.

```
printf("Three digit Armstrong numbers: ");
for (num = 100; num <= 999; num++) {
    /* separate digits */
    int u = num % 10;
    int t = num / 10 % 10;
    int h = num / 100;

    if (num == u * u * u + t * t * t + h * h * h)
        printf("%d ", num);
}
```

It uses a `for` loop to test each three digit number (100 to 999). For each number, the digits at the unit's, ten's and hundred's place are separated in variables `u`, `t` and `h`, respectively. Then an `if` statement tests if number `num` is an Armstrong number, i. e., if it is equal to the sum of the cubes of its digits. The output of a program containing this code is shown below.

Three digit Armstrong numbers: 153 370 371 407

e) Print four digit special perfect square numbers

The program segment given below prints four-digit special perfect square numbers in which the upper and lower two-digit numbers are perfect squares as well.

```
printf("Four digit special perfect square numbers: ");
for (num = 1000; num <= 9999; num++)
    int upper = num / 100; /* upper 2 digit no. */
    int lower = num % 100; /* lower 2 digit no. */

    int nroot = (int) sqrt(num);
    int uroot = (int) sqrt(upper);
    int lroot = (int) sqrt(lower);

    if (num == nroot * nroot && upper == uroot * uroot &&
        lower == lroot * lroot)
        printf("%d ", num);
}
```

A **for** loop is set up such that the loop variable **num** assumes four-digit integer values (1000 to 9999). Within this loop, the **upper** and **lower** two-digit numbers are first separated in variables **upper** and **lower**. Then the integer parts of the square roots of **num**, **upper** and **lower** are determined in variables **nroot**, **uroot** and **lroot** (all of type **int**), respectively. Subsequently, an **if** statement prints number **num** if it is a special perfect square number, i. e., if the numbers **num**, **upper** and **lower** are perfect squares. Observe the use of the logical AND (**&&**) operator in the condition in the **if** statement. The output of a program containing this code is given below.

```
Four digit special perfect square numbers:
1600 1681 2500 3600 4900 6400 8100
```

Observe that the code given above is inefficient as it determines the upper and lower two-digit numbers and their square roots for each number **num**. Thus, the **sqrt** function, which is compute-intensive, is evaluated 27,000 times (9000 numbers  $\times$  3).

A program segment using an alternative, more efficient strategy is given below.

```
for (num = 1000; num <= 9999; num++) {
    int nroot = (int) sqrt(num);

    if (num == nroot * nroot) { /* num is perfect square */
        int upper = num / 100; /* upper 2 digit no. */
        int uroot = sqrt(upper);
```

```

        if (upper == uroot * uroot) { /* upper is perfect square */
            int lower = num % 100; /* lower 2 digit no. */
            int lroot = sqrt(lower);

                if (lower == lroot * lroot) /* lower is perfect square */
                    printf("%d ", num);
            }
        }
    }
}

```

In this code, we determine the upper two-digit number (`upper`) and its square root only if the number `num` is a perfect square. Also, the lower-two digit number (`lower`) and its square root is determined only if the upper two-digit number (and of course the number `num`) are perfect squares. It can be verified that the `sqrt` function is now evaluated only 9075 times.

### 7.2.2 Using `if` Statements within `while` and `do ... while` Loops

We have seen how an `if` statement can be used within the body of a `for` loop. The `if` statement can also be included in the body of the `while` and `do ... while` loops as shown below.

```

while (expr1) {
    if (expr2)
        statement1 ;
    else statement2 ;
}

do {
    if (expr2)
        statement1 ;
    else statement2 ;
} while (expr1) ;

```

Although the braces surrounding the `if` statements are not essential, including them here is a good programming practice. Of course, if these loops contain additional statements (before and/or after the `if` statement) these braces are required. Note that the `else` clause is optional and the statements within the `if` and `else` blocks may be simple or compound. Moreover, the `if` statements may be *nested if* statements.

#### Example 7.5 Using `if` statements within the `while` and `do ... while` loops

##### a) Count number of odd and even digits in a given integer number

This program segment given below uses a straight-forward approach to count the number of odd and even digits in a given integer number (`num`).

```

nodd = neven = 0; /* count of odd and even digits */

while (num>0) {

```

```

        digit = num % 10; /* separate LS digit from number */
        if (digit % 2 == 1)
            nodd++;
        else neven++;
        num /= 10;           /* remove LS digit from num */
    }

printf("Odd digits : %d Even digits: %d\n", nodd, neven);

```

Initially, variables `nodd` and `neven`, the counts for odd and even digits, respectively, are initialized to zero. Then a `while` loop is used to count the odd and even digits. In each iteration of this loop, the least significant digit of number `num` is first determined in variable `digit`, then the appropriate counter is incremented using an `if-else` statement and the least significant `digit` is removed from number `num`. The loop is executed as long as the value of `num` is greater than zero, i. e., all digits in `num` are not processed. Finally, the values of `nodd` and `neven` are printed. Observe the similarity of this program segment with the code in Program 6.5 used to determine the sum of digits of an integer number.

The code has been written below concisely by using the conditional expression operator (`? :`) and eliminating variable `digit`.

```

nodd = neven = 0;           /* count of odd and even digits */

while (num>0) {
    (num % 2 == 1) ? nodd++ : neven++;
    num /= 10; /* remove LS digit from num */
}

printf("Odd digits : %d Even digits: %d\n", nodd, neven);

```

### b) Data entry validation

The program segment given below reads a number from the keyboard (in variable `num` of type `float`) and prints its square root. However, as the `sqrt` function requires a non-negative argument, the program uses a `do ... while` loop to read the data until valid data is entered.

```

do {
    printf("Enter a non-negative number: ");
    scanf("%f", &num);
    if (num < 0)
        printf("Error: Invalid data.\n");
} while (num < 0);

```

```
printf("Number: %f Square root: %f\n", num, sqrt(num));
```

The program containing this code is executed and the sample output is shown below.

```
Enter a non-negative number: -9
Error: Invalid data.
Enter a non-negative number: 9
Number: 9.000000 Square root: 3.000000
```

### Program 7.3 Test whether a number is a prime or not

Write a program to determine whether a given integer number is prime or not.

**Solution:** An integer number is *prime* if it is not evenly divisible by any number other than 1 and the number itself. Thus, to test whether a given number `num` is prime or not, we need to divide it by integers from 2 to `num - 1` and test whether the remainder is zero or not.

Variable `is_prime` is used as a flag to indicate whether the given number is prime or not. The `if` statement given below sets `is_prime` to 0 (i.e., *false*) if `num` is divisible by some integer number `d`, i.e., if `num % d` is zero.

```
if (num % d == 0)
    is_prime = 0;           /* num is not a prime */
```

To perform the above test for each divisor `d` from 2 to `num-1`, an `if` statement is used within a `while` loop as shown below.

```
is_prime = (num>1 ? 1 : 0); /* if num>1, assume it to be prime */
d = 2;
while (d<num) {
    if (num % d == 0)
        is_prime = 0; /* num is not a prime */
    d++;
}
```

Initially, if the given number is greater than 1, we assume it to be a prime number and initialize `is_prime` to 1; otherwise, `is_prime` is initialized to zero. The divisor `d`, which is used as the loop variable, assumes the values from 2 to `num-1`. For each value of `d`, the value of `is_prime` is reset to zero if `num` is divisible by `d`. Thus, after the execution of the `while` loop, `is_prime` will indicate whether `num` is prime or not. We can then use an `if` statement to print a message accordingly.

Note that if the remainder of any of division (`num % d`) in the `while` loop is zero, the given number is not prime and subsequent iterations of the loop are unnecessary. Thus, the value of variable `is_prime` can be used in the `while` loop control expression to avoid subsequent iterations as

```
while (is_prime && d < num)
{
    ...
}
```

The complete program using this approach is given below.

```
/* Test if a given number is prime or not */
#include <stdio.h>
int main()
{
    int num, d; /* given number and divisor */
    int is_prime; /* flag indicating if number is prime or not */
    printf("Enter an integer number: ");
    scanf("%d", &num);
    /* determine if num is prime or not */
    is_prime = (num > 1 ? 1 : 0); /* if num > 1, assume it to be prime */
    d = 2;
    while (is_prime && d < num) {
        if (num % d == 0)
            is_prime = 0; /* num is not a prime */
        d++;
    }
    if (is_prime)
        printf("%d is a prime number\n", num);
    else printf("%d is not a prime number\n", num);
    return 0;
}
```

The program is executed twice and the output is given below.

```
Enter an integer number: 24
24 is not a prime number
```

```
Enter an integer number: 13
13 is a prime number
```

Note that the code to determine whether a number is prime or not can be written concisely using a **for** loop as shown below.

```
is_prime = (num>1 ? 1 : 0);

for(d = 2; is_prime && d<num; d++)
    if (num % d == 0)
```

```
    is_prime = 0;      /* num is not a prime */
}
```

## 7.3 Nested Loops

The C language provides three loops (**for**, **while** and **do ... while**). As contained *statement* in the body of the loop can be any valid C statement, we can obtain several nested-loop structures by replacing this *statement* with another loop statement. Thus, if we replace the *statement* in a **for** loop with another **for** loop, we will get a two-level nested **for** loop as

```
for ( initial_expr1 ; final_expr1 ; update_expr1 ) {
    for ( initial_expr2 ; final_expr2 ; update_expr2 ) {
        statement ;
    }
}
```

Observe that the inner **for** loop is executed once for each iteration of the outer **for** loop. Thus, if the outer and inner loops are set up to perform  $m$  and  $n$  iterations, respectively, the *statement* contained within the inner **for** loop will be executed  $m \times n$  times. Also note the use of curly braces to improve code readability. The two-level nested **for** loops are commonly used for operations with two-dimensional arrays, i. e., matrices. This is discussed in the next chapter.

The general forms for two-level nested **while** and **do ... while** loops are given below.

```
while ( expr1 ) {
    while ( expr2 ) {
        statement ;
    }
}
do {
    do {
        statement ;
    } while ( expr2 );
} while ( expr1 );
```

If we replace the contained *statement* within a **for** loop with a **while** or **do ... while** statement, we can get other forms of two-level nested loops as shown below.

```
for (initial_expr ; final_expr ; update_expr ) {
    while (expr) {
        statement ;
    }
}
```

```
for (initial_expr ; final_expr ; update_expr )
    do {
        statement ;
    } while (expr);
```

}

Similarly, by replacing the *statement* contained within the `while` and `do ... while` loops with other types of loop statements, we can obtain other general forms of nested loops.

Note that as the *statement* contained within two-level nested loops can also be any valid C statement including a control statement, we can obtain higher levels of nested loops. Thus, we can replace the *statement* contained in a two-level nested **for** loop given above with another **for** loop to obtain a three-level nested **for** loop as shown below.

```
for ( initial_expr1 ; final_expr1 ; update_expr1 ) {  
    for ( initial_expr2 ; final_expr2 ; update_expr2 ) {  
        for ( initial_expr3 ; final_expr3 ; update_expr3 ) {  
            statement ;  
        }  
    }  
}
```

Such nested **for** loops are used for operations with three-dimensional arrays and also to perform matrix multiplication operations.

### Example 7.6 Nested loops

a) Print a multiplication table

The program segment given below prints a multiplication table from 21 to 30.

```
for (r = 1; r <= 10; r++)
    for (c = 21; c <= 30; c++)
        printf("%3d ", r * c);
    printf("\n");
}
```

The outer **for** loop is set up to print 10 rows in the table. The loop variable (**r**) takes values from 1 to 10 and acts as a multiplier. The loop body consists of two statements - a **for** loop that prints the elements in the *r*th row of the multiplication table and a **printf** statement that prints a *newline* after a row is printed. The loop variable in the inner **for** loop (**c**) takes values from 21 to 30. For each value of **c**, the **printf** statement prints the value of **r\*c** using the format "%3d". The multiplication table is printed as shown below (some rows are omitted to save space).

**b) Determine the sum of digits of a number repeatedly to obtain a single-digit number**

The program segment given below determines the sum of digits of a given number repeatedly until a single digit number is obtained. For example,  $5985 \Rightarrow 27 \Rightarrow 9$ , where symbol  $\Rightarrow$  indicates a digit sum operation. Thus, if digit sum exceeds 9, it is used as a number for subsequent digit sum operations.

```
do {
    sum = 0                                /* sum of digits of number */
    do {
        sum += num % 10; num /= 10; /* add LS digit to sum */
        num /= 10;                  /* remove LS digit from num */
    } while (num>0);

    num = sum;                                /* use digit sum as new number */
} while (num>9);                          /* as long as num has 2 or more digits */
printf("Sum of digits : %d\n", sum);
```

This code uses a two-level nested `do ... while` loop. The inner `do ... while` loop is used to determine the sum of digits of number `num` (see Example 6.4a). After the sum of digits is calculated in the inner `do ... while` loop, it is assigned to `num` as a new number. The outer `do ... while` loop determines the sum of digits repeatedly as long as `num` is greater than 9.

We can replace the inner `do ... while` loop with a `while` or `for` loop. The code obtained by replacing it with a `while` loop is given below.

```
do {
    sum = 0;
    while (num>0) {
        sum += num % 10;
        num /= 10;
    }
    num = sum;
} while (num>9);

printf("Sum of digits : %d\n", sum);
```

**c) Prime numbers in a given range**

In Program 7.3, we have seen how to test whether a given number is prime or not. The program segment given below uses a `for` loop to test each number in a given range and print it if it is prime.

```
for (num = m; num <= n; num++) {
```

```

/* determine if num is a prime or not */
int is_prime = (num > 1 ? 1 : 0); /* if num > 1, assume prime */
int d = 2; /* divisor */
while (is_prime && d < num) {
    if (num % d == 0)
        is_prime = 0; /* num is not a prime */
    d++;
}
if (is_prime)
    printf("%d ", num);
}

```

This program segment has a **while** loop nested within a **for** loop. We can replace the **while** loop with a **for** loop to obtain nested a for loop implementation as shown below.

```

for (num = m; num <= n; num++) {
    int d;
    int is_prime = (num > 1 ? 1 : 0);
    for(d = 2; is_prime && d < num; d++) {
        if (num % d == 0)
            is_prime = 0; /* num is not a prime */
    }
    if (is_prime)
        printf("%d ", num);
}

```

#### Program 7.4 Prime factors of a given number

Write a program to print all the prime factors of a given number.

**Solution:** As the name indicates, the prime factors of a given number are its factors that are also prime numbers, e. g., prime factors of 30 are 2, 3 and 5 but not 1, 6, 10, 15 and 30. One or more prime factors of a given number may repeat, e. g., prime factors of 120 are 2, 2, 2, 3 and 5.

To print all occurrences of a prime factor (say **divisor**) in given number **num**, use the **while** loop given below.

```

while (num % divisor == 0) {
    printf("%d ", divisor); /* divisor is a factor, print it */
    num /= divisor;          /* remove the factor from num */
}

```

The statements within this loop are executed repeatedly as long as **num % divisor** is zero, i. e., **divisor** is a factor of **num**. Within the loop, first print the **divisor** as a factor and then remove it from **num**. Note that if **divisor** is not a factor of **num**, nothing happens in the loop.

To print all prime factors of a given number `num`, we can embed this code in another loop that tests the values of `divisor` from 2 to `n-1`. Within this loop, we can first test whether `divisor` is a prime number or not and if it is, separate all its occurrences from `num` using the `while` loop given above. Note that when all the prime factors in a given number are printed and removed, the value of `num` will be 1. Thus, the outer loop should continue as long as the value of `num` is greater than 1. A straightforward algorithm that uses this approach is given below.

```
divisor = 2
while (num>1) {
    test if divisor is a prime and set is_prime flag /* this step is redundant */
    if (is_prime)                                /* this test is redundant */
        print and remove all occurrences of divisor in num (using above while loop)
    divisor++
}
```

However, the prime test in the above algorithm is unnecessary and can be omitted. Note that as we print and remove all factors starting from 2, the subsequent non-prime factors of original number `num` will no longer be factors of the modified value of `num`. Consider, for example, number 40. When we remove all occurrences of prime factor 2, the number is modified to 5. Thus, numbers 4, 8, 10, 20 which were the factors of the original number are not factors of modified number (5). The complete program using the nested `while` loop is given below, followed by the output.

```
/* Determine prime factors of a given number */
#include <stdio.h>

int main()
{
    int num, divisor; /* number and divisor */

    printf("Enter a number: ");
    scanf("%d", &num);

    /* determine and print factors */
    printf("Prime factors: ");
    divisor = 2;

    while (num > 1) {
        while (num % divisor == 0) {
            printf("%d ", divisor); /* divisor is a factor, print it */
            num /= divisor; /* remove the factor from num */
        }
        divisor++;
    }
}
```

```
    }
    return 0;
}
```

```
Enter a number: 120
Prime factors: 2 2 2 3 5
```

We can also use a **while** loop nested within a **for** loop as shown below.

```
printf("Prime factors: ");
for(divisor = 2; num>1; divisor++) {
    while (num % divisor == 0)  {
        printf("%d ", divisor); /* divisor is a factor, print it */
        num /= divisor;          /* remove the factor from num */
    }
}
```

### Program 7.5 Print Pythagorean triplets

Write a program to print all Pythagorean triplets  $a$ ,  $b$ ,  $c$  such that the values of  $a$  and  $b$  are less than a user-specified value  $max$ .

**Solution:** Three positive integer numbers  $a$ ,  $b$  and  $c$ , such that  $a < b < c$  form a Pythagorean triplet if  $c^2 = a^2 + b^2$ , i.e.,  $a$ ,  $b$  and  $c$  form the sides of a right-angled triangle. To select the values of  $a$  and  $b$  such that  $a \leq b$  and  $a, b \leq max$ , we can use nested **for** loops as shown below:

```
for (a = 1; a <= max; a++) {
    for (b = a; b <= max; b++) {
        ...
    }
}
```

We can determine the value of  $c$  within the inner **for** loop as  $c = \sqrt{a^2 + b^2}$  and print the triplet if  $c$  is an integer value. This can be done as follows:

```
float c = sqrt(a * a + b * b); /* third side */
if (c == (int) c)
    printf("%2d %2d %2d\n", a, b, (int) c);
```

Note the use of typecast in the **if** statement to test whether  $c$  contains an integer value or not and in the **printf** statement. The complete program is given below.

```
/* Determine and print Pythagorean triplets */
#include <stdio.h>
```

```

#include <math.h>

int main()
{
    int max; /* max value of side a or b */
    int a, b; /* two sides of a triangle */

    printf("Enter max value of sides a, b: ");
    scanf("%d", &max);

    for (a = 1; a <= max; a++) {
        for (b = a; b <= max; b++) {
            float c = sqrt(a * a + b * b); /* third side */
            if (c == (int) c)
                printf("%2d %2d %2d\n", a, b, (int) c);
        }
    }
    return 0;
}

```

The program output is given below:

```

Enter max value of sides a, b: 15
3 4 5
5 12 13
6 8 10
8 15 17
9 12 15

```

Note that the code to print the Pythagorean triplets can also be written as follows:

```

for (a = 1; a <= max; a++) {
    for (b = a; b <= max; b++) {
        int c_sqr = a * a + b * b;
        int c = (int) sqrt(c_sqr);
        if (c_sqr == c * c)
            printf("%2d %2d %2d\n", a, b, c);
    }
}

```

## 7.4 Nested Control Structures involving **switch** Statement

The **switch** statement can contain any valid C statement. Thus, we can use various control statements, namely, **if-else**, **for**, **while** and **do ... while** within a **switch** statement. We can also use a **switch** statement within another **switch** statement. Moreover, a **switch** statement can also be included in other control statements.

### Example 7.7 Nested control structures involving **switch** statements

#### a) Print a given integer number in words

Consider that we wish to print a given positive integer number in words, as a sequence of digit strings. For example, number 123 should be printed as *One Two Three*. This might be required in financial applications, for example, to print the cheque amount in words.

We can use a loop to separate the digits of a given number and print each of them as a word using a **switch** statement. However, as the digits are separated from the right side (i. e., least significant digit) of a number, we need to reverse the digits of given number before this loop. The program segment that uses this approach is given below.

```
/* reverse given number num */
rev = 0;
while(num>0)    {
    rev = rev * 10 + num % 10;
    num /= 10;
}
/* print number as digit strings */
while(rev>0)    {

    switch(rev % 10)  {
        case 0 : printf ("Zero "); break;
        case 1 : printf ("One "); break;
        case 2 : printf ("Two "); break;
        case 3 : printf ("Three "); break;
        case 4 : printf ("Four "); break;
        case 5 : printf ("Five "); break;
        case 6 : printf ("Six "); break;
        case 7 : printf ("Seven "); break
        case 8 : printf ("Eight "); break
        case 9 : printf ("Nine "); break;
    }
    rev /= 10;
}
```

**b)** Determine the number of days in a given month

The code given below uses a `switch` statement to determine the number of days in a given valid month `mm` (1 to 12). The case for February includes an `if-else` statement that checks whether the given year `yy` is a leap or not and accordingly assigns either 29 or 28 to variable `mdays`. Observe that the assumption of month `mm` being valid greatly simplifies the code by eliminating the cases for the months having 31 days.

```
/* determine number of days - assume mm is valid */
switch (mm) {
    case 2: /* check if year yy is leap */
        if (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0)
            mdays = 29;
        else mdays = 28;
        break;

    case 4:
    case 6:
    case 9:
    case 11:
        mdays = 30;
        break;

    default:
        mdays = 31;
        break;
}
printf("Days: %d\n", mdays);
```

However, if variable `mm` may contain an invalid value, we can use the above `switch` statement within an `if-else` statement as shown below:

```
if (mm >= 1 && mm <= 12) {
    switch (mm) {
        ... /* code omitted to save space */
    }
} else {
    printf("Invalid month.\n");
    mdays = 0;
}

printf("Days: %d\n", mdays);
```

## Program 7.6 Simple calculator – Modified

Modify the simple calculator program given in Program 5.3 to avoid the division by zero error and to handle invalid operators.

**Solution:** The modified program to avoid the division by zero error and to handle invalid operator is given below. It uses an **if-else** statement within the **case** for the division operator which evaluates expression  $a/b$  only if the value of  $b$  is not equal to zero; otherwise, it prints the error message *Division by zero*. The **default** case is introduced to handle an invalid operator entered from the keyboard and print the error message *Invalid operator*. Also observe the use of the variable **error** that keeps track of error status and prints the result after the **switch** statement only if there is no error.

```
/* Simple calculator - Modified to avoid division by zero error */
#include <stdio.h>

int main()
{
    float a, b, c; /* operands */
    char op;        /* operator: + - * or / */
    int error = 0;

    printf("Enter expression of the form a op b:\n");
    scanf("%f %1c %f", &a, &op, &b);

    switch(op) {
        case '+':
            c = a + b;
            break;

        case '-':
            c = a - b;
            break;

        case '*':
        case 'x':
        case 'X':
            c = a * b;
            break;
        case '/':
            if (b != 0)
                c = a / b;
            else {
                printf("Division by zero\n");
                error = 1;
            }
            break;
        default:
            printf("Invalid operator\n");
            error = 1;
    }

    if (error == 0)
        printf("Result = %f\n", c);
}
```

```
        error = 1;
    }
    break;

default:
    printf("Invalid operator\n");
    error = 1;
    break;
}

if (!error)
    printf("c = %f\n", c);
return 0;
}
```

## 7.5 Loop Interruption

We know that each loop has a termination condition. The statements within the loop are executed as long as the specified condition evaluates as true. However, sometimes we need to interrupt the execution of a loop when a specific condition is satisfied. C language provides the **break** statement for this purpose. The C language also provides the **continue** statement that enables us to terminate the execution of the current iteration and continue with the next iteration.

### 7.5.1 The **break** Statement

We have seen that a **break statement** is usually used in a **switch** statement after the statements in each **case**. The execution of such a **break** statement causes the execution of the **switch** statement to be terminated and the control to be transferred to the statement following the **switch** statement.

C also allows the **break** statement to interrupt the execution of a loop and transfer the control to the statement following that loop. The format of this statement is

**break;**

It is obvious that within a loop, the **break** statement must be used in an **if** statement to transfer the control outside the loop only when a specified condition is satisfied; otherwise, the looping action will not take place.

The **break** statement terminates the execution of the nearest enclosing **for**, **while**, **do** or **switch** statement in which it appears. Thus, if we use the **break** statement within nested loops, its execution causes the control to leave the innermost loop in which it is contained. Note that the compiler will report an error if the **break** statement is encountered outside a **switch** or looping construct.

### Example 7.8 Using the **break** statement within loops

#### a) Add numbers until the desired sum is obtained or all the given numbers are exhausted

Consider the program segment given below that determines the sum of  $n$  given numbers until a desired value **max\_sum** is attained or all the numbers are exhausted:

```
sum = 0;
for (i = 0; i < n; i++) {
    scanf("%d", &num);
    sum += num;
    if (sum >= max_sum)
        break;
}
```

The **for** loop is set up to determine the sum of  $n$  numbers entered from the keyboard. However, if a specified **max\_sum** is attained, the execution of the loop is interrupted using a **break** statement.

Now let us print the result that includes the final value of **sum** and count of numbers required to attain this sum. This can be done inside the **if** statement in the above code. However, if the desired sum is not attained even after processing all the numbers, the result will not be printed. Thus, it is appropriate to print this result after the **for** loop using a **printf** statement as shown below.

```
printf("Sum = %d count = %d\n", sum, ??); /* replace ?? by
                                               count expr */
```

However, the problem here is in printing the count of numbers actually processed as indicated by **??** in the above statement. Observe that if **max\_sum** is attained, the control will surely leave the loop via the **break** statement and the desired count is equal to **i+1**; otherwise, the control leaves the loop after  $n$  iterations are over, in which case, the desired count will be equal to either  $n$  or  $i$ . Thus, we can write the above **printf** statement using a conditional operator (**?:**) as shown below:

```
printf("Sum = %d count = %d\n", sum, (sum >= max_num ? i + 1 : n));
```

Now assume that we wish to improve the result to indicate whether the desired **max\_sum** was attained or not. This can be done using an **if-else** statement after the loop:

```
if (sum<max_sum)
    printf("Max sum not attained. Sum = %d count = %d\n", sum, n);
else printf("Max sum attained. Sum = %d count = %d\n", sum, i + 1);
```

#### b) Test whether a given number is prime or not

In Program 7.3, we have seen how to test whether a given number `num` is prime or not. The code segment to determine the value of the `is_prime` flag has been reproduced below for convenience.

```
is_prime = (num > 1 ? 1 : 0);
d = 2;
while (is_prime && d < num) {
    if (num % d == 0)
        is_prime = 0; /* num is not a prime */
    d++;
}
```

The value of `is_prime` is then used to print whether the given number is prime or not. Note that the control leaves the `while` loop when `is_prime` becomes zero. Thus, we can rewrite the above code using the `break` statement as shown below.

```
is_prime = (num>1 ? 1 : 0);
d = 2;
while (d<num) {
    if (num % d == 0) {
        is_prime = 0; /* num is not a prime */
        break;
    }
    d++;
}
```

Observe that the test expression in the `while` loop is also simplified as the test of `is_prime` is not required. The equivalent code using a `for` loop is given below.

```
is_prime = (num>1 ? 1 : 0);
for(d = 2; d<num; d++) {
    if (num % d == 0) {
        is_prime = 0; /* num is not a prime */
        break;
    }
}
```

Also observe that if the given number `num` is prime, all the iterations of the loop will be completed and the value of the loop variable `d` will be equal to `num`. However, if the given number is not prime, the control will leave the loop, via `break` statement and the value of `d` will be less than `num`. We can thus use the value of `d` to test whether the given number is prime or not and the flag `is_prime` is unnecessary. The modified program segment is given below.

```
/* simplified code to det. whether a number num is prime or not */
```

```

for(d = 2; d<num; d++) {
    if (num % d == 0)
        break;
}
if (d == num)
    printf("Prime number\n");
else printf("Not a prime number\n");

```

### c) Prime numbers in a given range

The program segment given below prints prime numbers in a given range  $m$  to  $n$ .

```

for (num = m; num <= n; num++) {
    for(d = 2; d<num; d++) {
        if (num % d == 0)
            break;
    }
    if (d == num)
        printf("%d ", num);
}

```

The outer **for** loop is set up to process each number in the given range. Each number is tested within this loop using the simplified code given in the previous example. The execution of the **break** statement causes inner **for** loop to terminate as it is the nearest loop enclosing the **break** statement. If the inner **for** loop is completely executed, i. e., if condition  $d == num$  is true,  $num$  is printed as a prime number.

## 7.5.2 The **continue** Statement

The **continue Statement** is another loop interruption statement provided in the C language. It interrupts only the current iteration of the loop as opposed to the **break** statement which interrupts the execution of the entire loop. The format of the **continue** statement is as follows:

**continue;**

When this statement is executed, control is passed to the next iteration of the loop in which it appears, bypassing any remaining statements in that loop. In case of **while** and **do ... while**, the control is passed to the terminating condition, whereas in the **for** loop, the update expression is evaluated before the control is passed to the final expression.

### Example 7.9 Using the **continue** statement within loops

#### a) Addition of positive numbers in a given list

Consider the program segment given below that accepts  $n$  integer numbers from the keyboard and determines the sum of only positive numbers in this list.

```
sum = 0;
for (i = 0; i<n; i++) {
    scanf("%d", &num);
    if (num <= 0)
        continue;
    sum += num;
}
```

The `continue` statement causes control to be transferred to update expression (`i++`) and then to the final expression, i. e., `i < n`, skipping the execution of the statement that updates the value of `sum`. Note that this code can also be rewritten without the `continue` statement as shown below.

```
sum = 0;
for (i = 0; i<n; i++) {
    scanf("%d", &num);
    if (num>0)
        sum += num;
}
```

### b) Print four-digit special perfect square numbers

In Example 7.4e, we learnt how to print four-digit special perfect square numbers, i. e., the number itself as well as its upper two-digit and lower two-digit numbers are perfect squares, e. g., 1600, 1681, etc. For efficient implementation, we used the *nested if* statements to avoid the unnecessary evaluations of the `sqrt` function.

An alternative simplified implementation that uses the `continue` statement to eliminate *nested if* statements is given below.

```
for (num = 1000; num <= 9999; num++) {
    /* test if num is a perfect square */
    nroot = (int) sqrt(num);
    if (num != nroot * nroot)
        continue;

    /* test if upper two digits form a perfect square */
    upper = num / 100; /* upper 2 digit no. */
    uroot = (int) sqrt(upper);
    if (upper != uroot * uroot)
        continue;
```

```

/* test if lower two digits form a perfect square */
lower = num % 100; /* lower 2 digit no. */
lroot = (int) sqrt(lower);
if (lower != lroot * lroot)
    continue;
printf("%d ", num);
}

```

## 7.6 Advanced Concepts

### 7.6.1 Logical Operators and Nested if Statements

In this section, we will study the equivalent statements for **if** statements containing *logical AND* (**&&**) and *logical OR* (**||**) operators.

#### **if** Statements Containing Logical AND (**&&**) Operators

An **if** statement containing *logical AND* (**&&**) operators in the test expression can be equivalently written using *nested if* statements. Consider the following simple *if* statement:

```

if ( expr1 && expr2 )
    statement ;

```

The contained *statement* is executed when both *expr1* and *expr2* evaluate as true. Also, if *expr1* evaluates as false, *expr2* is not evaluated. Thus, this **if** statement is equivalent to the following *nested if* statement:

```

if ( expr1 )
    if ( expr2 )
        statement ;

```

Now consider the following **if-else** statement that uses the logical AND (**&&**) operator and its equivalent using a *nested if* statement:

<pre> <b>if</b> ( expr1 &amp;&amp; expr2 )     statement1 ; <b>else</b> statement2 ; </pre>	<pre> <b>if</b> ( expr1 )     <b>if</b> ( expr2 )         statement1 ;     <b>else</b> statement2 ; <b>else</b> statement2 ; </pre>
---	---

In these statements, *statement1* is executed provided both *expr1* and *expr2* evaluate as true; otherwise *statement2* is executed. Also, if *expr1* evaluates as false, *expr2* is not evaluated. Observe that in the *nested if* statement, *statement2* is included in the *else* blocks of the outer and inner **if** statements.

## **if** Statements Containing Logical OR (||) Operators

**if** statements containing *logical or* (||) operators in the test expression can be equivalently written using **if-else-if** statements. Consider the following simple **if** statement and its equivalent statement:

```
if (expr1 || expr2)
    statement ;
if (expr1)
    statement ;
else if (expr2)
    statement ;
```

The contained *statement* is executed when either *expr1* evaluate as true or when *expr1* evaluates as false and *expr2* evaluates as true. Observe that *expr2* is not evaluated if *expr1* evaluates as true (short-circuit evaluation). Now consider the following **if-else** statement and its equivalent using the **if-else-if** statement:

```
if (expr1 || expr2)
    statement1 ;
else statement2 ;
if (expr1)
    statement1 ;
else if (expr2)
    statement1 ;
else statement2 ;
```

Here *statement1* is executed when either *expr1* evaluate as true or when *expr1* evaluates as false and *expr2* evaluates as true; otherwise, *statement2* is executed. In addition, if *expr1* evaluates as true, *expr2* is not evaluated. Observe that in the equivalent statement, *statement1* is included in the *if* blocks of both *if* statements.

### **Example 7.10** Date validity check (using logic operators)

An algorithm to determine whether a given date is valid or not is given in Program 7.2. It uses *nested if* statements. We can greatly simplify this algorithm using logical AND (&&) operators instead of *nested if* statement as shown below.

```
determine mdays, the maximum number of days in month mm
if (yy != 0 && (mm>1 && mm < 12) && (dd>1 && dd<mdays))
    valid = 1;
else valid = 0;
```

The program segment given below determines date validity.

```
/* determine number of days in given month */
if (mm >= 1 && mm <= 12)
{
    if (mm == 2)
        mdays = (yy%4 == 0 && yy%100 != 0 || yy%400 == 0)? 29 : 28;
    else if (mm == 4 || mm == 6 || mm == 9 || mm == 11)
        mdays = 30;
```

```

    else mdays = 31;
}
else mdays = 0;
/* check date validity */
if (yy != 0 && (mm >= 1 && mm <= 12) && (dd      >= 1 && dd<= mdays))
    valid = 1;
else valid = 0;

```

Note that the code to determine the number of days in a month uses an **if-else-if** statement within an **if-else** statement. We can further simplify this code using a conditional expression as

determine *mdays*

```
valid = (yy != 0 && (mm > 1 && mm < 12) && (dd > 1 && dd < mdays)) ? 1 : 0;
```

The second algorithm presented in Program 7.2 to determine date validity uses the **if-else-if** statement. It can be simplified using the logical OR (**||**) operator as shown below:

determine *mdays*

```
if (yy == 0 || (mm < 1 || mm > 12) || (dd < 1 || dd > mdays))
    valid = 0
else valid = 1;
```

## 7.6.2 Nested Conditional Expression Operators

The C language allows nesting of conditional expression operators (**? :**). In the general format of the conditional expression given below,

*expr1* ? *expr2* : *expr3*

we can replace *expr2* and/or *expr3* with other conditional expressions. If we replace both *expr2* and *expr3* with conditional expressions, we get a two-level nested conditional operator expression, shown below.

*expr1* ? *expr2* ? *expr3* : *expr4* : *expr5* ? *expr6* : *expr7*

Observe that '?' and ':' are equal in number in such nested expressions. On the other hand, the number of **if** and **else** keywords in a *nested if* statements may not match as we can omit the **else** clause in one or more **if** statements.

As the conditional expression operator has right-to-left associativity, they are assigned to the operands from right to left. Note that a ':' in a conditional expression gets associated with the nearest '?' that is not yet associated with a '!'. This is similar to the association of the **else** clause in *nested if* statements. Thus, the above expression is equivalent to the following parenthesized expression which is more readable.

$$\text{expr1} ? (\text{expr2} ? \text{expr3} : \text{expr4}) : (\text{expr5} ? \text{expr6} : \text{expr7})$$

The execution of this statement proceeds as follows: Initially, *expr1* is evaluated. If it evaluates as true, *expr2* is evaluated. If *expr2* evaluates as true, the value of entire expression is equal to the value of *expr3*; otherwise it is equal to the value of *expr4*. However, if *expr1* evaluates as false, expression *expr5* is evaluated and depending on its value, the value of the entire expression is equal to either *expr6* or *expr7*. Thus, the meaning of the above two-level nested conditional operator expression can be expressed using the two-level *nested if* statement given below.

```
if ( expr1 )
  if ( expr2 )
    expr3;
  else expr4;
else
  if ( expr5 )
    expr6;
  else expr7;
```

However, note that they may not be always equivalent as the former is an expression, whereas the latter is a statement.

If we replace either *expr2* or *expr3* (but not both) in the general form  $\text{expr1} ? \text{expr2} : \text{expr3}$ , we have two other forms of the two-level nested conditional operator expression given below along with their equivalent parenthesized forms.

$$\begin{aligned}\text{expr1} ? \text{expr2} ? \text{expr3} : \text{expr4} : \text{expr5} &\equiv \text{expr1} ? (\text{expr2} ? \text{expr3} : \text{expr4}) : \text{expr5} \\ \text{expr1} ? \text{expr2} : \text{expr3} ? \text{expr4} : \text{expr5} &\equiv \text{expr1} ? \text{expr2} : (\text{expr3} ? \text{expr4} : \text{expr5})\end{aligned}$$

The meaning of these expressions can be expressed using *nested if* statements as shown below.

<pre>if ( expr1 )   if ( expr2 )     expr3 ;   else expr4 ; else expr5 ;</pre>	<pre>if ( expr1 )   expr2 ; else   if ( expr3 )     expr4 ;   else expr5 ;</pre>
--	--

By replacing one or more expressions in two-level nested conditional expressions, we get a higher level of nested conditional expressions. However, such expressions are difficult to understand and should generally be avoided. The expressions containing higher nesting levels may however be useful in *if-else-if* type situations. Consider the expression given below.

$$\text{expr1} ? \text{expr2} : \text{expr3} ? \text{expr4} : \text{expr5} ? \text{expr6} : \text{expr7} ? \text{expr8} : \text{expr9}$$

Observe that '?' and ':' alternate in this expression. This expression is equivalent to the parenthesized

expression given below.

```
expr1 ? expr2 : ( expr3 ? expr4 : ( expr5 ? expr6 : ( expr7 ? expr8 : expr9 )))
```

The meaning of this expression can be expressed using an **if-else-if** statement as shown below:

```
if ( expr1 )
    expr2;
else if ( expr3 )
    expr4;
else if ( expr5 )
    expr6;
else if ( expr7 )
    expr8;
else expr9;
```

### Example 7.11 Nested conditional operator expressions

#### a) Determine the maximum of three numbers

In Example 7.1, we used a two-level *nested if* statement to determine the maximum of three numbers. An equivalent code using nested conditional operators is given below.

```
max = a > b ? (a > c ? a : c) : (b > c ? b : c);
```

Since the conditional operator has a higher precedence than the assignment operator, the parentheses surrounding the expression to determine the maximum (i. e., the entire expression to the right side of the '=' operator) is unnecessary. However, we may include them to improve readability as shown below.

```
max = (a > b ? (a > c ? a : c) : (b > c ? b : c));
```

#### b) Count positive, negative and zero numbers in a given list

Let us use variables **pos**, **neg** and **zero** to represent the count of positive, negative and zero numbers in a given list of numbers. We can use an **if-else-if** statement to test the value of number **num** and increment the appropriate counter. We can also use a nested conditional expression to achieve the same result, as shown below.

```
(num > 0) ? pos++ : (num < 0) ? neg++ : zero++;
```

Note that the parentheses in the above statement are redundant. The code given below accepts *n* numbers from the keyboard and prints the counts of positive, negative and zero numbers in the given list.

```

pos = neg = zero = 0; /* initialize counters */
for (i = 0; i < n; i++) {
    scanf("%f", &num); /* read a number */
    (num>0) ? pos++ : (num < 0) ? neg++ : zero++;
}

printf("Positive: %d Negative: %d Zero: %d\n", pos, neg, zero);

```

**c)** Print the grade obtained by a student in a subject

Assume that we need to assign a grade to a student based on his/her marks in a single subject using the rules given below.

A+ : <i>marks</i> >= 85	A : 75 < <i>marks</i> < 85
B+ : 65 < <i>marks</i> < 75	B : 55 < <i>marks</i> < 65
C+ : 48 < <i>marks</i> < 55	C : 40 < <i>marks</i> < 48
D : 35 < <i>marks</i> < 40	F : <i>marks</i> < 35.

We can use a nested conditional operator to determine the grade as shown below.

```
(m >= 85) ? "A+" : (m >= 75) ? "A" : (m >= 65) ? "B+"
: (m >= 55) ? "B" : (m >= 48) ? "C+" : (m >= 40) ? "C"
: (m >= 35) ? "D" : "F"
```

We can then print this grade using a `printf` statement as shown below:

```
printf("Grade: %s", (m >= 85) ? "A+" : (m >= 75) ? "A"
: (m >= 65) ? "B+" : (m >= 55) ? "B" : (m >= 48) ? "C+"
: (m >= 40) ? "C" : (m >= 35) ? "D" : "F");
```

Note that if we wish to store the result string in an array of type the counts for odd and even digits, respectively, are `char`, we cannot use the assignment operator; instead we have to use `strcpy` function as shown below.

```
char grade[3];

strcpy(grade, (m >= 85) ? "A+" : (m >= 75) ? "A"
: (m >= 65) ? "B+" : (m >= 55) ? "B" : (m >= 48) ? "C+"
: (m >= 40) ? "C" : (m >= 35) ? "D" : "F"));
```

Finally, observe that the use of nested conditional operators result in concise code but is difficult to read and understand. Hence, such expressions should be avoided as far as possible.

## Exercises

1. Identify errors, if any, in the following **if-else-if** statements and rewrite them correctly.

```
a. if (ch > '0' && ch < '9')
    printf("Digit");
else if (ch > 'A' && ch < 'Z')
    printf("Uppercase letter");
else if (ch > 'a' && ch < 'z')
    printf("Lowercase letter");
else printf("Not a digit/letter");

b. if (m > 35)
    printf("Pass");
else if (m > 50)
    printf("Second class");
else if (m > 60)
    printf("First class");
else printf("First with dist");
```

2. Identify errors, if any, in the following program segments and rewrite them using a single **if-else-if** statement.

```
a. /* determine discount and cost */
if (qty >= 10) {
    discount = 0.05;
    if (qty >= 50)
        discount = 0.1";
    if (qty >= 100)
        discount = 0.2;
}
else discount = 0.0;
cost = qty * rate * (1-discount);

b. /* determine grade in an exam
if (marks >= 80 || marks <= 100)
    grade = 'A';
if (marks >= 60 || marks <= 79)
    grade = 'B';
if (marks >= 40 || marks <= 59)
    grade = 'C';
else grade = 'F';
/* print grade */
printf("Grade: %s", grade);

c. /* Calc Public Provident Fund */
```

```

if (basic_pay ≥ 10000)
    ppf = 500;
if (basic_pay ≥ 5000)
    ppf = 250;
else ppf = 100;
/* calc Profession Tax */
if (basic_pay ≥ 10000)
    pt = 200;
if (basic_pay >= 5000)
    pt = 100;
else pt = 50;

d. /* Print day name */
if (day = 0)
    printf("Sunday");
if (day = 1)
    printf("Monday");
if (day = 2)
    printf("Tuesday");
... /* skipped to save space */

if (day = 6)
    printf("Saturday");
else printf("Invalid day");

```

3. Determine the purpose and the output of the following program segments:

- ```
for (i = 1; i < 50; i++) {
    if (i % 3)
        printf("%d ", i);
}
```
- ```
for (j = 10; j <= 99; j++)
if (j % 10 + j / 10 == 9)
printf("%d ", j);
```
- ```
scanf ("%d", &n);
osum = esum = 0;
for (i = 0; i < n; i++) {
    scanf("%d", &m);
    if (m % 2) osum += m;
    else esum += m;
}
printf("%d %d", osum, esum);
```
- ```
for(ch = 'A'; ch <= 'Z'; ch++)
```

```

if (!(ch == 'A' && ch == 'E' &&
     ch == 'I' && ch == 'O' &&
     ch == 'U'))
    printf("%d ", ch);

```

4. Determine the output of the following program segments containing nested **for** loops:

a. 

```
for (r = 1; r < 10; r *= 2)
    for (c = 11; c < 15; c += 2)
        printf(" %3d", r * c);
```

b. 

```
m = 5;
for (r = 1; r <= m; r++)
    for (c = 1; c <= r; c++)
        printf("*");
printf("\n");
```

c. 

```
m = 5;
for (i = 0; i < m; i++) {
    for (k = 0; k < m; k++)
        puts(i == j ? " 1": " 0");
    printf("\n");
}
```

d. 

```
m = 4;
for (i = 0; i < m; i++) {
    for (k = 0; k < m; k++)
        if (i + k == m - 1)
            printf("*");
        else printf(".");
    printf("\n");
}
```

e. 

```
for (i = 0; i < 4; i++) {
    int len = 3 + 4 * ((float) rand() / RAND_MAX);
    for (j = 0; j < len; j++)
        printf("*");
    printf("\n");
}
```

5. Identify the errors in these program segments and rewrite them correctly.

a. 

```
/* Test for a palindrome */
int num, revnum;
while(num >= 0) {
    digit = num % 10;
    revnum += digit * 10;
```

```

num /= 10;
if (revnum == num)
    printf("Palindrome");
else printf("Not Palindrome");
}

b. // Test for Armstrong no.
int n, cubesum = 0;

scanf("%d", &n);
while (n > 0) do
{
    cube_sum += n % 10*n % 10*n %10;
    if (cube_sum == n) {
        printf("Armstrong No.");
    }
}

```

5. Determine the output of the following program segments:

- a. char ch;
`for (ch = 0; c < 'z'; ch++)
 if (!isprint(ch))
 continue;
 printf("%c", c);`
- b. char ch;
`for (ch = 'A'; ch < 'Z'; ch++) {
 if (ch == 'A' || ch == 'E' ||
 ch == 'I' || ch == 'O' ||
 ch == 'U') continue;
 putchar(ch);
}`
- c. m = 5;
`for (i = 0; i < m; i++) {
 for (k = 0; k < m; k++) {
 if (i + k > 5) continue;
 printf("%d ", i+k);
 }
 printf("\n");
}`
- d. m = 4;
`for (i = 0; i < m; i++) {
 for (j = 0; j < m; j++)
 if (i+1 == j)`

```
        break;
    printf("*");
}
```

7. Write complete programs for the following problems.
- Accept  $n$  numbers from the keyboard and calculate the sum of odd numbers and sum of even numbers in the list.
  - Accept several numbers and print only those having even sum of digits.
  - Accept several numbers and determine the minimum and maximum of them.
  - Accept the marks of a student in six subjects in the HSC examination and determine whether he/she has passed or not.

## Exercises (Advanced Concepts)

1. Rewrite the following program segments using logical operators:

```
a. if (a >= 0) {
    if (a <= 1.0)
        printf("Valid data");
}
else printf("Invalid data");

b. leap = 0;
if (year % 4 == 0) {
    if (year % 100 != 0)
        leap = 1;
}
if (year % 400 == 0)
    leap = 1;

c. if (month == 1)
    days = 31;
else if (month == 2)
    days = 28;
... /* skipped to save space */
...
else if (month == 12)
    days = 31;
printf("Month days: %d\n", days);

d. vowel = 0;
ch = toupper(ch);
if (ch == 'A') vowel = 1;
```

```

if (ch == 'E') vowel = 1;
if (ch == 'I') vowel = 1;
if (ch == 'O') vowel = 1;
if (ch == 'U') vowel = 1;
if (vowel) printf("vowel");
else printf("not a vowel");

e. int primary_color = 0;
   if (color == 'R') /* Red */
      primary_color = 1;
   if (color == 'Y') /* Yellow */
      primary_color = 1;
   if (color == 'B') /* Blue */
      primary_color = 1;

f. /* Decide "Tution fee waiver" scheme
   for Engg Adm - Not actual rules*/
   tfw = 'N';
   if (pcm_marks >= 200) {
      if (cet_marks >= 150) {
         if (ebc == 'Y')
            tfw = 'Y';
      }
   }
}

```

2. Write the following code using the nested conditional expression operators:

- ```

if (num > 0)
   pos = pos + 1;
else if (num < 0)
   neg = neg + 1
else zero = zero + 1;

```
- ```

if (marks < 35) puts("Fail");
else if (marks < 50) puts("Pass");
else if (marks < 60) puts("Second");
else if (marks < 75) puts("First");
else puts("Distinction");

```

# 8 functions

In Chapter 4, we studied the standard library functions that enable us to use the functionality defined in the standard C library. This chapter presents the user-defined functions that allow us to simplify the task of program development by decomposing it into smaller sub-programs and writing functions for each sub-program.

The first section revises the concept of functions. It discusses their advantages and the mechanism of a function call. The next section discusses user-defined functions. It covers various aspects including function definition, `return` statement and function call.

The advanced section discusses how function parameters can be used as a loop variable, `const` parameters and storage classes.

## 8.1 About Functions

A **function** is a subprogram used to perform a specific operation. We are already familiar with the concept of functions in C. Every C program contains at least one function called `main`. In addition, we have studied several commonly used standard library functions in Chapter 4 such as `scanf`, `printf`, `getchar`, `putchar`, `sqrt`, `log`, `sin`, `cos`, etc.

### 8.1.1 Function Call

Recall that a **function call** takes the form

*func\_name ( arg\_list )*

where *func\_name* is the name of the function being called and *arg\_list* is a commaseparated list of **arguments**. The number of arguments, their types and order must be in accordance with the function **parameters** specified in the function definition. When a function is called, the values specified in *arg\_list* are passed to the function. The called function will usually use or process these values in some way.

A function may return a value. When it does, we can call that function from within an expression. If a function does not return a value or if we are not interested in the value returned, a function call takes the form of a C statement as in

```
func_name( arg_list );
```

The examples given below illustrate the various ways in which a function `func` that returns a value can be called. Note that an argument may be a constant, variable or expression.

```
x = func(3);           /* RHS of an assignment statement */
x = c + func(a+b);    /* as an operand in an expression */
printf("%d", func(b)); /* as an argument for another function */
func(a - b * c);     /* ignore return value */
```

When a function is called, execution of the current function is suspended. The argument expressions (if present) are evaluated and their values assigned to the corresponding function parameters, and program control is transferred to the called function. The statements in the called function are then executed, starting from the first executable statement until a `return` statement is encountered or all the statements have been executed.

When the execution of the called function is complete, control is transferred to the calling function to the point from where the function was called. The return value, if any, is returned in place of the function call.

### 8.1.2 Advantages of Functions

There are several advantages in using functions. They are discussed below.

1. Functions allow the ***divide and conquer*** strategy to be used for the development of programs. When developing even a moderately sized program, it is very difficult if not impossible, to write the entire program as a single large `main` function. Such programs are very difficult to test, debug and maintain. The task to be performed is normally divided into several independent subtasks, thereby reducing the overall complexity; a separate function is written for each subtask. In fact, we can further divide each subtask into smaller subtasks, further reducing the complexity.
2. Functions help *avoid duplication* of effort and code in programs. During the development of a program, the same or similar activity may be required to be performed more than once. The use of functions in such situations avoids duplication of effort and code in programs. This reduces the size of the source program as well as the executable program. It also reduces the time required to write, test, debug and maintain such programs, thus reducing program development and maintenance cost.
3. Functions enable us to *hide the implementation details* of a program, e. g., we have used library functions such as `sqrt`, `log`, `sin`, etc. without ever knowing how they are implemented. However, although we need to know the implementation details for user-defined functions, once a function is developed and tested, we can continue to use it without going into its implementation details. Another consequence of hiding implementation details is improvement in the readability of a program. Proper use of functions leads to programs that are easier to read and understand.
4. The divide and conquer approach also allows the parts of a program to be developed, tested and debugged independently and possibly concurrently by members of a programming team. The

involvement of several programmers, which is the norm in the development of a software project, reduces the overall development time.

- 5. The functions developed for one program can be used, if required, in another with little or no modification. This further reduces program development time and cost.

## 8.2 User-defined Functions

C allows programmers to define their own functions. Such functions are called **user defined functions**. In fact, the **main** function that must be present in every C program is a user-defined function. A programmer may define additional functions in the following situations:

- 1. The required functionality is not available as a library function either in the standard C library or in the additional libraries supplied with the C compiler. It is a good idea to take a quick look at the summary of functions and other facilities available in the standard C library given in Appendix D.
- 2. Some functionality or code is repeated in a program with little or no modification.
- 3. An existing function is quite large. It is a good idea to divide such functions, if possible, into smaller ones.

### 8.2.1 Function Definition

The general format to define a function is as follows:

```
ret_type func_name (param_list)
{
    declarations
    executable_statements
}
```

where *func\_name* is the name of the function being defined, *ret\_type* is the type of value returned by the function (also called **function type**) and *param\_list* is a comma-separated list of function **parameters**. For each parameter, we must specify the type of the parameter followed by its name. Thus, the *param\_list* takes the following form:

*type<sub>1</sub>* *param<sub>1</sub>*, *type<sub>2</sub>* *param<sub>2</sub>*, ..., *type<sub>n</sub>* *param<sub>n</sub>*

where *type<sub>1</sub>*, *type<sub>2</sub>*, ... are the types of parameters *param<sub>1</sub>*, *param<sub>2</sub>*, ..., respectively. Note that the rules for naming these parameters (as well as the function name) are the same as those for a variable name and that the type has to be specified separately for each parameter, even if consecutive formal parameters in *param\_list* are of the same type.

A function may not have any parameters. This is indicated by using the keyword **void** in place of *param\_list* in the function definition. However, **void** may also be omitted in such cases.

*ret\_type* is the type of the value returned by a function. A function can return only one value or none at all. If a function does not return a value, it is indicated by writing the keyword `void` in place of *ret\_type*. Also note that *ret\_type* may be omitted, in which case, the function is assumed to return a value of type `int`.

The body of a function consists of declarations followed by executable statements enclosed within braces '{' and '}'. The entities declared within the function body, which are usually variables, are *local* to the function being defined, i. e., they are accessible only within the function body and not outside it. We can declare a local variable (or some other entity) with the same name as that of a function. The formal parameters of a function are treated as local variables declared in the beginning of the function body.

The executable statements in the definition of a function can be one or more valid C statements. When a function is called, these statements are executed until a return statement (discussed shortly) is encountered or all the executable statements are executed.

### Example 8.1 Function parameters and return types

The examples given below illustrate how parameters and return types are specified in a function definition. In these examples, the function body is empty as indicated by ellipsis (...) and is written on the same line to save space.

#### a) Function to print a line on the screen

```
void print_line(void) { ... }
```

In this example, the function `print_line` does not have any parameters and does not return any value as indicated by the `void` keywords. Alternatively, we can write this function definition as follows:

```
void print_line() { ... }
```

Note that the pair of parenthesis is required after the function name even if the parameter list is empty. In fact, it is the parenthesis '(' that identifies `print_line` as a function name.

#### b) Function to calculate the area of a circle

```
float area(float radius) { ... }
```

In this example, the function `area` has a single parameter, `radius` of type `float`, and returns a value of type `float`.

#### c) Function to determine the maximum of three integers

```
int max3(int a, int b, int c) { ... }
```

In this example, function `max3` has three parameters `a`, `b`, and `c` (all of type `int`) and returns a value

of type int. As the default return type for a function is int, the return type may be omitted.

A beginner often forgets that a data type must be specified for each formal parameter and may use the variable declaration syntax to declare the function parameters as shown below:

```
/* wrong definition - types missing for parameters b and c */
int max3(int a, b, c) { ... }
```

### Example 8.2 Function definition

#### a) Function to print a welcome message

```
void welcome()
{
    printf("Hello friends,\n");
    printf("Welcome to the wonderful world of C.\n");
}
```

This function illustrates how we can use a function to display a particular message or text on the screen. The function has no parameters and does not return any value. It also does not declare any local variables. When this function is called, the printf statements within the function body are executed causing the following message to be displayed on the screen:

```
Hello friends,
Welcome to the wonderful world of C.
```

We can write similar functions to display a help message, table report heading, etc.

#### b) Function to print the area and circumference of a circle

```
void area_circum(float radius)
{
    const float PI = 3.1415927;
    float area, circum;

    area = PI * radius * radius;
    circum = 2 * PI * radius;
    printf("Radius      : %.3f\n", radius);
    printf("Area       : %.3f\n", area);
    printf("Circumference: %.3f\n", circum);
}
```

This example defines a function named area\_circum that accepts the radius of a circle in parameter

**radius** of type **float** and prints its area and circumference. The function does not return any value. It declares a constant **PI** of type **float** with value 3.1415927 and two variables **area** and **circum** also of type **float**. The function calculates the values of **area** and **circum** and prints them along with the radius of the circle.

### c) Function to print a line of a given character

We may often require a horizontal line of a dash, underline or some other character to be printed, particularly when printing results in a tabular format. Consider a situation in which we require a line containing a fixed number of a specific character (e. g., 65 dash characters) to be printed several times. We can define a function to print such a line, as shown below:

```
void print_65_dash_line()
{
    int i;
    for (i = 0; i < 65; i++)
        putchar('-');
    putchar('\n');
}
```

The function **print\_65\_dash\_line** does not have any parameter and it does not return any value. The body of the function first declares index variable **i**. Then it prints a line containing 65 dashes followed by a newline character. Although we could have used a single **printf** statement to print the desired line of dashes, the **for** loop allows us to quickly change the line length. However, if we require lines of different lengths to be printed, it is better to pass the line length as a parameter to this function. The modified function **print\_dash\_line** is given below.

```
void print_dash_line(int len)
{
    int i;
    for (i = 0; i < len; i++)
        putchar('-');
    putchar('\n');
}
```

Now we can further generalize this function to accept a character to be used to print a line in addition to the line length as shown below.

```
void print_line(int len, char ch)
{
    int i;
    for (i = 0; i < len; i++)
```

```
    putchar(ch);
    putchar('\n');
}
```

The order of the parameters (`len` as first parameter and `ch` as the second) is the programmer's choice. If we change their order, we do not have to modify the function body. However, all function calls and declarations will need to be modified accordingly. Hence, it is a good idea to decide carefully the order of the function parameters.

Finally, note that sometimes we may like to avoid printing a newline character at the end of a line. Hence, we can further generalize the above function by including an additional parameter that gives us a control over the printing of a newline character, as shown below.

```
void print_line(int len, char ch, int print_nl)
{
    int i;
    for (i = 0; i < len; i++)
        putchar(ch);
    if (print_nl)
        putchar('\n');
}
```

### 8.2.2 The **return** Statement

The **return statement** is used to terminate the execution of a function and transfer program control back to the calling function. In addition, it can specify a value to be returned by the function. A function may contain one or more **return** statements. The general format of the **return** statement is given below.

```
return expr;
```

where *expr* is the expression whose value is returned by the function. The execution of the **return** statement causes the expression *expr* to be evaluated and its value to be returned to the point from which this function is called. The expression *expr* in the **return** statement is optional and if omitted, program control is transferred back to the calling function without returning any value.

As we know, a function returns a value if the return type other than **void** is specified in the function definition or if the return type is omitted. Such a function must return a value using the **return** statement(s). Note that a function can return only a single value through its name.

If a function does not return any value, the **return** statement may be omitted, in which case all the statements within the function body are always executed before control is transferred back to the calling program. However, we can include one or more **return** statements (without *expr*) to transfer control back to the calling function.

### Example 8.3 Functions returning a value

#### a) Function to return the larger of two numbers

The function given below accepts two numbers of type **double** and returns the larger of them which is also a value of type **double**.

```
double large(double a, double b)
{
    double large;

    if (a > b)
        large = a;
    else large = b;
    return large;
}
```

The function declares a local variable named **large**, the larger of the two numbers **a** and **b**. Note that its name is the same as that of the function. The value of variable **large** is determined using an **if-else** statement and then returned using a **return** statement.

As a function can contain several **return** statements, we can simplify this function by eliminating the variable **large** and returning the values from both the **if** and **else** clauses as shown below.

```
double large(double a, double b)
{
    if (a > b)
        return a;
    else return b;
}
```

We can further simplify this function by using the conditional expression operator (**?:**) within a single **return** statement as shown below.

```
double large(double a, double b)
{
    return a > b ? a : b;
}
```

#### b) Function for leap year test

A leap year is one that is divisible by 4 but not by 100, or it is divisible by 400. The function **is\_leap**

to test whether a given year is leap or not is given below.

```
int is_leap(int y)
{
    if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)
        return 1;
    else return 0;
}
```

Note that the function returns an integer value 1 (true) if the year  $y$  is leap and 0 (false) otherwise. This function can also be simplified using the conditional expression operator (?:) as shown below.

```
int is_leap(int y)
{
    return (y % 4 == 0 && y % 100 != 0 || y % 400 == 0) ? 1 : 0;
}
```

Finally, note that the logical expression

$$(y \% 4 == 0 \&\& y \% 100 != 0) \mid\mid y \% 400 == 0)$$

evaluates as true (1) when the year is leap and false (0) otherwise. Thus, we can directly return the value of the logical expression eliminating the conditional operator as shown below.

```
int is_leap(int y)
{
    return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
}
```

### c) Function to return maximum of three numbers

A function to return the maximum of three numbers is given below. It accepts three parameters  $x$ ,  $y$ , and  $z$ , each of type **double**, and returns a value of type **double**.

```
double max3(double x, double y, double z)
{
    double max;

    if (x > y)
        max = x;
    else max = y;

    if (z > max)
```

```
    max = z;  
  
    return max;  
}
```

A local variable **max** is used to hold the maximum of the three given numbers. Initially, the larger of **x** and **y** is stored in variable **max** using an **if-else** statement. Then the value of **z** is assigned to **max** if it is greater than **max**. Finally, the value of **max** is returned.

#### d) Function to return the sum of digits of an integer number

The function given below calculates the sum of digits of a given integer number **n** using the logic explained in Program 6.5 and returns the value.

```
int digit_sum(int n)  
{  
    int sum = 0; /* digit sum */  
    while (n > 0) {  
        sum += n % 10;  
        n /= 10;  
    }  
    return sum;  
}
```

#### e) Function to return the factorial of an integer number

The function **fact** given below accepts an integer number, calculates its factorial as explained in Program 6.1 and returns the result of type **long int**.

```
long int fact(int n)  
{  
    int i;  
    long int fact = 1;  
  
    for (i = n; i > 1; i--)  
        fact *= i;  
    return fact;  
}
```

Note that the factorial is calculated in local variable **fact** (same name as the function) of type **long int**.

### 8.2.3 Function Declaration or Prototype

If a function call precedes its definition in a program, we should declare the function before the call. A **function declaration** takes the following general form:

```
ret_type func_name (param_type_list );
```

where *func\_name* is the name of the function, *ret\_type* is the type of the return value and *param\_type\_list* is a comma-separated list of function parameter types. Note that this format is similar to the function definition except that the parameter declaration list is replaced by the parameter type list and the function body is replaced by a semicolon. Also note that formal parameter names may be included in a function declaration and are treated as comments. They may be replaced with other names as well. Thus, to include a function declaration in a program, we can simply copy the first line of the function definition (and insert a semicolon after it).

Function declarations are usually written at the beginning of a program, before the `main` function. Such declarations outside any function are called global declarations. A function may also be declared within the function from which it is called. The problem with such a **local declaration** is that its scope is restricted to the function in which it is declared, i. e., it is available only within that function. Thus, if this locally declared function is also called from some other function, we must provide its declaration in that function as well.

Although a program can contain only one definition of a function, there is no restriction on the number of declarations. A function may be declared more than once within the same scope. However, note that each declaration must be consistent with its definition with regard to the number of parameters, their types and the return type.

A function declaration provides valuable information (function name, number and type of parameters and return type) to the compiler. The compiler uses this information to ensure that the function call is consistent with its definition. Thus, the compiler can check that the number of arguments in a function call and the types of arguments are as per the requirements of the function and that the function call is consistent with respect to the type of the value returned by the function. The compiler can report errors or warnings when a function call is not consistent with its definition. Note that if a function definition precedes its calls, the compiler obtains the necessary information from the function definition itself thereby eliminating the need for a function declaration. However, this is not good programming practice.

If the type of an argument in a function call is different from the type of the corresponding parameter in function definition, automatic conversion is applied if possible, to convert the argument to the type of the parameter. If automatic conversion is not possible, the compiler will report an error. Also note that if a function declaration or definition is not available to a compiler before the function call is encountered, the compiler can neither perform such automatic conversions nor it can report any errors if such conversions are not possible. The compiler may, however, give a warning that a function call is encountered without a function prototype. The compiler assumes that this function returns a value of type `int`. Thus, if the function actually returns a different type of value, the compiler can report an error or warning.

#### Example 8.4 Function declarations or prototypes

The declarations of the `large`, `is_leap` and `max3` functions in Example 8.3 are given below.

```
double large(double, double);
int is_leap(int);
double max3(double, double, double);
```

As the formal parameter names can be included, these declarations may be alternatively written as follows:

```
double large(double a, double b);
int is_leap(int y);
double max3(double x, double y, double z);
```

Consider the program segment given below which contains several declarations of the function `fact` – two global declarations before the `main` function and two local declarations in the `main` function. This program segment is perfectly valid as all the declarations of function `fact` are consistent. Note that data type `long` is the same as type `long int`.

```
long int fact(int n);
long int fact(int number);

int main()
{
    long int fact(int);
    long fact(int n);
    ...
}
```

### 8.3 Program Structure

A program may contain any number of functions in addition to the `main` function. There are two approaches to write such a program: top-down and bottom-up.

In the **top-down approach**, which is more convenient and is used in this book, the `main` function is written before other user-defined functions. The advantage of this approach is that we can quickly locate the `main` function, which contains the top-level code of the program. However, the user-defined functions will obviously be used before they are defined. Hence, we need to declare these functions before calling them. As these functions may be used in the `main` function and in other functions, they are usually declared before the `main` function. Thus, a program takes the following general format:

include directives

global declarations

function declarations (i. e. prototypes)

```
int main()
{
```

...

```
}
```

function definitions

The top-level functions, i. e., the functions called from the `main` function, will be defined first followed by the functions that are called from these top-level functions and so on. It is a good idea to provide function declarations in the order in which these functions are defined.

In the **bottom-up approach**, we need to define each function before its use. The advantage of this approach is that function declarations are not required. However, this approach implies that the `main` function is defined at the end of the program, the top-level functions called from the `main` function are defined before the `main` function and so on.

Finally, note that it is also possible to define functions in any order. In this approach, we require a prototype for each function that is called before its definition.

### Program 8.1 Factorial of an integer number

Consider the program given below, written using the top-down approach, that calculates the factorial of a given integer number using function `fact` defined in Example 8.3e. Function `fact` is called in the `main` function from a `printf` function call and the `long int` value returned by it is printed using the `%ld` conversion specification.

```
/* Factorial of an integer number using a function: top-down approach
#include <stdio.h>

long int fact(int n); /* prototype for function fact */

int main()
{
    int n;

    printf("Enter a small integer number: ");
    scanf("%d", &n);
    printf("%d! = %ld\n", n, fact(n));

    return 0;
}
```

```
/* return factorial of an integer number */
long int fact(int n)
{
    int i;
    long int fact = 1;
    for (i = n; i > 1; i--)
        fact *= i;

    return fact;
}
```

Observe that the **main** function is defined first followed by the function **fact**, which is called before it is defined. Thus, we need to declare **fact** in the **main** function or before it. In the program given above, **fact** is declared before the **main** function. The program output is given below.

```
Enter a small integer number: 10
10! = 3628800
```

A bottom-up implementation of the above program is given below in which **fact** is defined first followed by the **main** function. As the definition of **fact** is available before its call in the **main** function, a prototype is not required.

```
/* Factorial of an integer number using a function: bottom-up approach
#include <stdio.h>

/* return factorial of an integer number */
long int fact(int n)
{
    int i;

    long int fact = 1;
    for (i = n; i > 1; i--)
        fact *= i;

    return fact;
}

int main()
{
    int n;

    printf("Enter a small integer number: ");
    scanf("%d", &n);
```

```
    printf("%d! = %ld\n", n, fact(n));
}
```

## Program 8.2 Area and circumference of a circle

Let us write a program to calculate the area and circumference of a circle using user-defined functions. A function `area_circum` to accept the radius of a circle and print its area and circumference is given in Example 8.2. However, rather than printing the results in a function, it is good practice to return the results to the calling function, where they can be printed or used for further processing, if required. Hence, let us return the results (area and circumference) to the calling function.

As a function can return only one value through its name, we will require two functions in this program to calculate the area and circumference: `area` and `circum`. The program given below first defines the `main` function followed by these functions. The declarations of `area` and `circum` are given before the `main` function. Also, `PI` is defined as a global constant as it is required in both functions.

```
/* Calculate area and circumference of a circle using functions */
#include <stdio.h>

const float PI = 3.1415927;

float area(float radius);
float circum(float radius);

int main()
{
    float radius;

    printf("Enter radius: ");
    scanf("%f", &radius);
    printf("Area : %.3f\n", area(radius));
    printf("Circumference: %.3f\n", circum(radius));

    return 0;
}

/* return area of a circle */
float area(float radius)
{
    return PI * radius * radius;
}
```

```
/* return circumference of a circle */
float circum(float radius)
{
    return 2 * PI * radius;
}
```

Note that we can change the order of definitions of `area` and `circum`. If we do so, we can also change the sequence of their declarations, though it is not necessary. Note that we can also use a bottom-up approach and first define the functions `area` and `circum` in any order followed by the `main` function. As we know, this approach does not require declarations of `area` and `circum`.

## 8.4 Methods of Parameter Passing

The C language supports two methods for passing parameters to a function: *call by value* and *call by reference*. **Call by value** is the default method for passing parameters. In this method, the values of the arguments in a function call are assigned (i. e., copied) to the corresponding function parameters when a function is called. The parameters and arguments are thus different entities. If we modify the parameter values within the body of a function, the values of the arguments are not affected. Thus, such parameters can be considered as input only parameters. They allow us to pass the information from a calling function to the called function, but not in the reverse direction. The functions defined in this chapter use the call by value mechanism, which is the default.

The **call by reference** mechanism involves the use of pointers and arrays. This will be covered in Chapter 11.

### Example 8.5 Call by value mechanism

Consider the program given below.

```
#include <stdio.h>

void func(int a, int b);

int main()
{
    int a = 10, b = 20;
    printf("In main(), before function call : a = %d b = %d\n");
    func(a, b);
    printf("In main(), after function call : a = %d b = %d\n");

    return 0;
}
```

```

void func(int a, int b)
{
    printf("In func(), before initialization: a = %d b = %d\n");
    a = b = 30;
    printf("In func(), after initialization : a = %d b = %d\n");
}

```

This program defines a function named `func` which has two parameters, `a` and `b` of type `int`. Within its body, this function first prints these parameter values, initializes them to 30 and prints them again. The `main` function declares two variables, `a` and `b` of type `int`. It first prints the values of these variables and then calls the function `func` with variables `a` and `b` as its arguments. Finally, the `main` function prints the values of variables `a` and `b` again. The output of this program is given below.

```

In main(), before function call : a = 10 b = 20
In func(), before initialization: a = 10 b = 20
In func(), after initialization : a = 30 b = 30
In main(), after function call  : a = 10 b = 20

```

Observe from output lines 2 and 3 that the values of variables `a` and `b` are passed to the function and are then initialized to 30. However, the values of variables `a` and `b` in the `main` function are unaffected after the function call. This is expected, as function `func` uses the default call by value mechanism, making `a` and `b` input-only parameters. Also note that although the function parameters and the argument variables have same name, they are different entities, and modifying the function parameters does not affect the argument variables.

## 8.5 Recursion

We have seen that a function can be called from another function. In fact, the C language allows a function to be called from within itself. Such a function is called a ***recursive function***.

Several problems are naturally expressed in recursive form. For example, the recursive definition to determine the factorial of an integer number is given as

$$n! = n * (n-1)!$$

Thus, a larger problem of determination of  $n!$  is expressed in terms of a smaller problem of determination of  $(n-1)!$ . By the above definition, we can calculate  $(n-1)!$  as  $(n-1) * (n-2)!$ . This definition can be used repetitively to calculate the value of  $n!$  as shown below.

$$\begin{aligned} n! &= n * (n - 1)! \\ &= n * (n - 1) * (n - 2)! \end{aligned}$$

$$= n * (n - 1) * (n - 2) * (n - 3)!$$

...

Obviously, this repetitive substitution should stop somewhere. Thus, a recursive definition must have a *stopping criterion*. As  $0! = 1$ , the complete recursive definition for  $n!$  is thus given as

$$\begin{aligned} n! &= n * (n - 1)! && \text{if } n > 0 \\ &= 1 && \text{if } n = 0 \end{aligned}$$

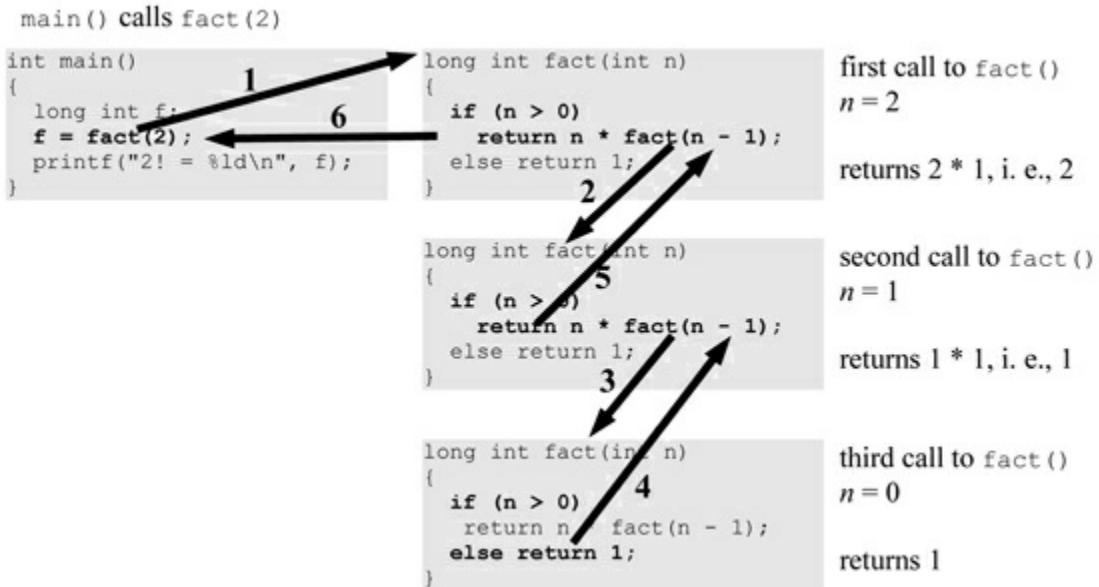
Once a recursive definition for the solution of a problem is available, writing recursive functions is very straightforward. A recursive function **fact** to determine the factorial of a given non-negative integer number **n** is given below.

```
/* recursive function to determine factorial of integer number n */
long int fact(int n)
{
    if (n > 0)
        return n * fact(n - 1);
    else return 1;
}
```

Note that the function returns a **long int** value since the factorial numbers grow very rapidly. Now to understand how this function works, assume that it is called from the **main** function, as shown below.

```
int main()
{
    long int f;
    f = fact(2);
    printf("2! = %ld\n", f);
}
```

The execution of this program is illustrated in Fig. 8.1 in which the arrows indicate the transfer of control from one function to another and the statements executed are shown in bold face. When function **fact** is called (from the **main** function), parameter **n** is assigned the argument value (i. e., 2) and control is transferred to its beginning. As the value of **n** is greater than 0, the **fact** function is called a second time from the **if** statement.



**Fig. 8.1 Execution of a recursive function fact**

Control is again transferred to the beginning of function **fact**, with argument value  $n - 1$ , i.e., 1. Thus, the function parameter **n** assumes value 1 in this function call. Since this value is greater than 0, the **fact** function is called again (third time) with argument value 0.

Control is again passed to the beginning of **fact** function and the parameter **n** is assigned the value 0. Now since the parameter value equals 0, the condition in the **if** statement evaluates as false and the function returns value 1. This value is returned to the second call of the **fact** function (for which **n** had value 1).

Now the expression  $n * \text{fact}(n-1)$  is evaluated as  $1 * 1$ , i.e., 1 and the value is returned by this function call to the calling function, i.e., the first call of the **fact** function in which the parameter **n** had a value of 2.

Now the expression  $n * \text{fact}(n-1)$  is evaluated again in the first call of the **fact** function as  $2 * 1$ , i.e., 2 which is returned to the **main** function where it is printed using the **printf** function.

Note that the **fact** function given above can be written concisely using the conditional expression operator as shown below.

```

long int fact(int n)
{
    return (n > 0) ? (n * fact(n - 1)) : 1;
}

```

Finally, consider the function definition given below in which the function name has been reused as a

local variable name.

```
long int fact (int n)
{
    long int fact;
    if (n > 0)
        fact = n * fact(n-1);
    else fact = 1;
    return fact;
}
```

As the function name has been redefined as a local variable, the compiler gives an error when the function is recursively called. The solution to this problem is to use different names for function and local variables.

#### Example 8.6 Examples of recursive functions

##### a) Sum of digits of an integer number

A recursive approach to determine the sum of digits of a non-negative integer number  $n$  can be given as the addition of the rightmost digit of  $n$  and the sum of digits of the number obtained by removing this digit from  $n$ , i. e.,  $n/10$  (integer division). Note that this definition is applicable as long as  $n$  has more than one digits, i. e., it is greater than 9. The terminating condition is  $n \leq 9$ , i.e., when  $n$  has only one digit in it. In this case, the sum of digits is  $n$  itself. A recursive definition to calculate the sum of digits of a non-negative number is given below.

$$\begin{aligned} \text{digitsum}(n) &= n \% 10 + \text{digitsum}(n / 10) && \text{if } n > 9 \\ &= n && \text{otherwise} \end{aligned}$$

A recursive definition for the `digitsum` function is given below.

```
/* determine sum of digits using recursive function */
int digitsum(int n)
{
    if (n > 9)
        return n % 10 + digitsum(n / 10);
    else return n;
}
```

##### b) GCD of two numbers

A recursive definition to calculate the greatest common divisor (GCD) of two integer numbers  $a$  and  $b$  is given below, followed by the definition of the recursive function.

$$\begin{aligned} \gcd(a, b) &= \gcd(b, a \% b) && \text{if } b \neq 0 \\ &= a && \text{otherwise} \end{aligned}$$

```
/* determine GCD using recursive function */
int gcd(int a, int b)
{
    if (b != 0)
        return gcd(b, a % b);
    else return a;
}
```

### c) Calculate $x^n$ using a recursive function

A recursive definition to calculate the value of  $x^n$  where  $x$  is a real number and  $n$  is a non-negative integer is given below.

$$\begin{aligned} x^n &= x * x^{n-1} && \text{if } n > 0 \\ &= 1 && \text{if } n = 0 \end{aligned}$$

A recursive function `power` that uses the above recursive definition to calculate  $x^n$  is given below.

```
double power(double x, int n)
{
    if (n == 0)
        return 1;
    else return x * power(x, n - 1);
}
```

It can be easily verified that the above function requires  $n$  multiplications to determine  $x^n$ . Thus, it is inefficient compared to the iterative version due to overheads associated with the recursive calls.

However, we can write an efficient version of this function by noting the fact that the value of  $x^n$  can be calculated as  $(x^2)^{n/2}$  when  $n$  is even and as  $x (x^2)^{n/2}$  when  $n$  is odd. The complete recursive definition including the stopping condition is given below.

$$\begin{aligned} x^n &= 1 && \text{if } n = 0 \\ &= (x^2)^{n/2} && \text{if } n \text{ is even} \\ &= x * (x^2)^{\lfloor n/2 \rfloor} && \text{otherwise, i. e., if } n \text{ is odd} \end{aligned}$$

This definition is very efficient and requires very few multiplications compared to the iterative method or the recursive method given above. For example, using this definition, the value of  $x^{32}$  can be calculated using only 5 multiplications (instead of 32) as shown below:

$$x^{32} = (x^2)^{16} = ((x^2)^2)^8 = (((x^2)^2)^2)^4 = ((((x^2)^2)^2)^2)^2$$

Similarly, we can verify that value of  $x^{31}$  can be calculated using only 8 multiplications as shown below.

$$x^{31} = x(x^2)^{15} = x(x^2(x^4)^7) = x(x^2(x^4(x^8)^3)) = x(x^2(x^4(x^8(x^{16}))))$$

Here, one multiplication is required for calculating each terms in the above expansion ( $x^2$ ,  $x^4$ ,  $x^8$  and  $x^{16}$ ) and four multiplications are required to multiply these terms. An efficient implementation using this recursive definition is given below.

```
double power(double x, int n)
{
    if (n == 0)
        return 1;
    else if (n % 2 == 0)
        return power(x * x, n / 2);
    else return x * power(x * x, n / 2);
}
```

### Program 8.3 Tower of Hanoi

Write a program to obtain the solution for the *Tower of Hanoi* problem.

**Solution:** The Tower of Hanoi problem consists of three poles, *left*, *middle*, and *right*. One of the poles (say, the *left*) contains  $n$  disks of different sizes placed on each other, as shown in Fig. 8.2. The objective of the problem is to transfer all the disks from the left pole to right pole such that only one disk can be moved at a time (to any pole) and a larger disk cannot be placed on top of a smaller disk.



**Fig. 8.2 Tower of Hanoi**

This problem can easily solved for small values of  $n$ . However, the level of difficulty increases rapidly as the number of disks increases. It is also very difficult to write a program for this problem using loops and other control structures. However, the use of recursion leads to a very simple and elegant solution.

The first step in implementing recursion is to find out how we can reduce the size of the problem under consideration. The problem of moving  $n$  disks, can be decomposed into smaller problems of moving  $n - 1$  disks as shown below:

1. Move  $n - 1$  disks from the *source* pole to the *intermediate* pole

2. Move remaining disk from the *source* pole to the *target* pole
3. Move  $n - 1$  disks from the *intermediate* pole to the *target* pole.

The second step is to decide the terminating condition. Note that the problem of moving  $n - 1$  disks in steps 1 and 3 will actually be solved in terms of the smaller problem of moving  $n - 2$  disks, which in turn will be solved in terms of the still smaller problems involving  $n - 3$  disks and so on for as long as there is at least one disk to be moved. Thus, the terminating condition for this problem is that the number of disks to be moved should be equal to zero.

The complete program that includes a recursive function `hanoi` is given below. The function accepts four parameters. The parameter `ndisk` of type `int` specifies the number of disks to be moved. The other three parameters, namely, `source`, `target` and `other`, are of type `char` and specify a single character identification of the source, target and intermediate poles, respectively.

```
/* Tower of Hanoi program */
#include <stdio.h>

void hanoi(int ndisk, char left, char right, char mid);
int main()
{
    int ndisk;

    printf("--- Tower of Hanoi ---\n");
    printf("Enter number of disks: ");
    scanf("%d", &ndisk);

    hanoi(ndisk, 'L', 'R', 'M');

    return 0;
}
/* Recursive function to move 'ndisk' disks from
'source' pole to 'target' pole using 'other' as intermediate pole */
void hanoi(int ndisk, char source, char target, char other)
{
    if (ndisk > 0) {
        hanoi(ndisk - 1, source, other, target);
        printf("Move disk from %c to %c\n", source, target);
        hanoi(ndisk - 1, other, target, source);
    }
}
```

The program output is given below.

```
--- Tower of Hanoi ---
```

```
Enter number of disks: 3
Move disk from L to R
Move disk from L to M
Move disk from R to M
Move disk from L to R
Move disk from M to L
Move disk from M to R
Move disk from L to R
```

## 8.6 Advanced Concepts

This section covers some advanced concepts related to functions. These include the `const` parameters and storage classes. A beginner may skip this section.

### 8.6.1 Function Parameter as a Loop Variable

We know that the C language uses the call by value mechanism to pass parameters and that a function parameter is a copy of the argument specified in the function call. Thus, a function parameter can be modified without affecting the corresponding argument in the calling function. This fact can often be used effectively to save some memory. Consider function `fact` given below to determine the factorial of an integer number.

```
long int fact(int n)
{
    int i;

    int fact = 1;
    for (i = n; i > 1; i--)
        fact *= i;

    return fact;
}
```

As parameter `n` is local to function `fact`, we can use it to control the loop instead of variable `i` as shown below, thus saving the memory used for variable `i`.

```
long int fact(int n)
{
    int fact = 1;

    for ( ; n > 1; n--)
        fact *= n;
```

```
    return fact;  
}
```

## 8.6.2 **const** Parameters

Sometimes we may want that a function should not modify the value of a parameter passed to it, either directly within that function or indirectly in some other function called from it. This can be achieved using **const** parameters. Consider, for example, the function given below to calculate the sum of the first  $n$  integer numbers.

```
/* sum of first n integer numbers (1 to n) */  
int sum_1_n(const int n)  
{  
    int i, sum;  
  
    n = 0;  
    for (i = 1; i <= n; i++)  
        sum += i;  
  
    return sum;  
}
```

You have probably noticed that the function is not correct as it modifies the value of  $n$  (probably by mistake) instead of variable **sum**, replacing the parameter value. The compiler does not give any error or warning in this situation making the problem difficult to identify. This problem can be avoided by declaring  $n$  as a **const** parameter as shown below (the function body is omitted to save space).

```
int sum_1_n(const int n)
```

Now the compiler reports an error whenever the value of  $n$  is modified in the function body. Note that we can pass either a **const** or a non-**const** variable as an argument where a **const** parameter is expected. However, the compiler gives a warning when we pass a **const** variable where a non-**const** parameter is expected.

Finally, note that this feature is particularly useful for passing arrays and while using the call by reference mechanism.

## 8.6.3 Storage Classes

Every variable has a type associated with it which decides the values that can be assigned to it and the operations that can be performed on it. In addition, we can specify a **storage class** for a variable which decides the following:

1. The type of storage to be used (either memory or CPU registers)
2. The **scope** or the part of the program from which that variable can be accessed
3. The **lifetime**, i. e., how long the variable will continue to live or exist
4. The default initial value, i. e., the value assigned to the variable if it is not explicitly initialized.

The C language provides four storage classes, namely, **automatic**, **register**, **static** and **external**. The keywords for these storage classes are **auto**, **register**, **static** and **extern**, respectively. The characteristics of the storage classes are summarized in Table 8.1.

**Table 8.1 Characteristics of storage classes**

Storage class	Storage	Scope	Lifetime	Default initial value
Register	CPU Registers	Local to block in which variable is defined	Till control remains within the block in which variable is defined	Garbage
Automatic	Memory		Till the end of program execution	0 for arithmetic types; NULL for pointers
Static				
External		Global		

A storage class is specified for a variable using the general format shown below.

*storage\_class data\_type var1, var2, ...;*

The order of the storage class and data type can be altered and the **int** data type can be omitted. Consider the example given below.

```
auto int a, b;
register int i, j;
static float p, q;
extern char x, y;
```

This example declares **a** and **b** as automatic variables of type **int**, **i** and **j** as register variables of type **int**, **p** and **q** as static variables of type **float** and **x** and **y** as external variables of type **char**.

Note that if we do not specify the storage class for a variable, the defaults are applicable. Thus, a variable defined within a block without any storage class is an automatic variable and a variable defined outside any function without any storage class is an external variable.

We can use any storage class while defining a variable within a block. However, for the variables defined outside any function, the storage class can be static or be omitted. The compiler will give an error if we use **auto** or **register** class for such variables defined outside any function.

## Automatic Storage Class

As mentioned earlier, if we do not specify a storage class while defining a variable inside a block, the **automatic storage class** is assumed. However, we can use the **auto** keyword to explicitly specify the automatic storage class for such variables.

As we know, **automatic variables** are not initialized to any specific value by default. Thus, if we do not initialize an automatic variable when it is defined, it will have a garbage value, i. e., a random, unpredictable value which is likely to change each time the program is executed.

Automatic variables are stored in memory. Their scope is local to the block in which they are defined, from the point of definition to the end of that block. Thus, once a variable is defined, it can be used in any subsequent part of that block including the blocks nested within it. However, it cannot be used outside that block.

An automatic variable is created every time control enters the block in which it is defined and it *lives* until control remains in that block. It is destroyed when control leaves that block.

Note that irrespective of the storage class used, the variable names must be unique within a block. Thus, we cannot redefine a variable within the same block with a different type and/or storage class. However, it is possible to reuse that variable name for other variables of any data type and/or storage class defined within other blocks including the contained blocks. When we access such a variable from within a block, the local variables (i. e., the variables defined in that block) are given preference. However, if the accessed variable is not defined in that block, variable access refers to the definition in the nearest outer block in which this block is contained. To understand this concept clearly, study the example given below.

```
int main()
{
    int x = 1;
    printf("%d ", x);           /* prints 1 */
    {
        printf("%d ", x);       /* prints 1 */
        {
            int x = 2;          /* variable name x reused */
            printf("%d ", x);   /* prints 2 */
        }
        printf("%d ", x);       /* prints 1 */
    }
    printf("%d ", x);           /* prints 1 */
    return 0;
}
```

The comments after the **printf** statements specify the values printed by these statements. The code is easy to understand and its output is given below.

Now consider another example given below.

```
void show_result(int marks)
{
    char result[10];

    if (marks >= 40) {
        char result[] = "Pass";
    }
    else {
        char result[] = "Fail";
    }
    printf("Result: %s", result);
}
```

In this function, the variable **result** is first declared as a **char** array. Then this variable is reused to define and initialize two **char** arrays (with values "Pass" and "Fail") within the **if** block and **else** block of an **if-else** statement. Depending on the outcome of the condition in the **if** statement, one of these arrays will be created. However, it will be destroyed as soon as control leaves that block. Thus, the **printf** statement will print the value of the **result** array defined in the beginning of this function. As this is an automatic array, its elements are not initialized with any specific value and thus have garbage values. When this function was called in Code::Blocks (with value of **marks** as 50), the output was displayed as shown below.

Result: X"■ b◀&w-[  
wh↑@

The reader is encouraged to correct this function.

### Register Storage Class

CPU **registers** are much faster than memory. Thus, if we store frequently used variables in CPU registers, the program will run faster. We can request the C compiler to store one or more variables in CPU registers by specifying the **register storage class** when these variables are defined. However, note that a CPU has a limited number of registers. Hence, it may not be possible for the compiler to honor all requests for allocation of CPU registers. If the compiler is not in a position to allocate a CPU register for a requested variable, that variable is stored in memory and is treated as an automatic variable.

Apart from the storage location, register variables are similar to automatic variables. Thus, they have scope local to the block in which they are defined, they are destroyed when control leaves that block and the compiler does not initialize them with any specific values (they have garbage values), if we do not explicitly initialize them.

The register storage class should be used for frequently used variables, e. g., control variables in loops. Consider the example given below.

```
int sum(int n)
{
    register int i, sum;
    sum = 0;
    for(i = 1; i <= n; i++)
        sum += i;

    return sum;
}
```

This code segment defines variables `i` as well as `sum` as register variables in an attempt to speed up program execution.

Note that whether a variable will be stored in a CPU register or not depends, besides the availability of a CPU register, on the possibility of accommodating that variable in the CPU register. As modern CPUs have 64 bit registers, it is possible to allocate CPU registers for variables of basic data types (`char`, `int`, `float` and `double`) as well as pointer variables. Remember that we should not request a register storage class for bigger data objects such as arrays or large structures. However, if we do so, the compiler will ignore the request and treat those variables as automatic variables, as mentioned earlier.

### Static Storage Class

Like automatic variables, **static variables** are also stored in memory and have local scope. However, they differ in two respects, lifetime and default initial values.

Static variables come into existence when the execution of the block in which they are defined begins for the first time and they are not destroyed when control leaves that block. Thus, they *live* till the end of program execution. Note that the values assigned to these variables are available the next time that block is executed. Thus, static variables can be used to remember data values in a function across function calls.

If we do not explicitly initialize a static variable when it is defined, the compiler initializes it with the default initial value, which is zero for arithmetic types and NULL for pointers. This initialization is done only once, the first time control reaches that block. Initialization is not done in subsequent executions of that block.

To further understand the concept of static variables, consider the program given below.

```
#include <stdio.h>
void func(void)

int main()
```

```

{
    func();
    func();
    func();

    return 0;
}

void func(void)
{
    static int x = 100;
    printf("%d ", x++);
}

```

In this example, the function **func** is called three times from the **main** function. This function defines a static variable **x** with initial value 100. This value is printed and then incremented in a **printf** statement.

The first time the function **func** is called, variable **x** is initialized to 100. The **printf** statement prints this value and increments it to 101. When control leaves function **func**, **x** is not destroyed as it is a static variable and its value is preserved for use in subsequent calls to the function.

When the function **func** is called a second time, **x** is not initialized again. Thus, the **printf** statement prints the value of **x** (101) and increments it to 102. Finally, the third call to function **func** prints the value 102 (and increments it to 103). The output of the program is given below.

**101 102 103**

What would program output be if we declare variable **x** in function **func** as shown below?

**static int x;**

### External Storage Class

External variables, as the name indicates, are defined outside any function. These variables are similar to static variables in terms of storage location, lifetime and default initial values.

External variables are stored in memory. They come to existence when program execution begins and live till the end of program execution. Also, the compiler initializes them with default initial values (0 for arithmetic types and **NULL** for pointers), if they are not explicitly initialized.

However, unlike all other variables (automatic, register and static which have local scope), external variables have global scope. Hence, these variables are also called global variables. They can be accessed in all parts of the program including all the functions in the same program file as well as other program files in a multi-file C program. Thus, they provide an easy mechanism to share data between different functions without explicitly passing them around. This is particularly useful if a large number of

variables are shared by multiple functions. It is a convention to define the external variables at the beginning of a program file, as illustrated below.

```
#include <stdio.h>

int x = 10;
int y;
void func(void);

int main()
{
    printf("%d %d ", x, y);
    x = 100, y = 200;
    func();
    printf("%d %d ", x, y);
    return 0;
}

void func(void)
{
    printf("%d %d ", x, y); /* global x and y */
    {
        int x = 5, y = 2;
        printf("%d %d ", x, y); /* local x and y */
    }
    x = 50, y = 60;
}
```

In this program, the variables `x` and `y` are external variables initialized to 10 and 0 (default value), respectively. These variables are global and are accessible in both the functions, `main` and `func`. The `main` function first prints their values and assigns new values to them (100 and 200). The `func` function is then called which first prints these values. A block in the `func` function reuses these variable names to define local variables `x` and `y` with values 5 and 2, respectively, and prints them. As local variables get preference over global variables, the values are printed as 5 and 2. The `func` function then assigns new values (50 and 60) to variables `x` and `y` (global variables) outside the block. Finally, control returns to the `main` function where the values of `x` and `y` (global variables) are printed again as 50 and 60. The program output is given below.

10 0 100 200 5 2 50 60

You have probably noticed that the `extern` keyword was not used in the previous program. This keyword *declares* that a variable is external, i. e., its definition is provided in some other part of the program. By default, an external variable is accessible from the point of its definition till the end of that

file. However, if we wish to access an external variable in a file before it is declared, we need to use the `extern` declaration, as illustrated below.

```
#include <stdio.h>
void func(void);

int main()
{
    extern int x; /* declaration */
    printf("%d ", x);
    x = 20;
    func();
    printf("%d ", x);

    return 0;
}

int x = 10; /* definition */
void func(void)
{
    printf("%d ", x);
    x = 30;
}
```

In this example, variable `x` is *defined* as an external variable with value 10 after the `main` function. Thus, it is not accessible in the `main` function and an `extern` declaration is required to enable this. However, `x` is accessible from function `func` as it is defined before that function. Thus, explicit declaration of variable `x` is not required in function `func`.

When this program is executed, the external variable `x` is first initialized to 10. The `main` function prints its value and assigns a new value (20) to it. Then it calls function `func` where this value of `x` is printed; new value (30) is then assigned. Control then returns to the `main` function where this value of `x` is printed again. Thus, the output of the program is

10 20 30

Note that the `extern` declaration of a variable should match with its definition with regard to data type, type modifier and storage class; otherwise, the compiler will report an error. The `extern` declaration does not create variable and it can be specified any number of times, even within the same scope.

Finally note that the C language allows a program to be split into multiple program files (.c files) and header files (.h files). To access an external variable defined in another file, we need to declare that variable using an `extern` declaration in files where that variable is to be accessed. Consider the

program given below.

main.c file:

```
#include <stdio.h>

int x = 10;
int y = 20;

int main()
{
    func1();
    func2();
    printf("%d %d ", x, y);
    return 0;
}
```

func.c file:

```
#include <stdio.h>
extern int x;

void func1(void)
{
    printf("%d ", x);
    x = 100;
}
void func2(void)
{
    extern int x, y;
    printf("%d %d ", x, y);
    x = 200, y = 300;
}
```

The **main.c** file defines two external variables, **x** and **y**, with initial values 10 and 20, respectively. The **main** function in this file calls two functions, **func1** and **func2**, which are defined in another file, **func.c** and then prints the values of **x** and **y**.

The **func.c** file declares **x** as an external variable at the beginning. Thus, **x** in this file refers to variable **x** defined in the **main.c** file. The function **func1** prints the value of **x** (10) and assigns it a new value (100). The function **func2** declares **x** and **y** as external variables. Thus, these variables refer to variables **x** and **y** defined in the **main.c** file. The function **func2** then prints the values of **x** (100)

and **y** (20) and then assigns them new values (200 and 300, respectively). These modified values are printed by the **printf** statement in the **main** function. The program output is given below.

```
10 100 20 200 300
```

Note that variable **x** is declared twice in the **func.c** file. An external variable can be declared any number of times. However, it can be defined only once.

## Exercises

1. Answer the following questions:
  - a. What is the default parameter passing method in the C language?
  - b. What is another name for a function declaration?
  - c. State the name for the following: expression written in a function call.
  - d. Name the technique in which a function calls itself.
  - e. Which statement is used to send back the values to a calling function
2. Define the functions for the following operations on an integer number:
  - a. Reverse the digits of a given number.
  - b. Determine the sum of digits of a given number until a single digit sum is obtained.
  - c. Test whether a given number is prime or not.
  - d. Test whether a given number is an Armstrong number or not, i. e., the sum of cubes of the digits is equal to the number itself.
3. Define functions for the following operations on characters.
  - a. Test whether a given character is a vowel or not.
  - b. Test whether a given character is a multiplicative operator or not.
  - c. Convert a character to uppercase.
  - d. Determine the number of consonant characters between two given letters
4. Define functions for following operations:
  - a. Determine the number of days in a given month. Return zero if the given month is invalid.
  - b. Calculate the magnitude of a complex number.
  - c. Calculate simple interest on deposit.
  - d. Determine the median of three given numbers.
  - e. Calculate  $x^i$  where  $x$  is a floating-point number and  $i$  is an integer.

5. Identify the errors in the following function definitions:

- a. /\* min of three numbers \*/

```
float min3(float a, b, c)
{
    float (min = b > c ? c : b);
    if (min > a) {
        min = a;
        return min;
    }
}
```
- b. /\* smaller of two numbers \*/

```
int small(int a, int b)
{
    int small = a;
    if (b < small)
        small = b;
}
```
- c. /\* return cube root of a number \*/

```
double cube_root(double x)
{
    return sqrt(sqrt(x));
}
```
- d. /\* return cube of a number \*/

```
double CUBE(double x)
    return x * x * x;
```
- e. /\* return square of a number \*/

```
int sqr(double x)
(
    sq = x * x;
    return sq;
)
```
- f. /\* sum of two numbers \*/

```
float sum(int x, int y, int z)
{
    float z = a + b;
    return z;
}
```
- g. /\* test if a number is divisible
 by another number \*/

```
int is_divisible(a, b);
```

- {
  - if (a % b == 0)
    - return 0;
    - else return 1;
  - }
- h. int is\_leap(int y); /\* decl \*/
  - int is\_leap(int) /\* definition \*/
  - {
    - if (y % 4 == 0)
      - return 1;
      - else return 0;
    - }
- i. /\* magnitude of complex number \*/
  - float mag(float re, float im)
  - {
    - mag = sqrt(re \* re + im \* im);
    - return mag;
  - }
- j. /\* test for uppercase letter \*/
  - void isupper(char c)
  - {
    - return c >= A && c <= Z;
  - }
- k. /\* exchange two numbers \*/
  - void exchange(int x, y)
  - {
    - temp = x;
    - x = y;
    - y = temp;
  - }
- l. /\* factorial using recursion \*/
  - long fact(n)
  - {
    - if (n > 0)
      - fact = n \* fact(n - 1);
    - else fact = 0;
    - return fact;
  - }
- 5. Determine the output of the following programs. The #includes and function prototypes are omitted to save space:

a.

```
int main()
{
    int n = 5;
    do {
        func(n--);
    } while(n > 0);
    return 0;
}
void func(int a)
{
    for(i = 0; i < a; i++)
        printf("*");
    printf("\n");
}
```

b.

```
int main()
{
    int x = 1234;
    rev(x);
    return 0;
}
void rev(int x)
{
    if (x > 0) {
        printf("%d", x % 10);
        rev(x / 10);
    }
}

```

c.

```
int main()
{
    f(); f(); f();
    return 0;
}
void f() {puts("Hello");}
```

d.

```
int main()
{
    printf("%d", f(1, 2, 3));
    return 0;
}
int f(int a, int b, int c) {
    return a + b * c;
}
```

```

e. int main()
{
    int x = 10;
    increment(x);
    printf("%d ", x);
    return 0;
}
void increment(int x)
{
    if (x > 0)
        x++;
}

f. int main()
{
    float x = 1.5;
    f(x);
    printf("%f", x);
    return 0;
}
void f(int x)
{
    printf("%d ", x);
}

```

## **Exercises (Advanced Concepts)**

1. Answer the following questions.
  - a. How do you prevent a function parameter from being modified in the function body?
  - b. Which storage classes do not initialize variables to default values?
  - c. Which storage class provides global scope to variables?
  - d. What is the use of a static variable?
  - e. Which storage class can be used to speed up program execution?
2. Select the correct option for the following questions:
  - a. Which of the following is a keyword for a storage class in the C language?
    - i. **resister**
    - ii. **register**
    - iii. **register**

- iv. None of these
  - b. Which of the following types of variables are initialized with default values if they are not explicitly initialized by a programmer?
    - i. Automatic and register
    - ii. Static and register
    - iii. External and register
    - iv. Static and external
  - c. What is the default storage class for a variable defined outside any function?
    - i. Automatic
    - ii. Static
    - iii. Register
    - iv. External
  - d. Which of the following correctly describes a static variable of type int?
    - i. Storage: memory, Scope: global, Initial value: zero
    - ii. Storage: memory, Scope: local, Initial value: garbage
    - iii. Storage: memory, Scope: local, Initial value: zero
    - iv. None of these
  - e. Which of the following statements regarding storage classes is correct?
    - i. A storage class need not be specified. If omitted, an `int` data type is assumed.
    - ii. Every variable has a storage type associated with it.
    - iii. A storage class specifies the amount of memory a program should use for its execution.
    - iv. None of these.
3. Determine the output of the following programs or functions (the `#includes` and function prototypes are omitted to save space):
- a. 

```
int x;
int y = 5;
int main()
{
    int i;
    while(x < y)
        show_xy(++x, y--);
}
void show_xy(int a, int b)
{
```

```

        printf("%d %d\n", ++a, b--);
    }

b. int x;
int main()
{
    f();
    printf("%d ", x);
    f();
}
void f()
{
    static x = 5;
    printf("%d ", x++);
}

c. int main()
{
    extern int x;
    f(x);
    printf("%d", x);
}

int x;
void f(int x)
{
    printf("%d ", ++x);
}

d. int main()
{
    auto x = 5;
    for(i = 0; i < 3; i++)
        f(--x);
    printf("%d", x);
}
void f(int x)
{
    static y;
    x--, ++y;
    printf("%d %d ", x, y);
}

```

4. Identify errors, if any, in the following program segments:

a. void (const int a)  
{

```
    printf("%d ", ++a);
}

b. void f()
{
    extern a;
    printf("%d", a);
}
int a;

c. void f()
{
    extern int a;
    printf("%d", a);
}

d. void f()
{
    extern static int a;
    printf("%d", a);
}
int a;
```

# 9 Vectors or One-dimensional Arrays

An array is a collection of elements of the same data type. The C language provides one-dimensional arrays or vectors, two-dimensional arrays or matrices as well as multidimensional arrays.

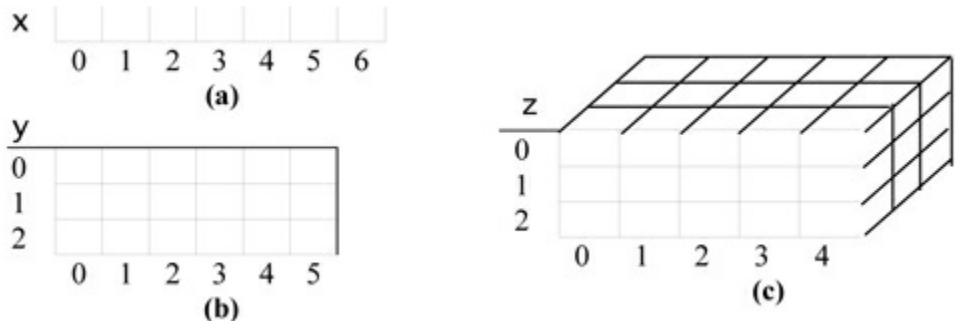
This chapter presents one-dimensional arrays. It covers basic operations on arrays (declaration, element access, initialization, etc.) and how they can be manipulated using loops. Next a discussion on how arrays can be used as function parameters is presented. Numerous examples of array manipulation are provided throughout this chapter. The *Advanced Concepts* section covers some aspects related to arrays. A beginner may skip this section in the first reading.

## 9.1 Introduction

The programs presented thus far in this book use *scalar* variables, each of which can store a single value. Thus, for each entity in a program, we have used a distinct variable. However, this approach is not suitable for processing a collection of related data items, particularly when the collection contains a large number of elements. For example, the marks of a student in six subjects, the names of 50 students in a class, the salary data of 250 employees of an organization, etc.

The C language provides arrays to store and process such collections of data. An **array** is a collection of data items of the same type. We can declare arrays of built-in data types such as `char`, `int`, `float`, `long int`, etc. as well as user-defined data types. Further, we can declare one-dimensional, two-dimensional and multidimensional arrays. However, one-dimensional arrays (often called **vectors**) and two-dimensional arrays (often called **matrices**) are required more often.

Fig. 9.1 shows one-, two- and three-dimensional arrays. Each data item in an array is called an **element**. The individual elements of an array are numbered starting from 0 (and not from 1). All the elements of an array are stored in consecutive memory locations. Although we can access the individual elements of an array, more often an operation on an array involves performing the same (or similar) operation on every element in it. This is usually done using nested `for` loops.



**Fig. 9.1** Types of arrays (a) one-dimensional (b) two-dimensional and (c) three-dimensional

## 9.2 One-dimensional Arrays

### 9.2.1 Array Declaration

Every array is given a name. The rules for naming arrays are the same as those used for variable names. As with scalar variables, an array must be declared before it is used. The **declaration** of a one-dimensional array takes the following form:

```
arr_type arr_name [ arr_size ] ;
```

where *arr\_name* is the name of array being declared, each element of which is of type *arr\_type*. The expression *arr\_size* enclosed in square brackets is an integral constant or a constant integral expression that specifies the array size, i. e., the maximum number of elements in the array.

We can declare arrays of any built-in type such as `char`, `float`, `unsigned int`, etc. In addition, we can declare arrays of user-defined types such as structures, enumerated data types, etc. Note that we can declare one or more arrays in a single declaration statement. We can also mix scalar variables and arrays in a declaration statement.

When a compiler comes across the declaration of an array, it reserves a block of memory for the specified number of array elements. For a one-dimensional array containing *n* elements, the memory required, i. e., the array size is given as  $n \times \text{sizeof}(\text{arr\_type})$  bytes.

Also note that when an array is declared in a block as shown above, the array elements are not initialized to any specific value and instead contain garbage values. This behaviour is similar to that of scalar variables.

#### Example 9.1 Array declaration

Several examples of array declaration are given below.

a) `int x[7];`

This statement declares a one-dimensional array named `x` having 7 elements, each of type `int` as shown in Fig. 9.1a.

b) `float num[100], min, max;`

This statement declares three entities of type `float`: an array `num` having 100 elements and scalar variables `min` and `max`.

c) `#define MAX_SUB 6  
int marks[MAX_SUB];`

This example first defines a symbolic constant `MAX_SUB` with value 6 followed by a one-dimensional array named `marks` of type `int` and size 6. This array can be used to store the marks of a student in six subjects.

## 9.2.2 Accessing Array Elements

The elements of an array are numbered starting from 0 (and not from 1) as illustrated in Fig. 9.1. Thus for a one-dimensional array having `arr_size` elements, the element subscripts or indexes are in the range 0 to `arr_size - 1`.

To access an individual elements of an array, C provides the **array subscript operator** [ ]. An array element can be accessed by writing the array name followed by a subscript expression within the array subscript operator as shown below.

`arr_name [ expr ]`

where `expr` is an integral expressions that specifies the position or *index* of an element in the array. Thus, the elements of an array `a` having 10 elements are referred as `a[0], a[1], ..., a[9]`. Also, the *i*th element is referred as `a[i]`.

For element access to be valid, the value of expression `expr` should be in the range 0 to `arr_size - 1`. Remember that C does not provide any mechanism to verify that the array element being accessed actually exists. Thus, we can declare an array `arr` having 10 elements and access a non-existent element at location 10 as `arr[10]` or even at location 1000 as `arr[1000]`. This is a very common source of errors. It is the programmer's responsibility to ensure that a program does not access non-existent array elements.

## 9.2.3 Operations on Array Elements

For an array of type `arr_type`, we can perform a number of operations on its elements like those we perform on a simple variable of that type. Thus, we can use an array element in an expression (arithmetic, relational, logical, etc.), assign a value to it, perform increment and decrement operations using the `++` and `--` operators, pass it as an argument to a function and so on. Note that the array subscript operator (along with some other operators) has highest **precedence** amongst all the operators. Thus, when we use an array element in an expression, we do not need to enclose it within parentheses.

To increment (decrement) a value of an array element, we can use the prefix or postfix increment (decrement) operator as shown below.

```
++ arr name [ expr ]      arr_name [ expr ] ++
-- arr name [ expr ]      arr_name [ expr ] --
```

We may often require the address of an array element, e. g., to read its value using the `scanf` function. This can be obtained it using the *address of* operator (`&`) as shown below.

```
&arr_name [ expr ]
```

### Example 9.2 Operations on array elements

Assume that array `x` has 10 elements, each of type `int` and `i`, `j` and `k` are variables of type `int`. The expressions given below illustrate several operations on array elements.

<code>x[0] + 10</code>	<code>x[2] &gt;= 1</code>	<code>x[0] &gt; 0    x[1] &gt; 0</code>
<code>2 * x[1] + 5</code>	<code>x[i] &gt; x[j]</code>	<code>x[i] &lt; x[j] &amp;&amp; x[j] &lt; x[k]</code>
<code>++x[i]</code>	<code>5 &gt;= x[i-j]</code>	<code>!(x[i] &lt; 0    x[j] &gt; 50)</code>

We can assign the value of an expression of appropriate type to an array element as illustrated in the assignment statements given below.

```
x[i] = i;
x[i] = x[i-2] + x[i-1];
```

We can perform console I/O on an array element using appropriate functions such as `scanf`, `printf`, `getchar`, `putchar` etc:

```
scanf("%d %d %d", &x[0], &x[1], &x[2]);
scanf("%d", &x[i]);
printf("x[%d] = %d\n", k, x[k]);
```

We can also pass an array element or an expression containing array elements as an argument to a function, as in

```
a = sqrt(x[0]);
b = log(x[i] + 2 * x[j]);
```

and return an array element from a function, as in

```
return x[0];
```

#### 9.2.4 Operations on Entire Arrays

Except initialization of arrays, which is discussed in the next section, the C language does not provide any statement or operator to perform operations on entire arrays. Basic operations such as array copy, console I/O, etc. are also not available directly. We can use a loop to perform operations on an entire array. As the number of elements to be processed from an array is often known, the **for** loop is an obvious choice.

The elements of a one-dimensional array arr of size  $n$  can be processed using a **for** loop as shown below.

```
for (i = 0; i < n; i++) {  
    /* process element arr[i] */  
}
```

Although **for** loops are more suitable for operations on arrays, the **while** and **do ... while** loops can also be used. Note that for operations on a string, which is a one-dimensional array of characters terminated by a null character, the **while** loop is more suitable.

### Example 9.3 Commonly used operations on one-dimensional arrays or vectors

This example illustrates some commonly used operations for one-dimensional arrays, namely, array initialization and console I/O. Assume that the arrays are declared as follows:

```
int num[10];  
float x[20];
```

#### a) Initialize a vector

```
for (i = 0; i < 10; i++)  
    num[i] = 0;
```

This **for** loop initializes each element of the array **num** to zero. As **num** contains 10 elements (**num[0]** to **num[9]**), the loop variable **i** takes values from 0 to 9. Thus, the values **i** correspond to the indexes of array elements. For each value of **i**, **num[i]**, i.e., the **i**th element of **num** is initialized to 0.

```
for (i = 0; i < 10; i++)  
    num[i] = i + 1;
```

This example is similar to the above example except that the **i**th element of array **num** is now initialized with the expression **i + 1**. Thus, the **for** loop initializes the elements of **num** to values 1, 2, 10. Alternatively, we can use the **for** loop given below to achieve the same effect:

```
for (i = 1; i <= 10; i++)  
    num[i - 1] = i;
```

**b) Print a vector on the screen**

```
for (k = 0; k < 10; k++)
    printf("%d ", num[k]);
```

This **for** loop prints all the elements of array **num** on the screen. Each array element is printed using the format specifier "%d ". Thus, each element is followed by a single space character. This causes the array elements to be printed on the same line. When an output line is filled, the cursor moves to the next line. Note that if we wish to print each array element on a line by itself, we should modify the format string as "%d\n".

**c) Read a vector from the keyboard**

```
for (j = 0; j<15; j++)
    scanf("%f", &x[j]);
```

This **for** loop reads 15 real numbers from the keyboard and stores them in array **x**. Note that the loop variable **j** takes values from 0 to 14 and the **scanf** function reads the value of the *j*th array element, i.e., **x[j]**. Thus, the numbers entered from the keyboard are stored in array elements **x[0]**, **x[1]**, ..., **x[14]** in that order. Note that the remaining array elements (**x[15]** to **x[19]**) are unused in this case and may have garbage values.

Data entry is facilitated by providing appropriate message prompts to the user, as illustrated below.

```
printf("Enter elements of array x:\n");
for (j = 0; j<15; j++) {
    printf(" x[%d] = ", j);
    scanf("%f", &x[j]);
}
```

This program segment first prints the message "Enter elements of array x:" and then for each element in the array, it displays a prompt indicating the element to be entered. A partial output of this program segment is shown below.

```
Enter elements of array x:
x[0] = 23
x[1] = 15
x[2] = 35
...
```

**d) Read a vector of desired size from the keyboard**

A more general program segment that allows the user to input a vector of desired size *n* is given below.

```

printf("Enter array size (max 20): ");
scanf("%d", &n);

printf("Enter array elements: \n");
for (j = 0; j < n; j++)
    scanf("%f", &x[j]);

```

Note that the array size should not exceed the dimension specified in the vector declaration. Thus, the value of *n* in the above example should not exceed 20. We can use a **do ... while** loop to ensure that the user enters the correct array size before accepting the array elements as shown below:

```

do {
    printf("Enter array size (max 20): ");
    scanf("%d", &n);
} while (n < 1 || n > 20);

```

### Program 9.1 Print a list of numbers in reverse order

Write a program to read several integer numbers and print them in reverse order.

**Solution:** In this program, an array **num** of type **int** is used to store the given integer numbers and variable **nelem** to store their count. First, the value of **nelem** is read. Then a **for** loop is used to read the given numbers and store them in array **num**. Finally, another **for** loop is used to print the given numbers in reverse order. The program is given below.

```

/* Print a list of numbers in reverse order */
#include <stdio.h>
int main()
{
    int num[50];      /* array with 50 elements */
    int nelem, i;     /* nelem: number of elements in array num */
    /* read array */
    printf("Enter number of array elements: ");
    scanf("%d", &nelem);
    printf("Enter array elements:\n");
    for (i = 0; i < nelem; i++)
        scanf("%d", &num[i]);

    /* print array elements in reverse order */
    printf("Array elements in reverse order:\n");
    for (i = nelem - 1; i >= 0; i--)
        printf("%d ", num[i]);
    printf("\n");
    return 0;
}

```

}

Observe that in the second **for** loop, the array elements are accessed as `num[i]` and index variable `i` assumes the values `nelem - 1, nelem - 2, ..., 0`. Thus, the array elements are printed in reverse order. Alternatively, we can print the array elements in reverse order using the **for** loop given below in which `i` assumes the values `0, 1, ..., nelem-1` and the array elements are referred as `num[nelem - 1 - i]`.

```
printf("Array elements in reverse order:\n");
for (i = 0; i < nelem; i++)
    printf("%d ", num[nelem-1-i]);
printf("\n");
```

#### Example 9.4 More operations on one-dimensional arrays or vectors

In this example, additional operations on one-dimensional arrays are presented, including array copy, vector addition, sum and average of all elements, maximum and minimum of all elements and reversal of a vector. Assume that the arrays are declared as follows:

```
int a[20], b[20], c[20];
```

The program segments given below can be used for arrays of any length `n`. However, for the above declarations of arrays, `n` should be less than or equal to 20.

##### a) Array copy

Consider that we need to copy array `a` having `n` elements to another array `b`. For this, we must copy each element of array `a` to the corresponding element of array `b`, i. e.,  $b_i = a_i$ . This is achieved using a **for** loop as shown below.

```
for (i = 0; i < n; i++)
    b[i] = a[i];
```

Note that array `b` should have sufficient space to accommodate the elements being copied, i. e., its size should be at least `n`.

##### b) Addition of two vectors

Consider the addition of vectors `a` and `b` having `n` elements each. This addition yields a vector (say `c`) of size `n`. Each element of this result vector is obtained by adding the corresponding elements of the two vectors, i. e.,  $c_i = a_i + b_i$ . The code for vector addition is given below.

```
for (i = 0; i < n; i++)
```

```
c[i] = a[i] + b[i];
```

#### c) Sum and average of elements of a vector

The program segment given below calculates the sum and average of  $n$  elements from array **a**.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
avg = (float) sum / n;
```

The variables **sum** and **avg** are assumed to be of type **int** and **float**, respectively. Initially, the variable **sum** is initialized to zero. Then a **for** loop is used to add all the elements of array **a** to variable **sum**. Finally, the average (**avg**) is calculated after the loop is over. Note the use of typecast (**float**) to avoid integer division.

#### d) Maximum and minimum elements in a vector

To determine the maximum amongst  $n$  elements of array **a**, first assign element **a[0]** to variable **max** and then use a **for** loop to process the remaining  $n - 1$  elements. In each iteration of this loop, the value of **max** is updated if the  $i$ th array element is greater than **max**.

```
max = a[0];
for (i = 1; i < n; i++) {
    if (a[i]>max)
        max = a[i];
}
```

We can easily modify this code to determine the minimum value by replacing variable **max** with **min** and the  $>$  operator with the  $<$  operator. To determine both the maximum and the minimum from a given array, use two variables **max** and **min**, both initialized to **a[0]**. Then a **for** loop is set up to process the remaining  $n - 1$  array elements. The **if-else-if** statement within the for loop updates the values of **max** and **min** as shown below.

```
min = max = a[0];
for (i = 1; i < n; i++) {
    if (a[i]<min)
        min = a[i];
    else if (a[i] > max)
        max = a[i];
}
```

What if we want to determine the position of the maximum element rather than the element itself? We

can use an integer variable `max_pos` to store the index of the element having the maximum value. The program segment to determine the value of `max_pos` is given below.

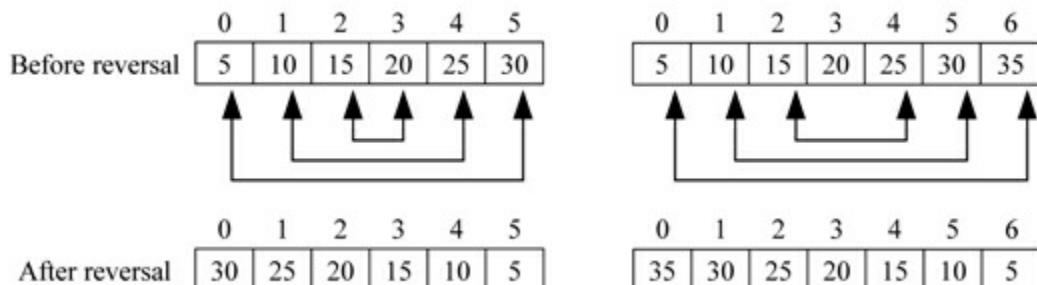
```
max_pos = 0;  
for (i = 1; i < n; i++) {  
    if (a[i]>a[max_pos])  
        max_pos = i;  
}
```

#### e) Reverse a vector

Now we can write the code to reverse the contents of a vector, e. g., if we are given a vector having six elements as 5, 10, 15, 20, 25 and 30, we should arrange the elements in the array in reverse order as 30, 25, 20, 15, 10 and 5.

To reverse a given vector, exchange the elements as illustrated in Fig. 9.2 for vectors containing 6 and 7 elements. Note that for a vector containing  $n$  elements, we need to perform  $[n / 2]$  exchanges. The `for` loop given below performs the desired reversal operation on  $n$  elements in vector `a`. It is set up to perform  $[n / 2]$  iterations with the values of index variable `i` from 0 to  $n/2$  (integer division). In the  $i$ th iteration, exchange the elements `a[i]` and `a[n-1-i]`.

```
for (i = 0; i < n/2; i++) {  
    int temp = a[i];  
    a[i] = a[n-1-i];  
    a[n-1-i] = temp;  
}
```



**Fig. 9.2** Reversing the elements of an array

The simplified code given below uses two index variables, `i` and `j`. The variables `i` assumes values 0 to  $[n/2]$  as before, whereas variable `j` assumes values  $n - 1, n - 2, \dots, n - 1 - [n / 2]$ .

```
j = n - 1;  
for (i = 0; i < n/2; i++) {
```

```

    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
    j--;
}

```

This code has been rewritten concisely below so that initialization and update of variable *j* are included in the loop control expressions using the comma operator.

```

for (i = 0, j = n - 1; i < n/2; i++, j--) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

```

### Program 9.2 Print the Fibonacci series (using arrays)

Write a program to print the first *n* terms of the Fibonacci series.

**Solution:** A program to print the first *n* terms of the Fibonacci series was presented in Program 6.3. Recall that the *i*th term in this series ( $F_i$ ) is expressed as a recurrence relation  $F_i = F_{i-1} + F_{i-2}$ . In this example, we need to use an array **fib** to store the terms of the Fibonacci series, starting with *i* = 0. Thus, the *i*th term of the series can be calculated directly using the recurrence relation as

```
fib[i] = fib[i-1] + fib[i-2];
```

In the program given below, array **fib** can store a maximum of 50 integer numbers. The program first accepts the value of *n*, the desired number of elements in the Fibonacci series. Then the first two terms in the series are initialized as

```
fib[0] = 0, fib[1] = 1;
```

and a **for** loop is used to calculate the remaining terms. Observe that the index variable **i** assumes the values from 2 to *n*-1. Finally, the series is printed using another **for** loop.

```

/* Print Fibonacci series -- using array */
#include <stdio.h>
int main()
{
    int fib[50];
    int n, i;

    printf("How many elements in Fibonacci series? ");
    scanf("%d", &n);

```

```

/* Generate Fibonacci series */
fib[0] = 0, fib[1] = 1;
for (i = 2; i < n; i++)
    fib[i] = fib[i-1] + fib[i-2];

/* print the series */
printf("Fibonacci series:\n");
for (i = 0; i < n; i++)
    printf("%d ", fib[i]);
return 0;
}

```

The program output is given below.

```

How many elements in Fibonacci series? 10
Fibonacci series:
0 1 1 2 3 5 8 13 21 34

```

Note that the second **for** loop used to print the Fibonacci series can be eliminated by printing the numbers as soon as they are generated. The modified code to generate and print the Fibonacci series is given below.

```

/* Generate and print Fibonacci series */
printf("Fibonacci series:\n");
fib[0] = fib[1] = 1;
printf("%d %d ", fib[0], fib[1]);

for (i = 2; i < n; i++) {
    fib[i] = fib[i-1] + fib[i-2];
    printf("%d ", fib[i]);
}

```

### Program 9.3 Vector addition

Write a program to add two vectors of size  $n$ .

**Solution:** For two vectors  $a$  and  $b$  having  $n$  elements each, the addition operation yields a vector (say  $c$ ) of size  $n$ . The  $i$ th element of the result vector is obtained by adding the corresponding vector elements, i. e.,  $c_i = a_i + b_i$ . The algorithm to perform the desired addition is given below.

Read  $n$ , the number of elements in given vectors

Read vector  $a$

Read vector  $b$

Perform vector addition  $c = a + b$

Print result vector  $c$

The program given below implements this algorithm in a straight-forward way using a separate **for** loop for the vector operation.

```
/* Vector addition */
#include <stdio.h>

int main()
{
    int a[10], b[10];          /* vectors to be added */
    int c[10];                /* result vector */
    int n, i;
    /* read vectors a and b */
    printf("Enter vector size: ");
    scanf("%d", &n);

    printf("Enter elements of vector a:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Enter elements of vector b:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &b[i]);

    /* perform vector addition */
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];

    /* print addition vector C */
    printf("Addition vector:\n");
    for (i = 0; i < n; i++)
        printf("%d ", c[i]);
    return 0;
}
```

The program output is given below.

```
Enter vector size: 5
Enter elements of vector a:
1 2 3 4 5
Enter elements of vector b:
```

2 3 4 5 6

Addition vector:

3 5 7 9 11

Note that we can combine the code for the last two steps in a single **for** loop as shown below.

```
/* add vectors a and b and print result */
for (i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
    printf("%d ", c[i]);
}
```

#### Program 9.4 Calculate the mean and standard deviation of a list of numbers

Write a program to calculate the mean and standard deviation of a given list of  $n$  numbers.

**Solution:** For a list of numbers  $x$  whose elements are referred to as  $x_0, x_1, \dots, x_{n-1}$  the mean  $\bar{x}$  and the standard deviation  $\sigma$  are defined as

$$\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} x_i \quad \text{and} \quad \sigma = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2}$$

The program given below calculates the values of  $\bar{x}$  and  $\sigma$ . It uses array  $x$  of type **float** to store the given numbers (maximum 50) and variable  $n$  of type **int** to represent the number of elements in array  $x$ . Also, the variables **sum**, **mean** and **sd** (all of type **float**) are used to represent the sum, mean and standard deviation, respectively, of the given numbers.

```
/* Calculate mean and standard deviation */
#include <stdio.h>
#include <math.h>

int main()
{
    float x[50]; /* max. 50 elements in array x */
    int n; /* number of elements in array x */
    float sum, mean, sd; /* sum, mean and standard deviation */
    int i;

    /* read data */
    printf("How many numbers (max 50)? ");
    scanf("%d", &n);
    printf("Enter numbers: ");
    for(i = 0; i < n; i++)
        sum += x[i];
    mean = sum / n;
    for(i = 0; i < n; i++)
        sd += (x[i] - mean) * (x[i] - mean);
    sd = sqrt(sd / n);
    printf("Mean: %f\nStandard Deviation: %f", mean, sd);
}
```

```

scanf("%f", &x[i]);

/* calculate mean */
sum = 0.0;
for(i = 0; i < n; i++)
    sum += x[i];
mean = sum / n;

/* calculate standard deviation */
sum = 0.0;
for(i = 0; i < n; i++)
    sum += (x[i] - mean) * (x[i] - mean);
sd = sqrt(sum / n);

printf("Mean = %6.3f\n", mean);
printf("Standard Deviation: %6.3f\n", sd);
return 0;
}

```

### Program 9.5 HSC mark list for a single student

Write a program to accept the marks of a student in the HSC examination and print the result (pass/fail) and if the student passes, print the total marks, percentage marks and class obtained.

**Solution:** In the HSC examination, there are six subjects each having a maximum of 100 marks. To pass the examination, a student has to score at least 35 marks in each subject. If a student passes, total marks and percentage marks are calculated and a class is awarded based on the percentage marks (`perce`) as follows:

$75 \leq \text{perce} \leq 100$	First class with Distinction
$60 \leq \text{perce} < 75$	First class
$50 \leq \text{perce} < 60$	Second class
$35 \leq \text{perce} < 50$	---

Let us use a one-dimensional integer array `marks` having six elements to store the marks obtained by a student in six subjects, variables `total` of type `int` and `perce` of type `float` to store total marks and percentage marks, respectively, and variable `result` of type `char` to store the examination result. The strings representing the result and class obtained by a student are not stored in any variables. Instead, they are printed directly on the screen. The algorithm for this program is given below.

```

Read marks in six subjects
Determine result ('P' or 'F')
if(result == 'P') {

```

```
Calculate total and perce  
Determine class  
Print total, perce, result and class  
}  
else print result
```

We can determine the result and total marks using the statements given below.

```
if (marks[0] >= 35 && marks[1] >= 35 &&  
    marks[2] >= 35 && marks[3] >= 35 && marks[4] >= 35 && marks[5] >=  
    result = 'P';  
else result = 'F';  
  
total = marks[0] + marks[1] + marks[2] + marks[3] + marks[4] + marks[5];
```

However, it is much more convenient to use the **for** loop to determine the result and total marks as shown in the program given below.

```
/* HSC marknlist program */  
#include <stdio.h>  
  
int main()  
{  
    int marks[6], total;      /* marks in 6 subjects and total marks */  
    float perce;             /* percentage marks */  
    char result;              /* char indicating result (Pass/Fail) */  
    int i;  
  
    /* accept marks */  
    printf("Enter marks in six subjects: ");  
    for (i = 0; i < 6; i++)  
        scanf("%d", &marks[i]);  
  
    /* determine result */  
    result = 'P'; /* assume student has passed */  
    for (i = 0; i < 6; i++) {  
        if (marks[i] < 35) {  
            result = 'F';  
            break;  
        }  
    }  
    if (result == 'P') {  
        /* Student has passed: calc total marks and percentage marks */  
    }
```

```

total = 0;
for (i = 0; i < 6; i++)
    total += marks[i];
perce = total / 6.0;

/* print result, total marks and percentage marks */
printf("Result: Pass\n");
printf("Total marks: %d\n", total);
printf("Percentage marks: %5.2f\n", perce);

/* determine and print class */
if (perce >= 75.0)
    printf("Class: First with distinction\n");
else if (perce >= 60.0)
    printf("Class: First\n");
else if (perce >= 50.0)
    printf("Class: Second\n");
else printf("Class: ---\n");
}

else printf("Result: Fail\n");

return 0;
}

```

The program is executed twice and the output is given below.

```

Enter marks in six subjects: 40 50 60 70 80 90
Result: Pass
Total marks: 390
Percentage marks: 65.00
Class: First

```

```

Enter marks in six subjects: 40 50 60 70 80 30
Result: Fail

```

### 9.2.5 Array Initialization

C allows **initialization** of an array by specifying an initialization list in the array declaration using the syntax given below.

*data\_type arr\_name [ arr\_size ] = { expr0, expr1, ..., expr\_n };*

where *arr\_name* is the name of the one-dimensional array having *arr\_size* elements, each of type

*data\_type* and the initialization list consists of zero or more constant expressions separated by commas and enclosed in curly braces. These expressions should be of the same type as that of the array; otherwise, their values are converted to type *data\_type* using standard conversions. However, if such conversion is not possible, the compiler will report an error.

This initialization statement causes the array elements to be initialized with the values of the specified constant expressions. The first array element (*arr\_name*[0]) is initialized with the value of the first expression (*expr0*), the second array element (*arr\_name*[1]) is initialized with the value of the second expression (*expr1*) and so on. If the initialization list has fewer elements than the size of the array being initialized, the remaining elements are initialized with the default value that is zero for arithmetic types such as **char**, **int**, **float**, etc. However, if the initialization list contains more elements than required, the compiler gives a *Too many initializers* error.

It is possible and often convenient to omit the array size as in

```
data_type arr_name [ ] = { expr0, expr1, ..., expr_n };
```

In this case, the array size is taken to be equal to the number of entries in the initialization list.

#### Example 9.5 Initialize one-dimensional arrays

Several examples of initialization of one-dimensional arrays are given below.

a) `int marks[6] = {57, 68, 98, 100, 85, 75};`

This example declares an array named **marks** having six elements of type **int** and initializes all the array elements with specified integer values (57, 68, 98, 100, 85 and 75) as shown below.

	0	1	2	3	4	5
marks	57	68	98	100	85	75

b) `float x[10] = {1.0, 1.25, 1.5, 1.75, 2.0};`

In this example, variable **x** is declared as an array having 10 elements each of type **float**. The first five elements (**x[0]** to **x[4]**) are initialized with the specified values (1.0, 1.25, 1.5, 1.75 and 2.0). The remaining array elements (**x[5]** to **x[9]**) are initialized to zero as the initialization list does not specify values for them. The contents of the array are shown below.

	0	1	2	3	4	5	6	7	8	9
x	1.0	1.25	1.5	1.75	2.0	0	0	0	0	0

c) `int mdays[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`

This example uses array **mdays** to store the maximum numbers of days in each month from January to December. The array is declared of type **int** and array size is not specified. As the months are numbered from 1 to 12, we need to use array elements **mdays[1]** to **mdays[12]** to store the desired

data. However, as the array elements are numbered from zero, we need to specify a dummy value (0 in above example) for element `mdays[0]`. The maximum number of days in month `mm` can thus be obtained as `mdays[mm]`. However, note that the maximum number of days in February is initialized as 28. We should adjust this value for a leap year. Finally, note that the size of the array is 13, the number of elements in the initialization list.

<code>mdays</code>	0	1	2	3	4	5	6	7	8	9	10	11	12
Month	—	Jan	Feb	Mar	Apr	May	June	July	Aug	Sept	Oct	Nov	Dec

d) `int letter_count[26] = {};`

This statement declares array `letter_count` having 26 elements each of type `int`. As the initialization list is empty, all the elements are initialized to zero. This array can be used to count the letters 'A' to 'Z' in the given text.

#### Program 9.6 Check the validity of given date

Write a program to check the validity of a given date.

**Solution:** An algorithm to determine the validity of a date, given in Program 7.2, is reproduced below for convenience.

```
valid = 0;
if (yy != 0) {
    if (mm ≥ 1 && mm ≤ 12) {
        determine max_days, the maximum number of days in month mm
        if (dd ≥ 1 && dd ≤ mdays)
            valid = 1;
    }
}
```

We can use `mdays` array in the previous example to determine the maximum number of days in any valid month `mm` as follows:

```
max_days = mdays[mm];
/* adjust max_days for February month in a leap year */
if (mm == 2 && (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0))
    max_days++;
```

Note that the value of `max_days` is first obtained as `mdays[mm]`. Then, if the month is February and the year `yy` is leap, we increment it by 1. We can now use this code to check the validity of a given date as shown below.

```

/* Date validity program using array */
#include <stdio.h>

int main()
{
    int dd, mm, yy;          /* given date */
    int valid;                /* flag to indicate date validity */
    int mdays[ ] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31};

    printf("Enter date (dd mm yyyy): ");
    scanf("%d %d %d", &dd, &mm, &yy);
    /* determine validity of given date */
    valid = 0;
    if (yy != 0) { /* check year */
        if (mm >= 1 && mm <= 12) { /* check month */
            /* determine maximum number of days in given month */
            int max_days = mdays[mm];
            if (mm == 2 && (yy % 4 == 0 && yy % 100 != 0 || yy % 400 == 0))
                max_days++;

            if (dd >= 1 && dd <= max_days) /* check date dd */
                valid = 1;
        }
    }

    /* print validity of date */
    if (valid)
        printf("Date is valid\n");
    else printf("Date is invalid\n");

    return 0;
}

```

The program is executed twice and the output is given below.

```

Enter date (dd mm yyyy): 29 2 2010
Date is invalid

```

```

Enter date (dd mm yyyy): 31 3 2010
Date is valid

```

Observe that this program is shorter and easier to understand than that given in Program 7.2. We can further simplify the evaluation of `max_days` using the conditional evaluation expression

```
max_days = mdays[mm] + (mm == 2 && (yy % 4 == 0 && yy % 100 != 0 ||  
                           yy%400 == 0)) ? 1 : 0;
```

Further, we can eliminate `max_days` by substituting the expression for `max_days` in the if statement that checks the value of `dd` as shown below.

```
if (dd >= 1 && /* check date dd */  
    dd <= mdays[mm] + (mm == 2 && (yy % 4 == 0 && yy % 100 != 0 ||  
                               yy % 400 == 0)) ? 1 : 0)  
    valid = 1;
```

Finally, note that we can write a single `if-else` statement to determine the validity of the given date as shown below:

```
if (yy != 0 && /* check year */  
    mm >= 1 && mm <= 12 && /* check month */  
    dd >= 1 && /* check date dd */  
    dd <= mdays[mm] + (mm == 2 && (yy % 4 == 0 && yy % 100 != 0 ||  
                               yy % 400 == 0)) ? 1 : 0)  
    valid = 1;  
else valid = 0;
```

Now we can rewrite the above program using functions. First, write a function `date_valid` to check the validity of given date. This function should accept a date in three integer parameters `d`, `m` and `y` and return 1 if the date is valid (if the year is non-zero, month is between 1 and 12 and the day is between 1 and the maximum number of days for that month) and 0 otherwise. Assuming that a function `month_days` is available that returns the maximum number of days in a given month, the function `date_valid` is written as follows:

```
/* determine validity of given date */  
int date_valid(int d, int m, int y)  
{  
    return y != 0 && /* check year */  
          m >= 1 && m <= 12 && /* check month */  
          d >= 1 && d <= month_days(m, y) /* check date dd */  
          ? 1 : 0;  
}
```

Now consider the implementation of function `month_days` given below that returns the maximum number of days in a given month.

```
/* determine maximum number of days in given month */  
int month_days(int m, int y)
```

```

{
    int mdays[] = {
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    return mdays[m] + (m == 2 && is_leap(y) ? 1 : 0);
}

```

This function uses the `is_leap` function that returns value 1 if the year is leap and 0 otherwise. A complete program for date validity check is given below (the definitions of the `date_valid` and `month_days` functions are omitted to save space). Observe that this program is much easier to read and understand compared to the earlier implementation.

```

/* Date validity check using functions */
#include <stdio.h>

int date_valid(int d, int m, int y);
int month_days(int m, int y);
int is_leap(int y);

int main()
{
    int dd, mm, yy; /* given date */
    printf("Enter date (dd mm yyyy): ");
    scanf("%d %d %d", &dd, &mm, &yy);

    if (date_valid(dd, mm, yy))
        printf("Date is valid\n");
    else printf("Date is invalid\n");

    return 0;
}

/* include here definitions of date_valid() and month_days() function

/* determine whether a given year is leap or not */
int is_leap(int y)
{
    return (y % 4 == 0 && y % 100 != 0 || y % 400 == 0) ? 1 : 0;
}

```

### Program 9.7 Increment a valid date

Write a program to accept a date and increment it.

**Solution:** Variables dd, mm and yy are used to represent a date, array mdays to store the maximum number of days in each month of the year and variable max\_days to store the maximum number of days in month mm. Assume that the given date is valid. To determine next date from a given valid date, we can use the following algorithm:

```
dd++;
if(dd > max_days) {
    dd = 1;
    mm++;
    if(mm > 12) {
        mm = 1;
        yy++;
    }
}
```

First increment the value of dd. If this value is greater than max\_days, the month mm is over. Hence, we initialize dd to 1 and increment the value of mm. Now if the value of mm is greater than 12, the year yy is over. Hence, initialize the month mm to 1 and increment the year yy.

We can further simplify this code by combining the increment and comparison operations. For this, we can use the prefix increment operators, as shown in the complete program given below.

```
/* Given a valid date, determine next date */
#include <stdio.h>

int main()
{
    int dd, mm, yy; /* given date */
    int max_days; /* max days in given month mm */
    int mdays[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31};

    printf("Enter date (dd mm yyyy): ");
    scanf("%d %d %d", &dd, &mm, &yy);

    /* determine max_days */
    max_days = mdays[mm];
    if (mm == 2 && (yy %4 == 0 && yy % 100 != 0 || yy % 400 == 0))
        max_days++;

    /* increment date */
    if (++dd > max_days) {
        dd = 1;
        if (++mm > 12) {
            mm = 1;
            ++yy;
        }
    }
}
```

```

    }
}

printf("Next date: %d/%d/%d\n", dd, mm, yy);

return 0;
}

```

The program is executed twice and the output is given below.

```

Enter date (dd mm yyyy): 28 2 2004
Next date: 29/2/2004

```

```

Enter date (dd mm yyyy): 31 12 2005
Next date: 1/1/2006

```

### 9.3 Arrays as Function Parameters

We can pass an array (one-, two- or multidimensional) as an argument to a function. The C language allows us to define functions that have one or more arrays as parameters. These parameters can be of different types and sizes. Moreover, we can mix scalar and array parameters. A function that uses a single one-dimensional array as a parameter can be defined as shown below.

```

ret_type func_name ( arr_type arr_name [ ] )
{
    ...
}

```

where parameter *arr\_name* is a one-dimensional array of type *arr\_type*. Note that the array size is not required in the above definition and is usually omitted. A call to this function takes the following form:

```
func_name ( arr )
```

where *arr* is a one-dimensional array. Although, we can pass arrays of different sizes to this function, the function body will process a fixed number of elements from these arrays.

We can generalize this function to process an array of any size by including another function parameter that specifies the number of elements to be processed from the array as:

```

ret_type func_name ( arr_type arr_name [ ], int n )
{
    ...
}

```

Such a function usually processes  $n$  elements in the beginning of the array (i. e., elements at positions 0 to  $n - 1$ ). If we wish to process an arbitrary range of elements, we can specify the range using two function parameters, say **first** and **last**.

Recall that *call by value* is the default mechanism for passing parameters in C language. When a function is called, the argument specified in the function call is copied to the corresponding function parameter specified in the function definition. The overhead of this copy operation would be substantial when an array of moderate to large size is being passed to a function. Hence, C language uses the *call by reference* mechanism to pass arrays efficiently. In this method, only the starting address of array argument (address of element 0) is passed to a called function.

The use of pass by reference makes an array parameter an input-output parameter. When the function modifies the elements of a parameter array, it actually modifies the elements of an argument array in the called function. Thus, the modified array is available in the calling function. This ability is extremely useful when we want to return the modified array to the calling function, e. g., read an array using a function, sort a given array, etc.

Finally, note that C language does not permit an array to be specified as a return type for a function.

#### Example 9.6 Passing one-dimensional arrays to a function

Several examples of passing one-dimensional arrays to a function are given below. These functions have been named *tvec\_op*, where *t* is one or more letters indicating the array type (*i* for **int**, *f* for **float**, etc.) and *op* specifies the operation to be performed. Thus, a function to print an array of **int** type will be named **ivec\_print**, a function to read a **float** array will be named **fvec\_read** and so on. This convention makes it easy to remember function names.

##### a) Function to print a vector on screen

```
void ivec_print(int x[])
{
    int i;
    for(i = 0; i < 5; i++)
        printf("%d ", x[i]);
    printf("\n");
}
```

This function accepts an array of type **int** and prints the first five elements in it. The return type of this function is **void** as it does not return any value. Note that this function is not useful while working with arrays of different sizes. Hence, we can generalize it to print an array of any size by including a size parameter (*n*) as shown below.

```
void ivec_print(int x[], int n)
{
```

```
int i;
for(i = 0; i < n; i++)
    printf("%d ", x[i]);
printf("\n");
}
```

The `main` function to print two arrays of size 5 and 8 is given below followed by its output.

```
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int b[] = {1, 2, 3, 4, 5, 6, 7, 8};

    ivec_print(a, 5);
    ivec_print(b, 8);

    return 0;
}
```

```
1 2 3 4 5
1 2 3 4 5 6 7 8
```

If we want to print a range of values in a given array, we can modify the function as follows:

```
void ivec_print(int x[], int first, int last)
{
    int i;

    for(i = first; i <= last; i++)
        printf("%d ", x[i]);
    printf("\n");
}
```

**b)** Function to read a vector from the keyboard

```
void ivec_read(int x[], int n)
{
    int i;

    for(i = 0; i < n; i++)
        scanf("%d", &x[i]);
}
```

This function uses a `for` loop to read the first `n` elements of a vector from the keyboard in array parameter `x`. These values are actually stored in the argument array specified in a call to this function and are thus available in the calling function.

c) Function to set elements of a vector to a given value

```
void ivec_set(int x[], int n, int val)
{
    int i;

    for(i = 0; i < n; i++)
        x[i] = val;
}
```

This function sets each of the first `n` elements of array `x` with a specified value `val`. The values assigned to array `x` are actually stored in the array specified in a call to this function.

d) Function to determine the maximum of vector elements

```
int ivec_max(int x[], int n)
{
    int i, max;

    max = x[0];
    for(i = 1; i < n; i++) {
        if (x[i] > max)
            max = x[i];
    }
    return max;
}
```

This function determines the maximum of the first `n` elements of vector `x` in variable `max` and returns it to the calling function.

e) Function to add two vectors

```
void ivec_add(int x[], int y[], int z[], int n)
{
    int i;

    for(i = 0; i < n; i++)
        z[i] = x[i] + y[i];
```

}

This function uses a **for** loop to add **n** elements of vectors **x** and **y** to obtain vector **z**. It has four parameters: vectors **x**, **y** and **z** of type **float** and vector size **n**. Each element of the addition vector **z** is obtained by adding the corresponding elements of vectors **X** and **y**, i. e.,  $z_i = x_i + y_i$ .

#### f) Function to insert an element in a vector

```
void ivec_insert(int x[], int n, int val, int pos)
{
    int i;

    /* shift right array elements x[n-1] to x[pos] */
    for (i = n-1; i >= pos; i--)
        x[i+1] = x[i];

    x[pos] = val;
}
```

This function inserts a specified value **val** at position **pos** in array **x** having **n** elements. For this, we have to first free the array position **pos** by shifting right array elements **x[n-1]** to **x[pos]** by one position starting with element **x[n-1]** as shown in Fig. 9.3. Now the value **val** can be written to **x[pos]**.

If an element is to be inserted at the end of the array (i. e., at position **n**), no element need to be shifted. On the other hand, if it is to be inserted in the array beginning (at position 0), all array elements must be shifted. Thus, on an average the insertion operation requires **n/2** shift operations which is quite costly, particularly when large arrays are involved.



**Fig. 9.3** Insert 7 at position 2 in a vector having 6 elements, i. e.,  $n = 6$

Note that, for an array of size **n**, we can insert an element at any positions from 0 to **n**. In this case, the array size should be increased by 1. The function given above does not check whether the insertion point is valid or not. Also, it does not increase the array size after the insertion is made. The user of this function should do the needful as illustrated in the code given below. We can however update the array size by using pass by reference mechanism (using a pointer) for parameter **n**.

```

int main()
{
    int a[10] = {1, 2, 3, 4, 5};          /* 5 positions available for ins
    ivec_print(a, 5);                  /* initially array has 5 element
    ivec_insert(a, 5, 0, 0);           /* insert 0 at position 0 */
    ivec_print(a, 6);                  /* now array has 6 elements */
    ivec_insert(a, 6, 7, 3);           /* insert 7 at position 3 */
    ivec_print(a, 7);                  /* now array has 7 elements */

    return 0;
}

```

The output of this code is given below.

```

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

```

### g) Function to delete an element from a vector

```

int ivec_delete(int x[], int n, int pos)
{
    int val, i;
    val = x[pos];      /* remember element being deleted */
    /* shift left array elements x[pos+1] to x[n-1] */
    for (i = pos+1; i < n; i++)
        x[i-1] = x[i];
    return val;         /* return deleted element */
}

```

This function deletes an element from the specified position **pos** from array **x** having **n** elements and returns the deleted element to the calling function. To delete an element, we must first store that element in variable **val** and then shift to the left all array elements starting from **x[pos+1]** to **x[n-1]** as shown in Fig. 9.4. Finally, the value **val** (i. e., the deleted element) is returned.



**Fig. 9.4** Delete element at position 2 from a vector having 6 elements ( $n = 6$ )

Note that no element needs to be shifted to delete the last array element. On the other hand, all array elements must be shifted to delete the first element. Thus, on an average, the deletion operation requires  $n/2$  element shifts, which is quite costly, particularly for large arrays.

Finally, note that the above function does not check whether the deletion position is valid or not. Also, it does not reduce the array size after an element is deleted. It is the user's responsibility to take necessary precautions. We can, however, update the array size by using the pass by reference mechanism (using a pointer) for parameter  $n$ .

#### Program 9.8 Calculate the mean and standard deviation of $n$ numbers (using functions)

Write a program to accept  $n$  numbers and calculate the mean and standard deviation.

**Solution:** A program to calculate the mean and standard deviation is given in Program 9.4. Now we rewrite it using functions. Besides the `main` function, we need the following functions in this program: `ivec_read` to read a list containing  $n$  elements and `ivec_mean` and `ivec_sd` to calculate the mean and standard deviation, respectively, of  $n$  elements in the given list. The complete program is given below.

```
/* Program to calculate mean and standard deviation using functions *
#include <stdio.h>
#include <math.h>

void ivec_read(float x[], int n);
float ivec_mean(float x[], int n);
float ivec_sd(float x[], int n);

int main()
{
    float num[50];
    int nelem;

    printf("Enter number of elements in list (max 50): ");
    scanf("%d", &nelem);

    printf("Enter elements in the list:\n");
    ivec_read(num, nelem);
    printf("Mean = %5.3f\n", ivec_mean(num, nelem));
    printf("Standard deviation = %5.3f\n", ivec_sd(num, nelem));

    return 0;
}

/* read a list of numbers */
```

```

void ivec_read(float x[], int n)
{
    int i;
    for(i = 0; i < n; i++)
        scanf("%f", &x[i]);
}

/* calculate mean of a given list of numbers */
float ivec_mean(float x[], int n)
{
    int i;
    float sum = 0.0;
    for (i = 0; i < n; i++)
        sum += x[i];

    return sum / n;
}

/* calculate standard deviation of a given list of numbers */
float ivec_sd(float x[], int n)
{
    int i;
    float mean = ivec_mean(x, n);
    float sum = 0.0;
    for (i = 0; i < n; i++)
        sum += (x[i] - mean) * (x[i] - mean);

    return (float) sqrt(sum / n);
}

```

## 9.4 Advanced Concepts

This section presents some advanced concepts about arrays. These include **const** array parameters, **static** arrays and external or global arrays. A beginner may skip this section in the first reading and return to it later.

### 9.4.1 **const** Vectors

If we desire that the elements of a vector should not be modified, we can declare that vector as a **const vector**. However, we must initialize this vector when it is declared, as it is not possible to modify it subsequently. Thus, a **const** vector can be declared and initialized as shown below.

```
const int mdays[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30,
                     31, 30, 31};
```

If we subsequently modify any element of this vector, the compiler gives an error. The compiler also gives a warning when this array is passed to a function that accepts a non-const vector.

Now we need to understand the use of **const** arrays as function parameters. As we know, the use of the pass by reference mechanism makes an array parameter an input-output parameter. Modifications to an array parameter actually causes the corresponding array argument in the calling function to be modified. This may not be desirable in some situations.

In several situations we do not want a function to modify the given array argument. This can of course be written as a comment at the beginning of the function. However, it does not guarantee that we (or some one else) will not modify the array inside the function. To avoid any inadvertent modification to the argument array, we should use a const array. For example, the **ivec\_print** function (Example 9.6a) used to print the elements of a given array does not and inadvertently should not modify the given array. Thus, we can use a **const** array as shown below (function body is omitted to save space):

```
void ivec_print(const int x[], int n) { ... }
```

Now if the function inadvertently modifies one or more array elements (directly or through some function call), the compiler will be able to catch the error. This feature is very useful and prevents the occurrence of several hard to trace bugs.

Note that as in case of a scalar variable, we can pass a const or non-const array as an argument where a const array parameter is expected. However, the compiler gives a warning when we pass a const array where a non-const array parameter is expected.

Finally note that while using a non-**const** array parameter, if we wish to modify the elements of the parameter array within a function without affecting the argument array, we will need to use a local copy of the array in the called function.

## 9.4.2 Static Arrays

In the last chapter, we studied **static** variables. This feature can be used with arrays as well. A static array has the following characteristics:

1. It has a local scope. Thus, it can be used only within the block in which it is defined.
2. It is initialized only once, the first time the control passes through its declaration.
3. If a static array is not explicitly initialized, its elements are initialized with the default value which is zero for arithmetic types (**int**, **float**, **char**) and **NULL** for pointers.
4. A static array has a lifetime till the end of program execution. Thus, a static array defined within a

function is not destroyed when control leaves that function and the value of this array is available the next time the function is called.

For example, in Program 9.6, we used the `month_days` function to determine the maximum number of days in a given month. This function uses an array `mdays` to store the number of days in each month. Note that this function is quite inefficient as `mdays` array will be initialized each time the function is called. We can overcome this problem by declaring `mdays` as a static array, as shown below.

```
/* determine maximum number of days in given month */
int month_days(int m, int y)
{
    static int mdays[] = {
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };

    return mdays[m] + (m == 2 && is_leap(y) ? 1 : 0);
}
```

Now `mdays` array is initialized only once, the first time this function is called. This array is preserved for use in subsequent executions of this function. Thus, program execution will speed up now, of course at the cost of memory used by this array. Further, we can declare this array as a `const` array to avoid any inadvertent changes being made to it.

```
static const int mdays[] = {
    0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

### Example 9.7 Using static arrays

#### a) Speedup obtained using a static array

To understand the speedup obtained using a `static` and a `static const` array, here is another minimal example. The `get_prime` function given below uses a static local array named `prime`, initialized with the first ten prime numbers, to return the  $i$ th prime number,  $1 < i < 10$ .

```
/* get i th prime number (1 <= i <= 10) */
int get_prime(int i)
{
    static int prime[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 27};

    return prime[i - 1];
}
```

This function was called ten million times from the `main` function using a `for` loop.

```
int main()
{
    int i;

    for (i = 0; i < 100000000; i++) {
        int p = get_prime(i % 10);
    }

    return 0;
}
```

The non-static array implementation required about 0.7 seconds for execution, whereas the static arrays required about 0.2 seconds only indicating a substantial speed-up.

### b) Using a static array as function memory

Consider the function `letter_count` given below

```
/* count letters A to Z */
void letter_count(char ch)
{
    static int count[26];

    ch = toupper(ch);
    if (ch >= 'A' && ch <= 'Z')
        count[ch - 'A']++;
}
```

This function uses a `static` array named `count` to store the counts of letters A to Z. Note that all elements of `count` array will be initialized to zero the first time this function is called and these counts will be remembered across function calls. The function accepts a character `ch`, converts it to uppercase using the `toupper` function and if it a letter, updates the corresponding letter count.

Although this function counts the letters correctly, it is not very useful as the `count` array is local to the function. Of course, we can use these counts for some useful purpose. For example, the above function has been modified below to print these counts after every 100 calls to this function. For this, it uses another static variable (`n`) that counts the function calls.

```
/* count letters A to Z and print counts after every 100 letters */
void letter_count(char ch)
{
```

```

static int count[26], n;

ch = toupper(ch);
if (ch >= 'A' && ch <= 'Z') {
    count[ch - 'A']++; /* update letter count */
    n++;                /* no of letters counted */
}

/* print letter counts after every 100 letter added */
if (n % 100 == 0) {
    int c;
    printf("\n");
    for(c = 'A'; c <= 'Z'; c++)
        printf("%3c", c);
    printf("\n");
    for(c = 0; c < 26; c++)
        printf("%3d", count[c]);
    printf("\n");
}
}

```

Now consider that these counts are to be used in some other function. As we know, a function cannot return an array. However, we can return a pointer to a **static** array.

#### 9.4.3 External or Global Arrays

As in case of scalar variables, we can also use **external or global arrays** in a program, i. e., the arrays which are defined outside any function. These arrays have global scope. Thus, they can be used anywhere in the program. They are created in the beginning of program execution and last till its end. Also, all the elements of these arrays are initialized to default values (zero for arithmetic types and NULL for pointers). Such arrays are usually defined at the beginning of a program, as illustrated below.

```

#include <stdio.h>
void ivec_read();
void ivec_print();

int a[10];      /* external array */
int n;

int main()
{
    ivec_read();
    ivec_print();
}

```

```

    return 0;
}

void ivec_read()
{
    int i;

    scanf("%d", &n);
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
}

void ivec_print()
{
    int i;
    for(i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

```

Note that this approach avoids passing arrays as parameters and thereby simplifies the program and also saves some time. However, the use of global arrays has some limitations.

- 1. These functions work only with particular array(s) and cannot be used for performing similar operations on other arrays.
- 2. As the arrays can be modified from any part of the program, debugging such programs is more difficult. However, this approach can be particularly useful with `const` arrays that need to be shared among several functions. For example, the `month_days` array required in date processing applications.

As with scalar variables, external arrays can be defined anywhere in a program, but outside any function of course. If we use such an array before it is defined, we should provide an `extern` declaration, as illustrated in the program given below.

```

#include <stdio.h>
void ivec_read();
void ivec_print();

int n;

int main()
{
    ivec_read();

```

```

ivec_print();

return 0;
}

void ivec_read()
{
    int i;
    extern int a[];      /* external array declaration */

    scanf("%d", &n);
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
}

int a[10];           /* external array definition */

void ivec_print()
{
    int i;

    for(i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

```

The `extern` declaration of an array should match with its definition with respect to array size, data type, type modifier and storage class; otherwise, the compiler will report an error. However, note that the array size can be omitted in an `extern` declaration. This declaration does not create an array and it can be specified any number of times, even within the same scope.

## Exercises

1. Answer the following questions:
  - a. How is the last element of array `a` having 10 elements accessed?
  - b. What are the initial values of array elements if we do not explicitly initialize the array?
  - c. How many values will be initialized from an array `x` having 20 elements if the initializer list contains five elements?
  - d. How much memory is required for an array of type `float` having 10 elements?
  - e. Which parameter passing method is used to pass an array to a function?

- f. Is it possible to modify the parameter array within a function without modifying the corresponding argument array?
  - g. Can we return an array from a function using the `return` statement?
  - h. Is it possible to access an array element using the array subscript operator without mentioning the location of the array element?
    - i. Can we copy an array using the assignment operator?
  - l. Determine errors in the following program segments:
- a. 

```
float x[10];
for(i = 1; i <= 10; i++)
    scanf("%f", &x[i]);
```
  - b. 

```
float x(5);
for (j = 0; j < 5; j++)
    x(i) = j;
```
  - c. 

```
int b[10];
for (i = 0; i < 10; i++)
    for (j = 1; j <= 10; j++)
        b [i] = j;
```
  - d. 

```
int x[];
for (m = 1; m < 20; m++)
    x[i]++;
```
  - e. 

```
int show_arr(int a[], int 10)
{
    for (i =0; i < 10; i++)
        printf(a[i]);
}
```
  - f. 

```
int n = 10;
int a[n], b[n];
...
for (i = 0; i < n; i++)
    b[i] = a[i];
```
  - g. 

```
#define NMAX 20

int num[NMAX];
for(k = NMAX; k >= 0; k--)
    printf("%d", num[k]);
```
  - h. /\* generate and return an array having
n random numbers \*/

```
int[] ivec_rand(int n)
{
    int a[10];
    for(i = 0; i < n; i++)
        a[i] = rand();
    return a;
}
```

3. Determine the purpose of the following program segments:

a. `int a[10] = {1, 2, 3, 1, 2, 3};`  
`int n = 6;`

```
for(i = 0; i < n - 1; i++) {
    for(j = i + 1; j < n; j++) {
        if (a[i] == a[j])
            a[j--] = a[--n];
    }
}
```

b. `int x[20];`  
`int n = 8; /* elements in x */`  
`int loc;`

```
scanf("%d", &loc);
for( ; loc < n - 1; loc++)
    x[loc] = x[loc + 1];
n--;
```

c. `int num[10], i;`  
`for(i = 0; i < 10; i += 2)`  
    `if (num[i] > num[i+1]) {`  
        `int temp = num[i];`  
        `num[i] = num[i + 1];`  
        `num[i + 1] = temp;`  
    `}`

d. `int a[50], i;`  
`int n = 10; /* elements in a */`  
  
`for (i = 0, j = 0; i < n; i++)`  
    `if (a[i] < 0 && j < i)`  
        `a[j++] = a[i];`  
`n = j;`

e. `int x[50], i;`

```

int n = 10; /* elements in x */

for (i = 0, j = n - 1; i < j; i++)
    if (x[i] < 0) {
        int temp = x[j];
        x[i] = a[j];
        x[j] = temp;
        j--;
    }
n = j + 1;

f. int ivec_some_op(int vec[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for(j= i + 1; j < n; j++)
            if (vec[i] == vec[j])
                return 0;
    }

    return 1;
}

g. int num[12];

for (i = 0; i <= 11; i++)
    ++num[i];

```

- l. Define functions for the following operations on vectors (one-dimensional arrays) of numbers:
  - a. Perform pair-wise exchange on array elements starting from the beginning, i. e., exchange numbers at positions 0 and 1, 2 and 3, and so on.
  - b. Subtract the mean of array elements from each element in a given array.
  - c. Test whether the elements of a vector are sorted in ascending order or not.
  - d. Determine the second largest value in a given array.
  - e. Test if all the digits of a number are unique or not.
  - f. Delete duplicate elements in a vector, i. e., convert it to a set.
  - g. Obtain a vector that contains distinct elements from two vector (set union operation).
  - h. Merge two sorted arrays to obtain a sorted array.

## Exercises (Advanced Concepts)

l. Determine the output of the following programs (#includes have been omitted to save space):

a.

```
int a[5] = {1, 2};
void f();
int main()
{
    f();
    f();
}
void f()
{
    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", a[i]++);
}
```

b.

```
void f();
int main()
{
    f();
    f();
}

void f()
{
    extern int a[];
    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", ++a[i]);
}
int a[5];
```

c.

```
void f();
int main()
{
    f();
    f();
}

void f()
{
    int a[5], i;
    for (i = 0; i < 5; i++)
        printf("%d ", ++a[i]);
}
```

d. int main()  
{  
 void f();  
 f();  
 f();  
}  
  
void f()  
{  
 static int a[5], i;  
 for (i = 0; i < 5; i++)  
 printf("%d ", a[i]++);  
}

e. int a[5] = {1, 2, 3, 4, 5};  
void f();  
int main()  
{  
 f();  
 f();  
}  
  
void f()  
{  
 int i;  
 int a[5] = {};  
 for (i = 0; i < 5; i++)  
 printf("%d ", ++a[i]);  
}

f. int a[5] = {1, 2, 3, 4, 5};  
void f();  
void g();  
  
int main()  
{  
 g();  
 f();  
}  
  
void f()  
{  
 int i;  
 for (i = 0; i < 5; i++)

```
        printf("%d ", ++a[i]);
}
void g()
{
    static int a[] = {6, 7, 8};
}
```

2. Identify errors, if any, in the following programs (#includes have been omitted to save space):

a. void func();  
int main()  
{  
 func();  
}

```
void func()
{
    extern a[10];
    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", a[i]);
}
```

```
int a[5];
```

b. void func();  
int main()  
{  
 func();  
}

```
void func()
{
    extern int a[];
    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", ++a[i]);
}
```

```
unsigned int a[5];
```

c. void func();  
int main()  
{  
 extern int a[5];

```

        func();
    }

void func()
{
    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", ++a[i]);
}
int a[5];

d. void func();
extern int a[5];
int main()
{
    func();
}
extern int a[5];

void func()
{
    extern int a[5];
    extern int a[];
    extern a[];
    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", ++a[i]);
}
int a[5];

```

3. Answer the following questions:

- a. How can we ensure that a function does not modify the elements of a parameter array?
- b. When is it possible to omit the array size?
- c. What are the default values of elements in a static array?
- d. What is the effect of a storage class on the memory required for storing an array?
- e. Which properties of external and static arrays are similar?
- f. What is the difference between an automatic array and a static array?

# 10 Matrices and Multidimensional Array

In the last chapter, we learned that arrays are collections of elements of the same data type. This chapter presents two-dimensional arrays (also called matrices) and multidimensional arrays. It includes array declaration, elements access, operations on individual elements, operations on entire arrays using loops and array initialization. Next, a discussion on how these arrays are used as function parameters is presented. In the *Advanced Concepts* section, we will study the use of storage classes, memory requirements and some other concepts. Numerous examples are provided throughout this chapter.

## 10.1 Two-dimensional Arrays or Matrices

A **two-dimensional array** (commonly called a *matrix*) consists of elements of the same type arranged in rows and columns. The rows and columns of a matrix are numbered starting from 0 as illustrated in Fig. 9.1b. Thus, for an array of size  $rows \times cols$ , the row subscripts are in the range 0 to  $rows - 1$  and the column subscripts are in the range 0 to  $cols - 1$ .

### 10.1.1 Declaration

The declaration of a two-dimensional array takes the following form:

*arr\_type arr\_name [ rows ] [ cols ];*

where *arr\_name* is the name of the array being declared, each element of which is of type *arr\_type*. The expressions *rows* and *cols* enclosed in square brackets are constant integral expressions (or integral constants) that specify the number of rows and columns in the array. When a compiler comes across the declaration of an array, it reserves memory for the specified number of array elements. For a two-dimensional array, the memory required is given as *rows \* cols \* sizeof (arr\_type)*.

Note that we can declare matrices of built-in as well as user-defined types; we can also declare multiple matrices in a single declaration statement, separated by commas. Scalar variables, vectors and multidimensional arrays can also be declared along with matrices.

### 10.1.2 Accessing Matrix Elements

An element of a matrix can be accessed by writing the array name followed by row and column subscript expressions within separate array subscript operators as shown below.

```
arr_name [ row_expr ] [ col_expr ]
```

where *row\_expr* and *col\_expr* are integral expressions that specify the row and column positions, respectively, of an element in the array. As mentioned earlier, for an element access to be valid, the values of *row\_expr* should be in the range 0 to *rows* – 1 and the values of *col\_expr* should be in the range 0 to *cols* – 1.

### 10.1.3 Operations on Matrix Elements

We can perform various operations on the elements of a matrix, similar to those for scalar variables and elements of a vector. For a matrix of any built-in type, we can use its elements in an expression, assign a value to it, perform increment and decrement operations, pass it as an argument to a function and so on. In particular, the prefix and postfix increment (and decrement) operators and the address of operator can be used as follows:

```
++ arr_name [ row ] [ col ]
arr_name [ row ] [ col ] ++
&arr_name [ row ] [ col ]
```

In these expressions, the array subscript operators are bound first to their operands followed by the `++` and `&` operators. Thus, parentheses are not required in these expressions.

### 10.1.4 Operations on Entire Matrices

As with vectors, C does not provide any statements or operators to perform operations on an entire two-dimensional array. Nested for loops are usually used to perform operations on an entire array as illustrated below for a two-dimensional array *a* of size  $m \times n$ :

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        /* process element a[i][j] */
    }
}
```

The index variable *i* (of the outer `for` loop) assumes the values 0, 1, ..., *m* - 1. For each value of *i*, the index variable of the inner `for` loop (*j*) assumes the values 0, 1, ..., *n* - 1. Thus, the use of the expression *a*[*i*][*j*] causes the elements to be accessed in row-wise manner in the order *a*[0][0], *a*[0][1], ..., *a*[0][*n*-1], *a*[1][0], *a*[1][1], ..., *a*[1][9], ..., *a*[*m*-1][0], *a*[*m*-1][1], ..., *a*[*m*-1][*n*-1]. Note that we can use the expression *a*[*j*][*i*] to access array elements in a column-wise manner.

#### Example 10.1 Operations on matrices

This example illustrates simple operations on matrices. These include array initialization and console

I/O. Assume that array **a** is an integer matrix of size  $10 \times 10$ , declared as follows:

```
int a[10][10];
```

#### a) Initialize a matrix

```
for (i = 0; i < 10; i++) {  
    for (j = 0; j < 10; j++)  
        a[i][j] = 0;  
}
```

This 2-level nested **for** loop initializes each element of matrix **a** to 0. However, if we want to initialize a matrix of size  $m \times n$  (where  $m, n < 10$ ), we need to modify the above program segment as follows:

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++)  
        a[i][j] = 0;  
}
```

#### b) Print a matrix on the screen

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++)  
        printf("%4d ", a[i][j]);  
    printf("\n");  
}
```

This code segment prints matrix **a** of size  $m \times n$  on the screen. The index variable in the outer **for** loop (**i**), corresponds to a row of the matrix. Thus, the inner **for** loop prints the *i*th row of the matrix. Observe the **printf("\n")** statement in the outer **for** loop that causes the cursor to move to the next line after the *i*th row is printed. Thus, each row is printed on a line by itself.

Next observe that the **printf** statement in the inner **for** loop uses the format "%4d ". Thus, each array element is printed using a field width of 4 character positions. This causes the columns in the array to be aligned properly (assuming that each array element fits in 4 characters).

#### c) Read a matrix from the keyboard

```
printf("Enter matrix size (max 10 x 10): ");  
scanf("%d %d", &m, &n);  
printf("Enter the elements of matrix:\n");  
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++)  
        scanf("%d", &a[i][j]);
```

```
}
```

This program segment reads a matrix of size  $m \times n$  from the keyboard and stores it in array a of size  $10 \times 10$ . The remaining elements, if any, of matrix a are unused. The matrix size is accepted first followed by the elements of the matrix in a row-wise manner. The outer **for** loop is set up to read all the rows in the matrix, whereas the inner **for** loop is set up to read all the elements of the  $i$ th row. Note the use of the **printf** statements to display prompt messages to facilitate data entry. The output of the program containing this code is given below.

```
Enter matrix size (max 10 x 10): 2 3
```

```
Enter the elements of matrix:
```

```
11 12 13
```

```
21 22 23
```

Observe that each matrix row has been typed on a line by itself to improve readability although it is not really essential. We should be careful while entering data as the size of the input matrix should not exceed the matrix size specified in its definition. Thus, the values of  $m$  and  $n$  in this example should not exceed 10. We can use a **do ... while** loop to ensure that the user enters correct values for  $m$  and  $n$ , as shown below:

```
do {  
    printf("Enter matrix size (max 10 x 10): ");  
    scanf("%d %d", &m, &n);  
} while (m <= 0 || m > 10 || n <= 0 || n >10);  
  
printf("Enter the elements of matrix:\n");  
...
```

If the user enters wrong values for  $m$  and  $n$ , this loop prompts the user to reenter the matrix size as shown below.

```
Enter matrix size (max 10 x 10): 20 20
```

```
Enter matrix size (max 10 x 10): 3 20
```

```
Enter matrix size (max 10 x 10): 3 3
```

```
Enter the elements of matrix:
```

```
-
```

As shown above, the program accepts elements of matrix only when the user enters a valid matrix size. If the user enters invalid values, we may wish to print an appropriate message before the values are accepted again. This can be achieved by using a flag (named **valid**) as shown below.

```
valid = 0;  
do {  
    printf("Enter matrix size (max 10 x 10): ");  
    scanf("%d %d", &m, &n);  
    if (m <= 0 || m > 10 || n <= 0 || n > 10)
```

```
        printf("Invalid matrix size\n");
    else valid = 1;
} while (!valid);
printf("Enter the elements of matrix:\n");
...
```

This further facilitates data entry as shown below.

```
Enter matrix size (max 10 x 10): 20 20
Invalid matrix size
Enter matrix size (max 10 x 10): 3 3
Enter the elements of matrix:
```

—

#### d) Initialize a matrix with random values

During the program development phase, a program may contain several errors or bugs. To verify that the program is correct and free of bugs, we may have to test it several times. This process is time-consuming and tiresome particularly when we need to enter large volumes of data from the keyboard as in case of one or more matrices. In such situations, we can use a random number generator to automatically initialize matrices and speed up program development.

To generate random values, we can use the `rand` function from the `stdlib.h` header file. The program segment given below initializes an  $m \times n$  sized portion of matrix `a` with random values. The remaining elements are not affected.

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        a[i][j] = rand();
}
```

The `rand` function generates random integer numbers in the range 0 to `RAND_MAX`, a constant defined in the `stdlib.h` header file with value 32,767. Thus, the above code initializes a matrix with random integer values in the range 0 to 32,767. This may not be suitable in many situations. More often, we may have to initialize the matrix with random values in a specific range, say  $[a_{\min}, a_{\max}]$ . This can be achieved as follows:

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        a[i][j] = amin + (amax - amin) * (float) rand() / RAND_MAX;
}
```

Note the use of `(float)` typecast in this code to avoid integer division.

## Example 10.2 More operations on matrices

This example illustrates how various operations can be performed on matrices. These include matrix copy, addition/subtraction, multiplication, transpose, etc. Assume that the matrices **a**, **b** and **c** used in these examples are of integer type of size  $10 \times 10$ , defined as follows:

```
int a[10][10], b[10][10], c[10][10];
```

### a) Matrix copy

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++)  
        b[i][j] = a[i][j];  
}
```

In this example, a matrix **a** of size  $m \times n$  is copied to another matrix **b**. This involves copying each element of matrix **a** to the corresponding position of matrix **b**, i. e.,  $b_{ij} = a_{ij}$ .

### b) Matrix addition and subtraction

Consider two matrices **a** and **b** of the same size  $m \times n$ . Each element of the addition matrix **c** is obtained by adding the corresponding elements in matrices **a** and **b**, i. e.,  $c_{ij} = a_{ij} + b_{ij}$ . Note that the addition matrix is also of size  $m \times n$ . The program segment given below adds two matrices **a** and **b** of size  $m \times n$ .

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++)  
        c[i][j] = a[i][j] + b[i][j];  
}
```

Note that the matrix subtraction operation is similar to matrix addition. Thus, the code for subtraction of matrix **b** from matrix **a** (both of size  $m \times n$ ) can be obtained by replacing the '+' sign in last line with the '-' sign.

### c) Matrix Transpose

Consider matrix transpose operation  $b = a^T$  where **a** is a square matrix of size  $m$ . This transpose operation yields a square matrix of size  $m$  whose elements are given as  $b_{ij} = a_{ji}$ . The program segment given below uses nested **for** loops to obtain this transpose.

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < m; j++)
```

```

    b[i][j] = a[j][i];
}

```

Now consider the transpose of a matrix of size  $m \times n$ . In this case, the result matrix **b** is of size  $n \times m$ . The modified code is given below.

```

for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        b[i][j] = a[j][i];
}

```

#### d) In-place transpose of a square matrix

Consider a square matrix **a** of size  $m$ . We can write the code to obtain an in-place transpose of this matrix, i. e., the transpose matrix will be stored in matrix **a** itself. The elements of transposed matrix **a** are given as  $a_{ij} = a_{ji}$ . It is easy to understand that this operation actually requires an exchange of elements  $a_{ij}$  and  $a_{ji}$ . Consider the (incorrect) code given below to obtain an in-place transpose.

```

/* wrong code */
for (i = 0; i < m; i++) {
    for (j = 0; j < m; j++) {
        int temp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = temp;
    }
}

```

The problem is that for each pair  $(a_{ij}, a_{ji})$  the exchange operation is performed twice. Thus, the result is the original matrix. For correct in-place matrix transpose, we need to perform the exchange operation only once for each pair. For this, we can set up **for** loops such that the values of index variables **i**, **j** refer only to the elements in either the lower or the upper triangular matrix (excluding the elements in the leading diagonal). The modified code in which loop variables assume values corresponding to the lower triangular matrix is given below.

```

/* matrix transpose - loop variables assumes values corresponding to
   lower triangular matrix */
for (i = 1; i < m; i++) {
    for (j = 0; j < i; j++) {
        int temp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = temp;
    }
}

```

Alternatively, we can rewrite the code in which the loop variables assume the values corresponding to the upper triangular matrix as shown below:

```
/* matrix transpose - loop variables assumes values corresponding to
   upper triangular matrix */
for (i = 0; i < m-1; i++) {
    for (j = i+1; j < m; j++) {
        int temp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = temp;
    }
}
```

### e) Matrix multiplication

Consider two matrices  $a_{m \times n}$  and  $b_{n \times p}$ , i. e., the number of columns of matrix  $a$  is equal to the number of rows of matrix  $b$ . The matrix multiplication operation,  $a \times b$ , gives a matrix of size  $m \times p$ . Let us denote this matrix as  $c$ . Thus,  $c_{m \times p} = a_{m \times n} \times b_{n \times p}$ .

The  $ij$ th element of the multiplication matrix, i. e.,  $c_{ij}$  is calculated as the sum of products of each element of the  $i$ th row of matrix  $a$  with the corresponding elements of the  $j$ th column of matrix  $b$ . Mathematically,

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} * b_{kj}$$

The program segment given below uses three nested **for** loops to obtain the multiplication matrix  $c$ . Two outer **for** loops are set up to process each element in this matrix. The innermost **for** loop calculates the value of  $c_{ij}$ . Note that element  $c_{ij}$  is initialized to 0 before this loop.

```
for (i = 0; i < m; i++) {
    for (j = 0; j < p; j++) {
        c[i][j] = 0;
        for (k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
}
```

### Program 10.1 Matrix addition

Write a program to add two matrices of size  $m \times n$  and print the result matrix.

**Solution:** In Example 10.2b, we learned how to add two matrices of size  $m \times n$  using nested **for** loops. The result is also a matrix of size  $m \times n$ . Assume that the maximum size of the matrices to be added is

$10 \times 10$ . The steps required for this program are given below.

Read matrix size m, n

Read matrix a

Read matrix b

Add matrices, c = a + b

Print matrix c

The program given below declares three integer matrices a, b and c of size  $10 \times 10$  and variables m and n as the actual size of the matrices. It first accepts the values of m and n followed by matrix a and matrix b. Then these matrices are added to determine the addition matrix c. Finally, matrix c is printed.

```
/* Matrix addition program */
#include <stdio.h>

int main()
{
    float a[10][10], b[10][10], c[10][10];
    int m, n;          /* actual matrix size */
    int i, j;

    printf("Enter matrix size (rows and columns): ");
    scanf("%d %d", &m, &n);

    /* read matrix a */
    printf("\nEnter matrix a:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            scanf("%f", &a[i][j]);
    }

    /* read matrix b */
    printf("\nEnter matrix b:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            scanf("%f", &b[i][j]);
    }

    /* perform matrix addition */
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
    }
}
```

```

}
/* print addition matrix C */
printf("\nAddition matrix is:\n");
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        printf("%5.2f ", c[i][j]);
    printf("\n");
}
return 0;
}

```

The program output is given below.

```

Enter matrix size (rows and columns): 24

Enter matrix a:
1 2 3 4
5 6 7 8

Enter matrix b:
8 7 6 5
4 3 2 1

Addition matrix is:
  9.0 9.00 9.00 9.00
  9.0 9.00 9.00 9.00

```

Note that the last two steps can be combined to determine the result matrix c and print it as shown below.

```

/* add matrices and print result matrix */
printf("\nAddition matrix is:\n");
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        c[i][j] = a[i][j] + b[i][j];
        printf("%5.2f ", c[i][j]);
    }
    printf("\n");
}

```

#### 10.1.5 Initialization

A matrix can be initialized using a syntax similar to that used for the initialization of vectors. Thus, the declaration

```
int mat[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

declares an integer matrix mat of size  $3 \times 4$  and initializes it in a row-wise manner as shown in Fig. 10.1a. The readability of this initialization statement can be greatly improved by writing the initialization list in the form of a  $3 \times 4$  matrix as shown below:

```
int mat[3][4] = {  
    1, 2, 3, 4,  
    5, 6, 7, 8,  
    9, 10, 11, 12  
};
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

(a)

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

(b)

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	0	0	0

(c)

Fig. 10.1

Similarly, the following declaration initializes an integer unit matrix of size  $4 \times 4$  as shown in Fig. 10.1b.

```
int unit_mat[4][4] = {  
    1, 0, 0, 0,  
    0, 1, 0, 0,  
    0, 0, 1, 0,  
    0, 0, 0, 1  
};
```

It is possible to omit the first dimension (i. e., the number of rows) of a two-dimensional array in an initialization statement. The number of rows in the matrix is determined from the number of elements specified in the initialization list. Consider the example given below.

```
int x[][][5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

This initialization statement declares an array x with five elements in each row and initializes it with 12 elements. Thus, the number of rows is determined as  $\lceil 12/5 \rceil$ , i. e., 3. Moreover, as the initialization list does not specify the values for the last three elements in the third row, they are initialized to 0. Thus, the

array `x` is initialized as shown in Fig. 10.1c.

C language allows the initialization list for each row to be enclosed within a pair of braces. Thus, the declaration of array `mat` given above can alternatively be written as follows:

```
int mat[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

Observe that the initialization lists for different rows are separated by commas. The initialization list for each row is usually written on a separate line to improve readability as shown below.

```
int mat[3][4] = {
    { 1, 2, 3, 4},
    { 5, 6, 7, 8},
    { 9, 10, 11, 12}
};
```

Note that if the initialization list for a row of a matrix contains fewer values than required, the remaining elements in that row are initialized to the default value, which is 0 for arithmetic data types. Thus, the initialization of array `unit_mat` declared above can also be declared as follows:

```
int unit_mat[4][4] = {{1}, {0,1}, {0,0,1}, {0,0,0,1}};
```

Finally note that if the initialization list for the entire array or one of its row contains more elements than required, the compiler gives a *Too many initializers* error.

### 10.1.6 Matrices as Function Parameters

As mentioned in the previous chapter, we can pass one or more arrays as parameters to a function. These array parameters can be of different dimensions and types. Moreover, we can mix the scalar and array parameters. We can thus define functions having one or more matrices as parameters. A function that has one matrix parameter is defined as

```
ret_type func_name(mat_type mat_name [rows] [cols])
{
    /* process matrix mat_name */
}
```

The function `func_name` takes a matrix `mat_name` of type `mat_type` and size `rows × cols` as a parameter and returns a value of type `ret_type`. Note that the first array dimension in the above definition (i. e., `rows`) is optional and can be omitted. However, we cannot omit the second dimension. Thus, the above function definition can also be written as shown below.

```
ret_type func_name(mat_type mat_name [ ] [cols]) { ... }
```

The argument array specified in a call to this function can have any number of rows but it should have

*cols* number of columns, the same as that of the parameter array. However, if the number of columns is different, the compiler gives either a warning message or an error. For example, TC++ gives a warning (*Suspicious pointer conversion*) and allows program execution giving wrong results due to different interpretation of data in the argument matrix. Thus, a programmer should be careful particularly while passing matrices (and multidimensional arrays) using old compilers like TC/TC++. The Code::Blocks, on the other hand, reports an error (*Passing argument 1 from incompatible pointer type*) and does not allow program execution.

The function given above can be generalized to process matrices of different sizes by including two integer parameters, say *m* and *n*, that specify the actual size of the matrix (number of rows and columns, respectively) to be processed from the argument array. However, note that the maximum number of columns in the argument array in a function call should still match those in the parameter array in the function definition.

As we know, the operations on a matrix typically consist of performing the same or a similar operation on each of its element. Thus, a function that operates on the entire matrix typically uses a nested **for** loop as shown below.

```
void imat_func(int mat[][10], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            /* operate on mat[i][j] */
        }
    }
}
```

Some calls to this function are shown below.

```
int main()
{
    int a[10][10], b[5][10], c[20][10], d[10][5];
    imat_func(a, 10, 10);      /* ok */
    imat_func(b, 3, 4);       /* ok */
    imat_func(c, 15, 10);     /* ok */
    imat_func(d, 5, 5);       /* wrong: dimension of d should be [][][] */
    imat_func(b, 7, 4);       /* wrong: max value of m should be 5 */
    return 0;
}
```

Recall that for efficiency reasons (to avoid unnecessary copying of an entire array when a function is called), C language uses the *call by reference* mechanism to pass arrays. In this mechanism, only the starting address of array argument is passed to the called function. This makes an array an input-output parameter. When the function modifies elements of the parameter array, it actually modifies the

elements of the argument array in the called function. Thus, the modified array is available in the calling function. If this behavior is not desirable, we can use a `const` array or a *pointer to a const* mechanism (Chapter 12).

### Example 10.3 Operations on matrices using functions

In Example 10.1 and 10.2, we studied several operations on matrices. In this example, we will look at definitions of functions to perform various operations on matrices. These include console I/O, initialization, in-place transpose, addition and multiplication. The argument arrays are assumed to have 10 columns and the size of the actual matrix to be processed ( $m \times n$  in most cases) is accepted as the function parameters.

#### a) Print a matrix on the screen

```
void fmat_print(float a[][10], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++) {
        for(j = 0; j < n; j++)
            printf("%8.2f ", a[i][j]);
        printf("\n");
    }
}
```

This function prints a `float` matrix of size  $m \times n$  on the screen, where  $n \leq 10$ . Note that to properly align the columns of the matrix, each matrix element is printed using the format `8 . 2f`.

#### b) Read a matrix from the keyboard

```
void fmat_read(float a[][10], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++) {
        for(j = 0; j < n; j++)
            scanf("%f", &a[i][j]);
    }
}
```

This function reads a `float` matrix of size  $m \times n$  from the keyboard. The `scanf` statement reads the value of `a[i][j]`, i. e., the  $ij$ th element of the parameter matrix. As a matrix is passed by reference, an element read from the keyboard is actually stored in the  $ij$ th element of the argument array. Thus, the

matrix read from the keyboard is available in the argument matrix in the calling function.

#### c) Initialize a matrix with a specified value

```
void fmat_init(float a[][10], int m, int n, float val)
{
    int i, j;
    for(i = 0; i < m; i++) {
        for(j = 0; j < n; j++)
            a[i][j] = val;
    }
}
```

This function initializes each element of a matrix of size  $m \times n$  to a specified value `val`. Note that the initialization of matrix `a` in this function actually causes the argument matrix specified in the calling function to be initialized.

#### d) Initialize a square matrix as a unit matrix

```
void fmat_unit(float a[][10], int m)
{
    int i, j;
    for(i = 0; i < m; i++) {
        for(j = 0; j < m; j++) {
            if (i == j)
                a[i][j] = 1;
            else a[i][j] = 0;
        }
    }
}
```

This function initializes a square matrix of size  $m$  as a unit matrix. In a unit matrix, the elements in the leading diagonal are 1 and all other elements are 0. To initialize the elements, use an `if` statement within the nested `for` loops. If the values of loop variables `i` and `j` are equal, the element `a[i][j]` is initialized to 1; otherwise 0.

#### e) In-place transpose of a square matrix

```
void fmat_transpose(float a[][10], int m)
{
    int i, j;
```

```

for(i = 1; i < m; i++) {
    for(j = 0; j < i; j++) {
        float temp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = temp;
    }
}
}

```

This function obtains an in-place transpose of a given square matrix of size  $m$  using the technique explained in Example 10.two-dimensional.

#### f) Matrix addition

```

void fmat_add(float a[][10], float b[][10], float c[][10],
int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++) {
        for(j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
    }
}

```

This function adds two matrices **a** and **b**, each of size  $m \times n$ , to obtain matrix **c** which is also of size  $m \times n$ . As we know, each element of matrix **c** is obtained by adding the corresponding elements of matrices **a** and **b** as  $c_{ij} = a_{ij} + b_{ij}$ .

#### g) Matrix multiplication

```

void fmat_mult(float a[][10], float b[][10], float c[][10],
int m, int n, int p)
{
    int i, j, k;

    for(i = 0; i < m; i++) {
        for(j = 0; j < p; j++) {
            c[i][j] = 0;
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
}

```

}

This function multiplies matrix a of size  $m \times n$  with matrix b of size  $n \times p$  to obtain a matrix c of size  $m \times p$  as explained in Example 10.2e.

### Program 10.2 Matrix multiplication using functions

Write a complete program to multiply two matrices using functions.

**Solution:** We can use functions to read a matrix from the keyboard, print a matrix on the screen and multiply two matrices. The definitions for these functions are given in Example 10.3. Let the matrices to be multiplied be named a and b and the multiplication matrix as c. The steps to be performed in the main function can be written as follows:

Read m, n, p (the sizes of matrices a and b)

Read matrix a

Read matrix b

Perform multiplication of matrices a and b to obtain matrix c

Print matrix c

End

In the program given below, the definitions of functions mat\_read, mat\_print and mat\_mult are not included to save space.

```
/* Matrix multiplication using functions */
#include <stdio.h>
#include <stdlib.h>

void fmat_read(float a[][10], int m, int n);
void fmat_print(float a[][10], int m, int n);
void fmat_mult(float a[][10], float b[][10], float c[][10],
               int m, int n, int p);
int main()
{
    float a[10][10], b[10][10]; /* input matrices */
    float c[10][10]; /* output matrix */
    int m, n, p; /* matrix a of size m x n and b of size n x p */

    printf("--- Matrix multiplication ---\n");

    /* read matrix sizes */
    printf("Enter m, n, p (max 10): ");
    scanf("%d %d %d", &m, &n, &p);

    /* read matrices a and b */
```

```

printf("Enter elements of matrix a (%d x %d):\n", m, n);
fmat_read(a, m, n);
printf("\nEnter elements of matrix b (%d x %d):\n", n, p);
fmat_read(b, n, p);

/* perform multiplication and print result matrix */
fmat_mult(a, b, c, m, n, p);
printf("\nMultiplication of given matrices:\n");
fmat_print(c, m, p);

return 0;
}

/* include fmat_read, fmat_print and fmat_mult functions from Ex. 10.

```

The program output is given below.

```

--- Matrix multiplication ---
Enter Enter m, n, p (max 10): 3 2 3
Enter elements of matrix a (3 x 2):
1 2
3 4
5 6

Enter elements of matrix b (2 x 3)
1 2 3
4 5 6

Multiplication of given matrices:
    9.00  12.00  15.00
   19.00  26.00  33.00
   29.00  40.00  51.00

```

## 10.2 Multidimensional Arrays

C language permits the use of **multidimensional arrays**. It is easy to visualize arrays up to three dimensions. However, we may have difficulty in visualizing a four-, five- or in general, an  $n$ -dimensional array.

A **four-dimensional array** can be thought of as a one-dimensional array in which each element is a three-dimensional array or as a matrix in which each element itself is a matrix or even as a three dimensional array having one-dimensional arrays as its elements.

Consider for example a school having six classes (5 to 10) each having up to three divisions (A, B and C). Each division can have up to 60 students and in any particular examination, each student has to appear for up to 10 subjects depending on his/her class.

Consider now that we wish to store the marks scored by students in a particular examination. The marks obtained by a student can be stored in a vector of size 10, whereas we require a matrix of size  $60 \times 10$  to store the marks of all the students in any division and a three-dimensional array of size  $3 \times 60 \times 10$  to store the marks of all the students in a particular class. Now how can we store the marks of all the students in the school? By extending the above discussion, we can use a four-dimensional array of size  $6 \times 3 \times 60 \times 10$ . This array can now be considered as six arrays of size  $3 \times 60 \times 10$  or as a matrix of size  $6 \times 3$  whose individual elements are matrices of size  $60 \times 10$  or even as a three-dimensional array of size  $6 \times 3 \times 60$  where each element is a vector of size 10.

We can extend this discussion further and use a five-dimensional array to store the data of all the examination conducted by this school in an academic year and a six-dimensional array to store the data for several years.

### 10.2.1 Declaration

The **declaration** of an  $n$ -dimensional array takes the following form:

```
arr_type arr_name [ size1 ] [ size2 ] ... [ size_n ] ;
```

where *arr\_name* is the name of the array being declared, each element of which is of type *arr\_type*. The expressions *size<sub>1</sub>*, *size<sub>2</sub>*, ..., *size<sub>n</sub>* enclosed in square brackets are integral constants or constant integral expressions that specify the number of elements in each dimension of the array.

As with vectors and matrices, we can declare multidimensional arrays of any built-in or user-defined type and the array elements are numbered starting from 0 (and not from 1) in each dimension.

#### Example 10.4 Declaring multidimensional arrays

##### a) Array to store temperature data

```
int temp[4][12][31][24];
```

This statement declares a four-dimensional array that can be used to store the temperatures of four cities recorded each hour of the day over a period of one year. The numbers 12, 31 and 24 represent the months in a year, the maximum number of days in a month and the hours in a day, respectively. Note that some of the entries in this array will remain unused as several months do not have 31 days. To minimize the unused memory of the array, we can declare a three-dimensional array to store this data as follows:

```
int temp[4][366][24];
```

where 366 represents the maximum number of days in a year. However, the program logic may become involved as we may be more interested in taking into account the month and day-of-month rather than day-of-year.

#### b) Array to store the marks of students in a school

```
#define CLASS 6
#define DIV 3
#define STUD 60
#define SUB 10
int marks[CLASS][DIV][STUD][SUB];
```

The first four statements define symbolic constants CLASS, DIV, STUD and SUB with values 6, 3, 60 and 10, respectively. The last statement then declares a four-dimensional array **marks** of type **int** and size  $6 \times 3 \times 60 \times 10$ . We can use this array to store the marks of students in six classes each having up to three divisions with maximum of 60 students each having taken examinations in a maximum of 10 subjects.

### 10.2.2 Element Access and Operations on Elements and Entire Arrays

An element of a multidimensional array can be accessed by writing the array name followed by subscript expressions within subscript operators as shown below.

*arr\_name [ expr1 ] [ expr2 ] ... [ exprn ]*

We can perform various operations on the elements of multidimensional arrays, similar to those discussed in one- and two-dimensional arrays. Thus, for an array of a built-in type, we can use an array element in an expression, assign a value to it, perform increment and decrement operations on it, pass it as an argument to a function and so on.

As with one- and two-dimensional arrays, C language does not provide any statement or operator to perform operations on the entire multidimensional array. Nested **for** loops are usually used to perform operations on an entire array as illustrated below for a three-dimensional array **a** of size  $m \times n \times p$ :

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < p; k++) {
            /* process element a[i][j][k] */
        }
    }
}
```

### 10.2.3 Initialization

C language allows the initialization of multidimensional arrays using a syntax similar to that used for initialization of vectors and matrices. First, we can initialize them using a single comma separated list enclosed in braces as follows:

```
data_type arr_name[size1] [size2] ... [size_n] = {expr1, expr2, ...};
```

As with vectors and matrices, the initialization list may be empty causing all the array elements to be initialized to default values which is 0 for arithmetic types.

We can also use a nested syntax similar to that used for the initialization of a matrix. For example, a three-dimensional array of size  $2 \times 3 \times 4$  can be initialized as shown below.

```
int a[2][3][4] = {
    {
        {100, 101, 102, 103},
        {110, 111, 112, 113},
        {120, 121, 122, 123}
    },
    {
        {200, 201, 202, 203},
        {210, 211, 212, 213},
        {220, 221, 222, 223}
    }
};
```

Note that we can omit the first array dimension (2) in the above initialization. Also, we can specify fewer elements for any row or fewer rows for any matrix. In such cases, the remaining elements in that row or matrix will be initialized to default values. Consider for example, the initialization statement given below.

```
int b[2][3][4] = { { {1, 2}, {3, 4, 5} } };
```

This statement initializes array **b** as shown below.

<table border="1"><tr><td>1</td><td>2</td><td>0</td><td>0</td></tr><tr><td>3</td><td>4</td><td>5</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	0	0	3	4	5	0	0	0	0	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	0
1	2	0	0																						
3	4	5	0																						
0	0	0	0																						
0	0	0	0																						
0	0	0	0																						
0	0	0	0																						
Matrix b[0]	Matrix b[1]																								

#### 10.2.4 Multidimensional Arrays as Function Parameters

C language allows multidimensional arrays to be passed as actual arguments to a function. Most of the discussion about two-dimensional arrays in the previous section is applicable to multidimensional arrays as well. Thus, while declaring a multidimensional array as a function parameter, we may omit the

first array dimension only. All the remaining array dimensions must be specified. In the calling functions, the size of the argument array must match these specified array dimensions.

Note that as with vectors and matrices, we can pass the actual size of the array to be processed using additional parameters to this function. Also, recall that all the arrays including multidimensional arrays are passed by reference.

Consider the function definition given below that accepts an integer array of size  $M \times 10 \times 10$  and three parameters  $m$ ,  $n$  and  $p$  that specify the actual size of the array to be processed.

```
void func(int arr[][10][10], int m, int n, int p)
{
    /* process elements of array arr */
}
```

Some calls to this function are shown below.

```
int main()
{
    int a[10][10][10], b[5][10][10], c[50][10][10], d[10][10][5];

    func(a, 10, 10, 10);      /* ok */
    func(b, 3, 4, 5);        /* ok */
    func(c, 25, 10, 8);     /* ok */
    func(d, 5, 5, 5);       /* wrong: dimension of d should be [][][] */
    func(b, 10, 4, 5);      /* wrong: max val of 1st dim for b should be 5 */
}
```

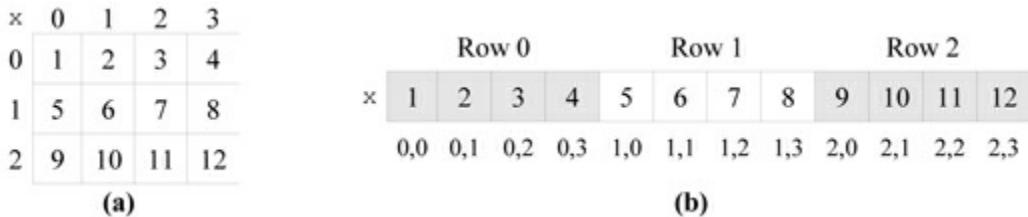
## 10.3 Advanced Concepts

### 10.3.1 const, static and extern Arrays

In last chapter, we have studied the concepts of `const`, `static` and `extern` arrays with reference to one-dimensional arrays or vectors. These concepts are applicable to matrices and multidimensional arrays as well.

### 10.3.2 Memory Allocation

C language uses **row-major allocation** for matrices and multidimensional arrays. In this allocation method, the elements of a matrix are stored row-by row. For example, consider a  $3 \times 4$  integer matrix  $X$  and its memory allocation shown in Fig. 10.2. Remember that each element requires either two or four bytes of memory.



**Fig. 10.2** Memory allocation for a matrix. (a) a  $3 \times 4$  matrix (b) its memory allocation

Assuming that the array in Fig. 10.2 is stored starting from memory location  $0x1000$  and the `int` type requires four bytes, the element  $X[1][3]$  is stored at memory location  $0x101B$ .

Now consider a three-dimensional array named `arr` of size  $2 \times 3 \times 4$ . In this, the memory for the  $3 \times 4$  matrix `arr[0]` is first allocated as explained above followed by memory for matrix `arr[1]`.

Knowing how memory is allocated to array elements is essential while processing these arrays using pointers.

### 10.3.3 Eliminate the Row and Column of a Square Matrix

Consider a square matrix  $A$  of size  $n$ . First minor  $M_{ij}$  of matrix  $A$  is the determinant of a square matrix of size  $n - 1$  obtained from matrix  $A$  by eliminating its  $i$ th row and  $j$ th column. The minors are useful in calculating determinant and inverse of a square matrix.

The function `fmat_elim` given below obtains a matrix `b` by eliminating the specified row and column from matrix `a` of size  $n$ .

```
/* obtain matrix b by eliminating a specified row and col
   from matrix a of size n */
void fmat_elim(float a[][10], int n, float b[][10], int row, int col)
{
    int i, j;      /* indices in matrix a */
    int r, c;      /* indices in matrix b */

    for (i = 0, r = 0; i < n; i++) {
        if (i == row)
            continue;

        for (j = 0, c = 0; j < n; j++) {
            if (j == col)
                continue;
            b[r][c++] = a[i][j];
        }
        r++;
    }
}
```

```
    }  
}
```

#### 10.3.4 Determinant of a Square Matrix

To understand how to calculate the **determinant** of a square matrix, consider a square matrix  $A$  of size 2 and its determinant, denoted as  $|A|$ , given below.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad |A| = ad - bc$$

Similarly, consider a square matrix  $B$  of size 3 and its determinant  $|B|$  given below:

$$B = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad |B| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

We can write functions to calculate these determinants in a straight-forward manner. However, it is more interesting and challenging to calculate the determinant of a square matrix of any given size, say  $n$ . This can be calculated using **Laplace's formula** which expresses the determinant of a matrix in terms of its first minors as shown below.

$$|A| = \sum (-1)^{i+j} a_{ij} M_{ij}$$

where first minor  $M_{ij}$  is the determinant of a square matrix of size  $n - 1$  obtained from matrix  $A$  by eliminating its  $i$  th row and  $j$  th column. Thus, the calculation of the determinant is expressed recursively. A recursive function to calculate the determinant of a square matrix is given below.

```
/* calculate determine of a square matrix a of size n */  
float fmat_det(float a[][10], int n)  
{  
    if (n == 1)  
        return a[0][0];  
    else if (n == 2)  
        return a[0][0] * a[1][1] - a[1][0] * a[0][1];  
    else {  
        float b[10][10], sum;  
        int c;  
        sum = 0;  
        for (c = 0; c < n; c++) {  
            fmat_elim(a, n, b, 0, c);  
            sum += pow(-1, c) * a[0][c] * fmat_det(b, n-1);  
        }  
        return sum;  
    }  
}
```

Observe that for terminating conditions, `n == 1` and `n == 2`, we directly return the value of the determinant. However, for a larger matrix, we need to set up a loop to calculate the desired determinant. In this loop, we first use the `fmat_elim` function to obtain matrix `b` by eliminating row 0 and column `c` from matrix `a`. We then calculate the determinant of matrix `b` using a recursive call to the `fmat_det` function, multiply it with `pow(-1, c)` and add it to `sum`. This sum, which is the determinant of the given matrix is then returned.

Also note that while calculating the determinant of a matrix of size 2 or more, the terminating condition `n == 1` will always evaluate as false. However, it is required in case we call this function for a matrix of size 1.

### 10.3.5 Cofactor of a Matrix

The **cofactor** of the  $ij$ th element of a square matrix  $A$  is defined as

$$C_{ij} = (-1)^{i+j} M_{ij}$$

As mentioned earlier,  $M_{ij}$  is the first minor, i. e., the determinant of a square matrix of size  $n - 1$  obtained from matrix  $A$  by eliminating its  $i$ th row and  $j$ th column. The function given below calculates the cofactor of a specified element (`r, c`) of square matrix `a` of size `n`. It first obtains matrix `b` by eliminating row `r` and column `c` from matrix `a` and then returns the determinant of this matrix (i. e.,  $M_{ij}$ ) multiplied by  $(-1)^{i+j}$ .

```
/* calculate cofactor of a given matrix element */
float fmat_cofactor(float a[][10], int n, int r, int c)
{
    float b[10][10];
    fmat_elim(a, n, b, r, c);
    return pow(-1, r+c) * fmat_det(b, n-1);
}
```

Now we can rewrite the equation to calculate the determinant of a matrix in terms of its cofactors as shown below.

$$|A| = \sum a_{ij} C_{ij}$$

Thus, the function `fmat_det`, to calculate the determinant, can be rewritten concisely using function `fmat_cofactor` as shown below.

```
/* calculate determine of a square matrix a of size n */
float fmat_det(float a[][10], int n)
{
    if (n == 1)
        return a[0][0];
```

```

    else if (n == 2)
        return a[0][0] * a[1][1] - a[1][0] * a[0][1];
    else {
        int c;
        float sum = 0;
        for (c = 0; c < n; c++)
            sum += a[0][c] * fmat_cofactor(a, n, 0, c);
        return sum;
    }
}

```

Observe that the `fmat_det` function calls the `fmat_cofactor` function which in turn calls the `fmat_det` function again. This form of recursion is called **mutual recursion**.

### 10.3.6 Inverse of a Matrix

The **inverse** of a square matrix is defined as shown below.

$$A^{-1} = \frac{1}{|A|} \text{Adj}(A)$$

where  $\text{Adj}(A)$  is the **adjoint matrix** which is a transpose of the cofactor matrix. Note that the inverse can not be computed if the given matrix is singular, i. e., if  $|A|$  is zero.

The function `fmat_inv` to determine the inverse of a square matrix is given below.

```

/* determine inverse of a matrix */
int fmat_inv(float a[][10], int n, float b[][10])
{
    int i, j;
    float d = fmat_det(a, n);
    if (d == 0) /* matrix is singular, inversion not possible */
        return 0;

    /* inverse is a transpose of cofactor matrix */
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            b[j][i] = (float) fmat_cofactor(a, n, i, j) / d;
    }

    return 1;
}

```

This function first calculates the determinant of given matrix `a` of size `n`. If the determinant is 0, it returns a zero value, indicating its inability to determine the inverse; otherwise, it uses a nested `for`

loop to determine the inverse matrix. Note that the cofactor of element  $ij$  is divided by  $|A|$  and written at the  $ji$ -th position in the inverse matrix (as the adjoint matrix is a transpose of the cofactor matrix). The function then returns 1, indicating success.

A complete program to determine the inverse of a square matrix is given below.

```
/* Determine inverse of a square matrix of size n */
#include <stdio.h>

int fmat_inv(float a[][10], int n, float b[][10]);
float fmat_cofactor(float a[][10], int n, int r, int c);
float fmat_det(float a[][10], int n);
void fmat_elim(float a[][10], int n, float b[][10], int row, int col)
void fmat_print(const char *msg, float a[][10], int m, int n);

int main()
{
    float a[10][10] = {
        {5, 2, 3, 4},
        {4, 3, 5, 7},
        {2, 3, 4, 5},
        {2, 4, 4, 7}
    };
    float b[10][10];
    int flag;

    fmat_print("Given matrix", a, 4, 4);
    flag = fmat_inv(a, 4, b);
    if (flag)
        fmat_print("\nInverse matrix", b, 4, 4);
    else printf("\nMatrix a is singular. Inverse doesn't exist.");
}

/* include definitions of functions fmat_inv, fmat_cofactor, fmat_det
fmat_elim */

/* print a matrix */
void fmat_print(const char *msg, float a[][10], int m, int n)
{
    int i, j;

    puts(msg);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
```

```

        printf("%6.2f ", a[i][j]);
        printf("\n");
    }
}

```

The program output is given below.

**Given matrix**

5.00	2.00	3.00	4.00
4.00	3.00	5.00	7.00
2.00	3.00	4.00	5.00
2.00	4.00	4.00	7.00

**Inverse matrix**

0.33	-0.11	-0.11	0.00
0.44	-1.04	0.63	0.33
-0.22	0.19	0.85	-0.67
-0.22	0.52	-0.81	0.33

## Exercises

1. Answer the following questions:
  - a. How can you access the last element of a matrix `a` having  $5 \times 5$  elements?
  - b. How many values will be initialized from a three-dimensional array `x` of size  $10 \times 10 \times 10$  if the initializer list contains 100 elements?
  - c. Which parameter passing method is used to pass an integer matrix to a function?
  - d. What is the meaning of the expression `elem = a[i++][++j]`?
  - e. What is the initial value assigned to each element of an integer matrix declared inside a function if we do not explicitly initialize it?
  - f. If a function parameter is declared as `int a[10][10]`, is it possible to pass an array of size  $15 \times 10$  as an actual parameter?
  - g. If a function parameter is declared as `float a[][10]`, is it possible to pass an array of size  $5 \times 5$  as an actual parameter?
2. Define functions to perform following operations on a matrix of numbers:
  - a. Test whether all elements in a specified row of a matrix are zero or not.
  - b. Test whether a given matrix is a unit matrix or not.
  - c. Test whether a given square matrix is symmetrical about the leading diagonal or not.

- d. Determine in-place transpose of a given square matrix.
  - e. Return two vectors mi n and max that contain minimum and maximum values in each row.
  - f. Perform addition of two matrices and return an addition matrix.
  - g. Perform multiplication of two matrices and return multiplication matrix.
3. Determine errors in the following program segments:

```

a. int m = 3, n = 3;
int a[m][n];
...
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        a[i][j] = 0;

b. #define m 3
int a[m][m];
...
for (i = 0; i < m; i++)
    for (j = 0; j < m; j++)
        if (i + j < m)
            aij = 0;
        else aij = 1;

c. int x[3][4][5];
for (i = 0; i < 3; i++)
    for (j = 0; j < 4; j++)
        for (k = 0; k < 5; k++)
            if (i == j && j == k)
                x[i, j, k] = 1;
            else x[i, j, k] = 0;

d. /* print a matrix of size m x n
 */
void print(float a[10][], int m,
int n)
{
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            printf("%f ", a[i][j]);
    printf("\n");
}

e. /* return unit matrix of size m */
int [10][10] unit_mat(int m)

```

```

{
    int arr[10][10];
    int r, c;
    for (r = 0; r < m; r++)
        for(r = 0; r < m; r++)
            arr[r][c] = (r == c ? 1 : 0);
    return arr;
}

f. /* addition of two matrices of size m x n */
void add_mat(float a[m][n], float b[m][n], float c[m][n])
{
    int row, col;
    for (row = 0; row < m; row++)
        for(col = 0; col < n; col++)
            c[row][col] = a[row][col] + b[row][col];
}

g. /* read a matrix of specified size */
void read_matrix(float mat[][], int cols, int rows)
{
    int c, r;
    for(r = 0; r < rows; r++)
        for(c = 0; c < cols; c++)
            scanf("%f", mat[row][col]);
}

```

- l. Write a complete C program that uses a three-dimensional array to store daily maximum temperatures in four different cities and print a matrix that shows the average monthly temperatures of each city.

## Exercises (Advanced Concepts)

1. Answer the following questions:
  - a. How much memory is required for a matrix of type `char` having  $10 \times 10$  elements?
  - b. What is the address of array element `a[4][3]` of type `double` if the array size is  $5 \times 7$  and the address of element `a[0][0]` is `0x1000`.
  - c. What is mutual recursion?
  - d. Define the terms *minor* and *cofactor*.
2. Define a function to test whether a given matrix is singular or not.
3. Write a program to determine the rank of a matrix.

# 11 Pointers

This chapter covers **pointers**, a very powerful feature of the C programming language. First, the pointer basics are covered. Then the use of pointers with functions is presented which includes the call by reference. The use of pointers with vectors, i. e., one-dimensional arrays, is explained in the next section. It explores the relationship between an array and a pointer and explains how pointers can be used to manipulate arrays and pass them to a function.

The *Advanced Concepts* section covers several topics that include the typecast and `sizeof` operator, returning a pointer from a function, matrices and multidimensional arrays and pointers, pointers to a pointer, array of pointers, dynamic memory allocation, dynamic arrays, pointers to functions, polymorphic functions using the void pointer and complex declarations involving pointers. A beginner can skip this section and return to it after reading subsequent chapters.

## 11.1 Pointer Basics

### 11.1.1 What is a Pointer?

A **pointer** is a very powerful and sophisticated feature provided in the C language. A pointer to a data item is nothing but the address of that data item. We can store this address in a variable, called the **pointer variable**, and use it to manipulate that data item.

Fig 11.1 illustrates the concept of a pointer. Fig 11.1a depicts two variables: an ordinary variable `x` of type `int` stored in memory location `0x1234` and a pointer variable `px`. The pointer variable `px` holds the address of variable `x` (which is an integer value) and is said to point to variable `x`. The type of this pointer variable is `int *`, i. e., a pointer to `int`. Although we can use any name for a pointer variable, the name `px` (which indicates pointer to `x`) is more meaningful. we can use any name for a pointer variable, the name `px` (which indicates pointer to `x`) is more meaningful.



Fig. 11.1 A pointer variable

It may be noted that as a C programmer, we do not need to know the actual address stored in a

pointer variable in most situations. Moreover, the pointer variable may physically be present anywhere in main memory, either before the variable it points to or after it. Hence, an arrow is used as shown in Fig 11.1b to show that a pointer variable points to a variable, i. e., it holds the address of that variable.

Note that Fig 11.1 uses an integer variable `x` for the purpose of illustration. However, we can have a pointer pointing to variables of any type. The type of pointer which points to a variable of type `T` is `T *`. However, note that irrespective of the type of a pointer variable, it holds an integer value, the memory address of the variable it points to. Thus, in a particular environment, a pointer variable usually requires the same amount of memory irrespective of the type of data item it points to. This is usually two bytes on C compilers running on 16-bit operating systems such as MS DOS, e. g., Turbo C/C++<sup>1</sup> and four bytes on 32-bit operating systems such as MS Windows and Linux, e. g., Dev-C++, Code::Blocks, GNU C/C++, etc.

As we will see shortly, we can manipulate the value of variable `x`, shown in Fig 11.1 using pointer variable `px` and this is a very powerful feature, although it does not appear to be. After all, why should we manipulate variable `x` using pointer `px` when we can directly manipulate it? There are some compelling situations where we cannot perform such manipulations. For example, if we are required to manipulate, from within a function, the data items local to some other function, we must use pointers.

It may be noted that during its lifetime, a pointer variable need not always point to the same variable. We can manipulate the pointer itself so that it points to other variables. This is particularly useful while working with arrays. For example, the `++` operator increments the address stored in a pointer such that it points to the next element in that array. Similarly, the `--` operator decrements a pointer so that it points to the previous array element. The use of pointers with arrays often leads to concise and efficient code.

There are several advantages in using pointers, some of which are listed below.

1. Pointers enable us to use **call by reference** mechanism. This enables changes to the formal parameters within a function to be reflected in arguments in the function call. Thus, the modified values are passed back to the calling function.
2. The use of pointers for manipulations of arrays and strings often leads to **concise and efficient programs**.
3. Using pointers, we can construct **advanced data structures** such as linked lists, trees, graphs, etc. to store and manipulate complex data. The use of such advanced data structures often leads to efficient programs.
4. Pointers permit more efficient use of memory, which is a very valuable and limited resource, using a technique called **dynamic memory management**.
5. Void pointers and pointers to functions enable us to write powerful **generic functions** that work with different data types.
6. Pointers also enable the **command-line arguments** to be passed to a program.

### 11.1.2 Declaring Pointer Variables

As already mentioned, the C language permits a pointer to be declared for any data type. The **declaration** of a pointer variable takes the following general form:

```
type *ptr_var;
```

where *type* is a valid C data type and *ptr\_yar* is the name of the pointer variable. We can read this declaration backwards as *ptr\_yar* is a pointer to type *type*. Another interpretation of this declaration is that *\*ptr\_var* has a value of type *type*, the value of the variable to which pointer *ptr\_var* will point. Thus, we can use the expression *\*ptr\_var* in place of the variable pointed by *ptr\_var*. The operator *\** is called as the **dereference operator** (note that the symbol is same as that of the multiplication operator) and is discussed shortly.

The declaration given above reserves memory for pointer variable *ptr\_var*. As we know, the C language does not automatically initialize the value of a local variable. Thus, variable *ptr\_var* will contain some garbage value, i. e., it will point to some arbitrary memory location and not to any specific variable. Such a pointer is called a **stray pointer**. Also note that a pointer variable having zero value does not point anywhere. Such a pointer is called a **null pointer**. Hence, we must initialize pointer *ptr\_var* to point to the desired variable before we use it, as explained shortly.

As with simple variables, we can declare multiple pointer variables of the same type in a single declaration statement using the following form:

```
type *ptr_var1, *ptr_var2, *ptr_var3, ...;
```

Here, each variable is declared as a pointer by prefixing it with the dereferencing operator. Also note that although we can mix the declaration of simple variables and pointers in a single declaration statement, it should be avoided to keep the code readable.

### Example 11.1 Declaring pointer variables

Consider the following example which declares several pointer variables.

```
char *pc;
int *pi;
double *pd;
unsigned long *pul;
```

Here, *pc*, *pi*, *pd* and *pul* are declared as pointers to *char*, *int*, *double* and *unsigned long*, respectively. The types of these pointers are *char \**, *int \**, *double \** and *unsigned long \**, respectively. These pointers have not yet been initialized. Thus, they are stray pointers.

Now consider an example in which we declare multiple pointers in each statement.

```
char *pc1, *pc2;
int *pi1, *pi2, *pi3;
```

This example declares variables `pc1` and `pc2` as pointers to `char` type and variables `pi1`, `pi2` and `pi3` as pointers to `int` type.

Finally, consider the following declarations in which the pointer declarations are mixed with declarations and initializations of simple variables.

```
float x, *px, y = 1.2, *py;
short a = 100, b, *pa, *pb;
```

Note that although these declarations are concise requiring only two lines, they are somewhat difficult to read. This style is not recommended.

Some authors suggest a slightly different style for declaration of a pointer variable in which the dereference operator is written immediately after the data type as shown below.

```
type* ptr_var;
```

Since C is a free-format language, this style is equivalent to the previous one. However, if we declare multiple pointer variables in the same declaration, this style can lead to errors. Consider the following example:

```
int* pa, pb, pc;
```

We may interpret this as the variables `pa`, `pb` and `pc` are declared as pointers to type `int`. However, this is not the case. You will be surprised to know that only variable `pa` is declared as a pointer, whereas `pb` and `pc` are simple variables of type `int`. Hence, this declaration style is discouraged. However, if you wish to use it, declare only one variable per declaration as insisted on by authors recommending this style.

### 11.1.3 Address Operator (&) and Dereference Operator (\*)

To manipulate data using pointers, the C language provides two operators: *address* (`&`) and *dereference* (`*`). These are unary prefix operators. Their precedence is the same as other unary operators which is higher than multiplicative operators.

The **address operator** (`&`) can be used with an lvalue<sup>2</sup>, such as a variable, as in `&var`. This expression yields the address of variable `var`, i.e., a pointer to it. Note that the address operator cannot be used with constants and non-lvalue expressions. Thus, the expressions `&100`, `&(a+5)` and `&'x'` are invalid. If the type of variable `var` is `T`, the expression `&var` returns a value of type `T *`, i.e., a pointer to type `T`. As mentioned earlier, we need not know the exact address where a particular variable is stored in memory. We can use the address operator to obtain its address, whatever it may be. This address can be assigned to a pointer variable of appropriate type so that the pointer points to that variable.

The **dereference operator** (`*`) is a unary prefix operator that can be used with any pointer variable, as

in `*ptr_var`. This expression yields the value of the variable pointed at by that pointer. Although the symbol for the dereference operator is the same as that of the multiplication operator, its use is unambiguous from the context as the latter is a binary infix operator used, as in `expr1 * expr2`.

#### 11.1.4 Pointer Assignment and Initialization

When we declare a pointer, it does not point to any specific variable. We must **initialize** it to point to the desired variable. This is achieved by assigning the address of that variable to the pointer variable, as shown below.

```
int a = 10;
int *pa;
pa = &a;           /* pointer variable pa now points to variable a */
```

In this example, the first line declares an `int` variable named `a` and initializes it to 10. The second line declares a pointer `pa` of type *pointer to int*. Finally, the address of variable `a` is **assigned** to `pa`. Now `pa` is said to point to variable `a`.

We can also initialize a pointer when it is declared using the format given below.

```
type *ptr_var = init_expr;
```

where `init_expr` is an expression that specifies the address of a previously defined variable of appropriate type or it can be `NULL`, a constant defined in the `<stdio.h>` header file. Consider the example given below.

```
float x = 0.5;
float *px = &x;
int *p = NULL;
```

The second line declares a pointer variable `px` of type `float *` and initializes it with the address of variable `x` declared in the first line. Thus, pointer `px` now points to variable `x`. The third line declares pointer variable `p` of type `int *` and initializes it to `NULL`. Thus, pointer `p` does not point to any variable and it is an error to dereference such a pointer.

Note that a character pointer can be initialized using a character string constant as in

```
char *msg = "Hello, world!";
```

Here, the C compiler allocates the required memory for the string constant (14 characters, in the above example, including the null terminator), stores the string constant in this memory and then assigns the initial address of this memory to pointer `msg`, as illustrated in Fig 11.2. We will study the operations on strings using pointers in next chapter. However, note that most of the discussion on pointers to arrays covered in Section 11.3 is applicable to string operations, except that the strings are null terminated.



**Fig. 11.2** A pointer variable pointing to a string constant

The C language also permits initialization of more than one pointer variable in a single statement using the format shown below.

```
type *ptr_var1 = init_expr1, *ptr_var2 = init_expr2,...;
```

It is also possible to mix the declaration and initialization of ordinary variables and pointers. However, we should avoid it to maintain program readability.

#### Example 11.2 Pointer assignment and initialization

Consider the example given below.

```
char a = 'A';
char *pa = &a;
printf("The address of character variable a: %p\n", pa);
printf("The address of pointer variable pa : %p\n", &pa);
printf("The value pointed by pointer variable pa: %c\n", *pa);
```

Here, **pa** is a character pointer variable that is initialized with the address of character variable **a** defined in the first line. Thus, **pa** points to variable **a**. The first two **printf** statements print the address of variables **a** and **pa** using the **%p** ( **p** for pointer) conversion. The last **printf** statement prints the value of **a** using the pointer variable **pa**. When the program containing this code is executed in Code::Blocks, the output is displayed as shown below.

```
The address of character variable a: 0022FF1F
The address of pointer variable pa : 0022FF18
The value pointed by pointer variable pa: A
```

Note that the addresses displayed in the output will usually be different depending on other variables declared in the program and the compiler/IDE used.

Another example is given below in which pointers are initialized with the addresses of variables of incompatible type.

```
char c = 'Z';
int i = 10;
float f = 1.1;
char *pc1 = &i, *pc2 = &f;
int *pi1 = &c, *pi2 = &f;
```

```

float *pf1 = &c, *pf2 = &i;

printf("Character: %c %c\n", *pc1, *pc2);
printf("Integer : %d %d\n", *pi1, *pi2);
printf("Float : %f %f\n", *pf1, *pf2);

```

Note that the character pointer variables `pc1` and `pc2` are initialized with the addresses of the `int` and `float` variables, respectively. Similarly, the `int` and `float` pointer variables are also initialized with addresses of variables of incompatible type. When the program containing this code is compiled in Code::Blocks, the compiler reports six warning messages (*initialization from incompatible pointer type*), one for each incompatible pointer initialization.

It is not a good idea to ignore such warnings associated with pointers. Although, the program executes in the presence of these warnings, it displays wrong results as shown below.

```

Character:
=
Integer : 90 1066192077
Float    : 0.000000 0.000000

```

### 11.1.5 Simple Expressions Involving Pointers

Once a pointer variable, say `ptr_var`, is initialized to point to another variable of appropriate type, we can manipulate the latter using the expression `*ptr_var`. Consider the following code that initializes a pointer `pd`, of type `double *`, to point to a variable `d` of type `double`.

```

double d = 1.0;
double *pd;
pd = &d;

```

Now we can manipulate the value of variable `d` using the pointer variable `pd`. For example, we can assign another value to `d` as shown below.

```
*pd = 2.0;
```

Since the expression `*pd` yields a value of type `double`, we can use it in any expression where `d` can be used as illustrated in the expressions given below.

<code>x = *pd;</code>	<code>/* assign value of d to x */</code>
<code>y = *pd + 10.0;</code>	<code>/* assign d + 10.0 to y */</code>
<code>*pd = *pd + 5.0;</code>	<code>/* add 5.0 to d */</code>
<code>*pd *= 10;</code>	<code>/* multiply the value of d by 10 */</code>
<code>++*pd;</code>	<code>/* increment the value of d by 1 */</code>
<code>a = *pd++;</code>	<code>/* assign value of d to a and increment pd */</code>

```
*pd >= 10.0           /* compare value of d with 10.0 */
```

As the dereference operator has higher precedence than binary arithmetic operators (\*, /, +, -) and relational and equality operators, we will not have any difficulty while mixing the dereference operator with such operators. Thus, the following expressions are equivalent.

```
y = *pd + 10.0;  
y = (*pd) + 10.0;
```

We can also use pointer dereference expressions in function calls as shown below.

```
int a = 100;  
int *pa = &a;  
printf("Value of a = %d\n", *pa);  
some_func(*pa);
```

The second line initializes pointer `pa` to point to variable `a`. Then expression `*pa` is used as an argument to the `printf` and `some_func` functions in lines 3 and 4, respectively.

## 11.2 Call by Reference

We know that when a function is called, the parameters are passed to it by value, i. e., the values of arguments in a function call are copied to the parameters of the called function. Since a function parameter is a copy of the argument variable and is local to the function, any change in its value in the body of the function modifies only the local copy and not the corresponding argument in the calling function.

We may often come across situations where we need to write functions that should modify the values of argument variables. For example, consider that we wish to write a function to modify the value of a variable or exchange the values of two variables. The first problem involves modification of a single variable and can be easily solved using a function that returns a value which can be assigned to that variable as shown below.

```
a = next_val(a);
```

However, the second problem is more difficult as it involves the modification of two variables. We cannot modify two (or more) variables in the calling function as the function returns only one value (and by default, the function parameters are passed by value, i. e., they are input parameters). Moreover, since the argument variables are in a different function (i. e., outside the scope of the called function), they cannot be directly accessed or modified by the called function. Even declaring these variables to be modified as global variables (so that they can be accessed from any part of the program) is not really useful as the function cannot be used to modify different variables each time it is called (as expected in the case of a function to exchange two values).

The solution to this problem is the **call by reference** mechanism in which pointer variables are used as function parameters. Thus, the prototype of the **swap** function, used to exchange the values of two variables, takes the following form:

```
void swap(int *, int *);
```

When this function is called, the addresses of the variables to be modified are passed as arguments to the pointer parameters. Thus, to exchange the values of variables **a** and **b**, this function is called as follows:

```
swap(&a, &b);
```

Note that since the address operator cannot be used with constants and expressions (non-lvalue), we cannot use **swap** function to exchange values of constants and such expressions.

The **swap** function uses the call by reference mechanism for both parameters. However, it is possible to mix the call by value and call by reference mechanisms in a single function.

Another situation in which call by reference is useful is when we need to return two or more values from a function, e.g., a function to return both area and perimeter of a circle. The call by reference mechanism is also very useful when passing large structures to a function and may significantly improve program efficiency.

Can you now figure out why we need to pass the addresses of variables to be read from the keyboard in a call to the **scanf** function? Actually, when **scanf** is called, it has to store the values entered from the keyboard in argument variables that we pass to it. Thus, the **scanf** function has to modify the actual arguments through its parameters. As this is possible by using pointers, the **scanf** function requires the addresses of the variables.

### Example 11.3 Functions using call by reference

#### a) Function to exchange the values of variables

Consider the function **bad\_swap** given below that attempts to exchange the values of two variables passed as its arguments.

```
/* Exchange two integers: INCORRECT FUNCTION */
void bad_swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

This function accepts two parameters, `x` and `y` of type `int`. It uses the usual three step procedure to exchange the values of these parameters. This function can be called from some other function, e. g., `main`, as shown below.

```
int main()
{
    int a = 10, b = 20;
    printf("Before exchange a = %d b = %d\n", a, b);
    bad_swap(a, b);
    printf("After exchange a = %d b = %d\n", a, b);
}
```

When the `bad_swap` function is called, the values of argument variables `a` and `b` are passed to it, i. e., they are copied to function parameters `x` and `y`. The `bad_swap` function correctly exchanges the values of `x` and `y` as may be verified by inserting a `printf` statement after the exchange operation. However, the values of variables `a` and `b` in the `main` function are not modified at all. Thus, the `bad_swap` function does not exchange the values of the variables passed as arguments which is evident from the output given below.

```
Before exchange a = 10 b = 20
After exchange a = 10 b = 20
```

Consider the correct implementation given below that uses the call by reference mechanism.

```
/* Exchange two integers */
void swap(int *px, int *py)
{
    int temp = *px;
    *px = *py;
    *py = temp;
}
```

Note that this function has two reference parameters, `px` and `py`, which are pointers to type `int`. As we know, the expressions `*px` and `*py` represent `int` values, the values of the argument variables passed to this function. The body of the `swap` function uses the three-step procedure to exchange the values pointed to by `px` and `py`.

The `main` function which calls the above `swap` function to exchange the values of two variables is given below.

```
int main()
{
    int a = 10, b = 20;
```

```

printf("Before exchange a = %d b = %d\n", a, b);
swap(&a, &b);
printf("After exchange a = %d b = %d\n", a, b);
}

```

The output of this function is given below.

Before exchange a = 10	b = 20
After exchange a = 20	b = 10

The values of **a** and **b** are correctly exchanged now. When the **swap** function is called, the addresses of variables **a** and **b** are passed (i. e., copied) to parameters **px** and **py**, respectively. Thus, inside the **swap** function, the parameters **px** and **py** point to variables **a** and **b**. Exchanging the values of **\*px** and **\*py** thus actually exchange the values of arguments **a** and **b**.

We can call the **Swap** function to exchange different pairs of **int** variables as illustrated below.

```

swap(&x, &y);          /* variables x and y */
swap(&a[i], &a[j]);    /* array elements a[i] and a[j] */
swap(&a[0][0], &x);    /* matrix element a[0][0] and variable x */

```

However, the following calls to the **swap** function are invalid as the address operator (**&**) cannot be used with a constant or non-lvalue expressions.

```

swap(&10, &20);
swap(&(a*b), &(a/b));
swap(&a++, &b--);
swap(&-x, &+y);

```

Note that the **swap** function given above exchanges the values of integer variables only. However, if we wish to exchange the values of variables of some other type, we should suitably modify the **swap** function (just replace the **int** keyword with the desired type). Thus, to exchange the values of two variables of type **float**, we can modify the **swap** function as shown below.

```

/* Exchange two variables of type float */
void swap(float *px, float *py)
{
    float temp = *px;
    *px = *py;
    *py = temp;
}

```

Of course, if we wish to exchange variables of several types (say, **int**, **float**, **char**, **long double**, etc.) in the same program, we can write several **swap** functions, one for each desired type.

and give them suitable names such as `iswap`, `fswap`, `cswap`, `ldswap`, etc.

### b) Reverse a vector

In this example, we will use the `swap` function along with a loop to reverse a vector. A function `ivec_rev` to reverse an integer vector is given below.

```
/* reverse an int vector */
void ivec_rev(int a[], int n)
{
    int i, j;
    for(i = 0, j = n - 1; i < j; i++, j--)
        swap(&a[i], &a[j]);
}
```

### c) Function to return the sum and average of three numbers

In this example, we will study how to return two (or more) values from a function. Consider the function `sum_avg` given below to calculate and return two results, the sum and average of three given numbers.

```
/* calculate sum and average of three numbers
   Uses call by reference to return both the sum and average */
void sum_avg(double x, double y, double z, double *sum, double *avg)
{
    *sum = x + y + z;
    *avg = *sum / 3.0;
}
```

This function uses five parameters: value parameters `x`, `y` and `z`, which represent three numbers to be processed and reference parameters `sum` and `avg`, which are used to return the sum and average of the given numbers. The function modifies the arguments pointed by the parameters `sum` and `avg` using pointer dereference (using `* sum` and `* avg`). Note that since the values are returned using reference parameters, the function does not require a `return` statement.

We can opt to return either of the two values, say sum of three numbers, using the function return value. Such an implementation is given below.

```
/* calculate sum and average of three numbers. Sum is returned using
   function return value and call by reference is used to return average
double sum_avg(double x, double y, double z, double *avg)
{
```

```
double sum = x + y + z;  
*avg = sum / 3;  
return sum;  
}
```

Of course, we can write two separate functions to calculate the sum and average of three numbers. However, it will not be efficient since it will involve the overhead of two function calls instead of one and possibly repetitive calculation of the sum of three numbers.

## 11.3 Vectors and Pointers

We have studied that an array is a powerful built-in data structure in the C language. It is a collection of data items of the same type stored in consecutive memory locations. An element of an array can be accessed using subscript notation, as in `a[i]`, `b[i][j]`, etc. Also, we can process entire arrays using loops and pass them to functions.

The power of arrays is enhanced further by having a close correspondence between arrays and pointers. The name of an array of type T is equivalent to a pointer to type T, whose value is the starting address of that array, i. e., the address of element 0. This correspondence between an array name and a pointer allows us to access array elements using pointer notation in addition to the subscript notation. We can write efficient code that runs faster using pointers for array processing.

In this section, we will study how various operations can be done on vectors (i. e., one dimensional arrays) using pointers. We will first study some operations on pointers that are useful in this section and then examine how to manipulate a vector by treating its name as a pointer as well as a separate pointer. Finally, we will study how a pointer can be used to pass a vector to a function. The operations on matrices and multidimensional arrays using pointers are discussed in Section 11.4.3.

### 11.3.1 Operations with Pointers to Vector Elements

A pointer variable contains the address of the memory location to which it points, i. e., an integer value. Although a large number of operations can be performed on integer values, there are restrictions on the operations that can be performed on pointers. The various operations that can be performed on pointer variables include the following:

1. add an integer to (or subtract an integer from) a pointer
2. increment/decrement a pointer
3. subtract two pointers and
4. compare two pointers.

Note that first two operations are meaningful when pointer points to an array element and last two operations are meaningful when both pointers point to the elements of same array.

## Add (Subtract) an Integer to (from) a Pointer

Consider a pointer `pa` that points to vector element `a[i]`. When an integer `k` is added to this pointer, we get a pointer that points to an element `k` elements past the current pointer position, i. e., `pa + k` points to element `a[i+k]`. We can access this element, without modifying the current pointer position, using the expression `*(pa+k)`. Similarly, `pa - k` points to an element `k` elements before the current pointer position, i. e., `a[i-k]`.

We can also advance a pointer by `k` array elements by using the assignment `pa += k`. However, we have to be careful while performing such operations as the resulting pointer may point to an element beyond the existing array elements, in which case, we should not access that element.

## Subtraction of Pointers

Consider two pointers `pa` and `pb` that point to vector elements `a[i]` and `a[j]`, respectively, where `i ≤ j`. Subtraction of these pointers, `pb - pa`, gives the value `j - i`.

Note that the array name is a pointer to the element at index 0. Thus, if `pa` is a pointer to element `a[i]` then the expression `pa - a` gives `i`, the index of element `a[i]`.

## Increment and Decrement Operations with Pointers

Consider a pointer `pa` pointing to `a[i]`, i. e., `i`th element of a vector `a`. The increment operation (`++pa` or `pa++`) causes a pointer to point to the next array element (and not the next memory byte as you might expect). Similarly, the decrement operation (`-- pa` or `pa--`) causes a pointer to point to the previous array element. Actually, for a pointer of type `T*`, the pointer value is incremented (or decremented) by `sizeof(T)` bytes, which is also the size of each array element. This is true even if the pointer is positioned beyond the array limits or even if it is not pointing to an array element.

The pointer dereference and increment/decrement operations can be combined to obtain three expressions: `++*pa`, `*++pa` and `*pa++`.

In expression `++*pa`, the pointer is dereferenced first, thus causing the value pointed by the pointer to increment (and not the pointer itself). On the other hand, the expression `*++pa` increments the pointer first to point to the next array element and then dereferences it.

In expression `*pa++`, the pointer `pa` is associated first with the increment operator followed by the dereference operator (right-to-left associativity). Thus, the increment operator increments the pointer and not the value pointed by it. However, since the increment operator is used as a postfix operator, the pointer value is used first (to dereference it) and then incremented. Thus, the statement given below first assigns the value pointed by pointer `pa` to variable `x` and then increments `pa`.

```
x = *pa++;
```

We can also use parentheses within these pointer expressions to obtain four more expressions: `++`

`(*pa)`, `(*pa)++`, `*(pa++)` and `*(++pa)`. First note that the following pairs are equivalent: `++(*pa)` and `++*pa`, `*(pa++)` and `*pa++`, `*(++pa)` and `*++pa`. This is obvious as the operator within the parentheses is the one which is associated first with the pointer variable `pa` in absence of parentheses. However, the expression `(*pa)++` is different as the pair of parentheses change the order of operator association. In this expression, the pointer variable is dereferenced first and then this value is incremented after it is used. Thus, the statement given below first assigns the value pointed by pointer `pa` to variable `x` and then increments this value.

```
x = (*pa)++;
```

#### Example 11.4 Using increment and decrement operators with pointers

Read carefully the program given below.

```
#include <stdio.h>

int main()
{
    int a[] = {1, 5, 9};
    int *pa = a; /* pa points to a[0] */
    int x, y, z;

    pa++;           /* pa points to a[1] */
    --pa;           /* pa points to a[0] again */
    x = ++*pa;      /* a[0]=2, x=2, pa still points to a[0] */
    y = *pa++;      /* y=2, pa points to a[1] */
    z = *++pa;      /* pa points to a[2], z=pa[2] i.e. 9, a[1] unchange
                     /* a[2]=10, pa still points to a[2] */
    (*pa)++;        /* a[2]=10, pa still points to a[2] */
    ++(*pa);        /* a[2]=11, pa still points to a[2] */
    pa++;           /* pa points to a[3] */

    printf("a[] = {%d, %d, %d}\n", a[0], a[1], a[2]);
    printf("x=%d y=%d, z=%d\n", x, y, z);
    printf("pa points to element: a[%d]\n", pa-a);
    return 0;
}
```

The program output is given below.

```
a[] = {2, 5, 11}
x=2 y=2, z=9
pa points to element: a[3]
```

## Comparison of Pointers

We can use relational operators ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ) to compare two pointers provided both of them point to elements in the same array. A pointer is smaller (greater) than another pointer, if it points to an element before (after) the element pointed to by another pointer. Thus, if pointers  $pa$  and  $pb$  point to elements  $a[2]$  and  $a[7]$ , respectively, the expressions  $pa < pb$  and  $pb > pa$  evaluate as true. Also, the expressions  $pa \leq pb$  and  $pb \geq pa$  evaluate as true.

Note that a pointer being compared may be beyond the last array element (or before the first element). However, we should not access (non-existent) array elements using such pointers. Such a pointer, beyond the last array element, is usually used as the termination criterion in loops. This will be illustrated shortly with the help of an example.

We can also compare two pointers using the equality operators ( $\==$ ,  $\!=$ ). The pointers are considered equal if they point to the same variable or array element. Note that the pointers being compared for equality need not point to the elements of the same array, as in case of relational operators. Thus, if  $pa$  and  $pb$  are pointers pointing to elements  $a[2]$  and  $a[7]$  as before and  $px1$  and  $px2$  are pointers pointing to integer variable  $x$ , the expressions  $pa \== pb$ ,  $pa \== px1$  and  $px1 \!= px2$  evaluate as false and the expressions  $pa \!= pb$ ,  $pa \!= px1$  and  $px1 \== px2$  evaluate as true.

### 11.3.2 Accessing Vector Elements by Using Array Name as a Pointer

Consider a vector  $a$  of type `int` declared as follows:

```
int a[10];
```

An array name  $a$  is treated as a pointer to its beginning. Thus, the value of  $a$  is the same as  $\&a[0]$  and the expression  $a+i$  is a pointer to  $i$ th element. When dereferenced as  $*(a+i)$ , it gives the value of the  $i$ th element, i. e., the expression  $*(a+i)$  is equivalent to  $a[i]$ . Note that the parentheses in expression  $*(a+i)$  are essential as an addition operator has lower precedence than the dereference operator. The array subscript expressions are summarized in Table 11.1 along with the equivalent pointer expressions.

**Table 11.1** Subscript expressions and equivalent pointer expressions for a vector

Description	Using array subscript	Using pointer
Value of the first array element	$a[0]$	$*a$
Address of the first array element	$\&a[0]$	$a$
Value of the $i$ th array element	$a[i]$	$*(a+i)$
Address of the $i$ th array element	$\&a[i]$	$a+i$

Note that when an array is declared, memory is allocated only for array elements but not for array name. Thus, an array name is not exactly identical to a pointer variable and certain operations that can be performed on pointer variables are not permitted on array names. For example, we cannot perform

assignment and increment/decrement operations on array names. Thus, if **a** and **b** are **int** arrays and **pa** is an **int** pointer, the following operations are not permitted.

```
a = pa; /* wrong: cannot assign/modify address of an array */
a++;    /* wrong: cannot increment/decrement an array (name) */
a = b;  /* wrong: cannot assign an array to another */
```

#### Example 11.5 Vector manipulations performed by treating its name as a pointer

Consider that **a** and **b** are vectors of type **int** and **i** is a variable of type **int** declared as

```
int a[10], b[10];
int i;
```

Several operations on vectors performed by treating their names as pointers are given below.

##### a) Print an array

```
for (i = 0; i < n; i++)
    printf("%d ", *(a+i));
```

This example prints the first *n* elements of array **a**. Note that the expression **\*(a+i)** is used in place of the subscript notation **a[i]**.

##### b) Read an array from the keyboard

```
for (i = 0; i < n; i++)
    scanf("%d", a+i);
```

This example reads first *n* elements of array **a** from the keyboard. Observe that the expression **a+i**, which is the address of the *i*th array element, is used in the **scanf** statement in place of **&a[i]** .

##### c) Copy an array

```
for (i = 0; i < n; i++)
    *(a+i) = *(b+i);
```

This example copies the first *n* elements from array **b** to array **a**.

### 11.3.3 Accessing Vector Elements Using Another Pointer Variable

We can initialize a pointer of appropriate type so as to point to any element of a vector and then manipulate the vector elements using this pointer. Thus, if **a** is an integer vector and **pa** is an integer

pointer, we can initialize pointer **pa** to point to the *i*th element of vector as

```
pa = &a[i];
```

Since the vector name is a pointer to its first element, the pointer **pa** can be concisely initialized to point to the first element **a[0]** as:

```
pa = a;
```

We can also combine pointer declaration and assignment in a single initialization statement as shown below (assuming that vector **a** is already declared).

```
int *pa = a;
```

Once pointer **pa** is initialized, it can be used to access vector elements. Thus, if **pa** is initialized to the first element (at index 0), the expression **pa+i** is the address of *i*th element, i. e., **&a[i]** and the expression **\*(pa+i)** gives the value of the *i*th element, i. e., **a[i]**.

Thus, to manipulate an entire vector using a separate pointer, first initialize a pointer to the first array element and then set up a loop to process each element using the element access expressions, **\*(pa+i)**. This technique is very similar to that used in the previous subsection in which an array name is used as a pointer. However, there is a small difference: the pointer variable can be initialized to any array element, whereas an array name always points to the beginning of the array only.

There is another more convenient and efficient method to manipulate a vector using a pointer variable in which we make the pointer step through the vector elements. In this method, first initialize a pointer to point to the first element and then set up a loop to process each element as before. However, as the loop progresses, the pointer is made to step through the vector elements by using the pointer-increment operation. As the pointer always points to the vector element to be processed, the value of the current element is obtained by dereferencing the pointer, as **\*pa**.

### Example 11.6 Vector manipulations using pointer variables

This example illustrates the two methods mentioned above to manipulate vectors (one-dimensional arrays) using pointer variables. Assume that the arrays and pointer variables are declared as follows:

```
int a[10], b[10], c[10];
int *pa, *pb, *pc;
```

#### a) Print an array

```
pa = a;
for (i = 0; i < n; i++)
    printf("%d ", *(pa+i));
```

In this example, the pointer **pa** is first initialized to the first element of vector **a**. Then a **for** loop is used to print the first *n* elements in the array. Note that the expression **\*(pa+i)** is used in place of the subscript notation **a[i]**.

We can do the same thing by stepping the pointer **pa** to each vector element as shown below.

```
pa = a;
for (i = 0; i < n; i++) {
    printf("%d ", *pa);
    pa++;
}
```

Since the pointer always points to the element to be printed, the value of the *i*th array element is accessed by simply dereferencing the pointer as **\*pa**. After an element is printed, the pointer variable is incremented, so as to point to the next element.

C is a very powerful language and it allows us to write very concise code. In the above code, **pa++** can be combined with the element access expression **\*pa** using the postfix increment operator, as **\*pa++**. This eliminates one line from the body of the loop making the braces redundant. The simplified code is given below.

```
pa = a;
for (i = 0; i < n; i++)
    printf("%d ", *pa++);
```

We can make the code more concise, by including the pointer initialization statement in the initialization expression of the **for** loop, using a comma operator as shown below.

```
for (i = 0, pa = a; i < n; i++)
    printf("%d ", *pa++);
```

Alternatively, the pointer increment can be written in the update expression of the **for** loop using a comma operator as shown below.

```
for (i = 0, pa = a; i < n; i++, pa++)
    printf("%d ", *pa);
```

We can also eliminate the index variable **i** and use pointer **pa** as the loop control variable as shown below.

```
for (pa = a; pa < a + n; pa++)
    printf("%d ", *pa);
```

Note that the expression **a+n** is the address of the element at index *n*, i. e., **&a[n]**. Alternatively, the code can also be written using a **while** loop as shown below.

```
pa = a;  
while (pa < a + n)  
    printf("%d ", *pa++);
```

**b)** Read an array from the keyboard

```
pa = a;  
for (i = 0; i < n; i++)  
    scanf("%d", pa+i);
```

This program segment reads the first  $n$  elements of vector **a** from the keyboard. Observe that the expression **pa+i**, which is the address of the  $i$ th array element, is used in the **scanf** statement in place of **&a[i]**. Alternatively, we can write this code using pointer **pa** to control the loop as shown below.

```
for (pa = a; pa < a + n; pa++)  
    scanf("%d", pa);
```

**c)** Add two arrays

```
pa = a;  
pb = b;  
pc = c;  
for (i = 0; i < n; i++)  
    *(pc + i) = *(pa + i) + *(pb + i);
```

This program segment first initializes pointers **pa**, **pb** and **pc** to point to the beginning of vectors **a**, **b** and **c**, respectively, and then sets up a loop to add vectors **a** and **b** to obtain vector **c**.

We can rewrite this code concisely in a number of ways. One such alternative is given below in which pointer **pa** is used as a control variable of the **for** loop and the pointers are incremented in element access expressions using the postfix increment operator.

```
for(pa = a, pb = b, pc = c; pa < a + n; ) /* update expr omitted */  
    *pc++ = *pa++ + *pb++;
```

Although concise, the programs written in such a coding style can be difficult to read. The use of the **while** loop is better in such situations as it leads to more readable code, as shown below.

```
pa = a, pb = b, pc = c;  
while(pa < a + n)  
    *pc++ = *pa++ + *pb++;
```

#### 11.3.4 Passing a Vector to a Function Using a Pointer

An *array of type T* is analogous to a *pointer to type T*. Thus, we can pass an array to a function using a pointer. Consider the following function prototype.

```
void func(int a[], int n);
```

The declaration of array **a** in this prototype can be equivalently written using a pointer as

```
void func(int *a, int n);
```

Within this function, we can access the array elements using either array notation or pointer notation. Moreover, since parameter **a** is a pointer variable, we can use pointer increment operations to obtain an efficient implementation of the function. Note that a call to this function is exactly same as that of the array version.

#### Example 11.7 Passing a vector to a function using a pointer

Consider the function given below which displays the first *n* elements of a vector of type **int**.

```
void ivec_print(int x[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", x[i]);
}
```

We can modify this function to use a pointer parameter by simply replacing **int x[]** with **int \*x**. However, we can write a more efficient version by treating **x** as a pointer while manipulating the array. Such a function is given below.

```
void ivec_print(int *x, int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", *x++);
}
```

Note that parameter **x** is a pointer variable that points to the argument array. It is local to the function **ivec\_print** and modifications to its value do not affect the argument array in the called function. (Don't get confused here, the value of variable **x** is what is being discussed and not the array elements pointed by **x**. Of course, any change in the values of array elements pointed by parameter **x** will be reflected in the calling function.) Since pointer **x** is incremented in each iteration of the loop, it steps through the elements of the argument vector as discussed in last section.

Further, we can eliminate the index variable `i` and use the pointer variable `x` in its place as illustrated below.

```
void ivect_print(int *x, int n)
{
    int *last = x + n;
    for ( ; x < last; x++)
        printf("%d ", *x);
}
```

Observe the use of a pointer variable `last` to remember the address of the element beyond the last element to be printed. It is essential as the value of pointer variable `x` is changing and we cannot use the expression `x < x + n` as a termination test.

Of course, this code would be more readable if we use a `while` loop as shown below.

```
void ivect_print(int *x, int n)
{
    int *last = x + n;
    while (x < last)
        printf("%d ", *x++);
}
```

## 11.4 Advanced Concepts

This section covers several advanced topics related to pointers. These include the pointer typecast and `sizeof` operator, returning a pointer from a function, use of pointers with matrices and multidimensional arrays, pointer to another pointer, array of pointers, dynamic memory management, pointer to a function, polymorphic functions and complex declarations involving pointers.

### 11.4.1 The Typecast and `sizeof` Operators

We can use **typecast operator** to convert the type of value returned by a pointer dereference expression to another suitable type. For example, if `pa` is a pointer to a variable `a` of type `int`, we can use typecast as `(double) *pa` to convert the integer value of `*pa` to type `double`.

The C language also allows us to use typecast to convert the type of a pointer, as in `(char *) pa`. This converts the pointer `pa`, which is assumed to be a pointer to `int`, to `char *`, i. e., a character pointer. Such typecasts are essential and are commonly used while dealing with *void pointers*. However, it should be used very carefully in other situations as it may lead to erroneous situations. For example, the expression `*pa` interprets the contents of two (or four) consecutive memory locations pointed to by pointer `pa` as an integer, whereas the expression `*((char *) pa)` interprets the contents of only

one byte pointed to by `pa`, i. e., some part of integer value, as a character. Nevertheless, there are situations where this kind of conversion is useful, as illustrated in Example 11.8.

The **sizeof operator**, when used with a pointer variable, as in `sizeof(pa)`, gives us the size of a pointer variable, i. e., memory required for its storage in bytes.

### Example 11.8 Function to print binary representation of numbers

Consider that we wish to print binary representations of numbers of different data types (`int`, `short int`, `float`, `long double`, etc.). One approach to do this is to write a separate function for each data type. However, this is tedious and time-consuming. Instead, we can write a single function named `print_binary` as shown below.

```
void print_binary(unsigned char *pc, int n)
{
    int i;
    for(i = n - 1; i >= 0; i--) /* process bytes from MSB to LSB */
        print_char_bits(pc[i]); /* print a byte in binary */
}
```

To handle different data types and sizes, this function accepts an unsigned character pointer to the variable to be printed and its size in bytes. The `for` loop is set up to process individual bytes starting with the most significant byte (which is the last byte on most computers as the numbers are stored starting with the least significant byte). The bit pattern of each byte is printed by calling the `print_char_bits` function given below.

```
void print_char_bits(unsigned char c)
{
    int i;
    unsigned char mask = 0x80; /* i.e. mask = 10000000 (binary) */
    for(i = 7; i >= 0; i--) {
        printf("%c", (c & mask ? '1' : '0')); /* print bit at position
mask >>= 1; /* shift '1' in mask to right side */
    }
    printf(" ");
}
```

This function accepts a byte as an unsigned character and uses a `for` loop to process individual bits in a character starting with the most significant bit (i. e., bit 7). A mask is used to separate individual bits. It is initialized to `0x80` (i. e.,  $1000\ 0000_2$ ) before the loop and in each iteration, the bit pattern is shifted to the right by one position. The bit-wise and (`&`) operator is used along with the conditional operator to print a bit as

```
printf("%c", (c & mask ? '1' : '0')); /* print bit at position i */
```

Note that to simplify the code, we can eliminate variable **i** and use variable **mask** as the loop variable as shown below.

```
void print_char_bits(unsigned char c)
{
    unsigned char mask;
    for(mask = 0x80; mask > 0; mask >>= 1) /* process bits MSB to LSB
        printf("%c", (c & mask ? '1' : '0')); /* print current bit */
    printf(" ");
}
```

The **main** function used to print the binary representations of two numbers (of type **short int** and **float**) is given below.

```
int main()
{
    short int a = 6;
    float x = 1.23;

    printf("short int : %4hd Binary: ", a);
    print_binary((unsigned char *) &a, sizeof(a));

    printf("\nfloat : %0.2f Binary: ", x);
    print_binary((unsigned char *) &x, sizeof(x));

    return 0;
}
```

Observe how the address of the variable to be printed is typecast to **unsigned char \***. The program output is given below.

```
short int : 6     Binary: 00000000 00000110
float      : 1.23  Binary: 00111111 10011101 01110000 10100100
```

### 11.4.2 Returning a Pointer from a Function

So far we have studied functions that either return a value or have a **void** return type. A function can also return a pointer to a data item of any type. However, we must be careful while returning pointers from a function. A common mistake would be to return a pointer to a local variable or value parameter in that function as they are destroyed when control returns to the calling function.

In situations where we have to return a pointer to a local variable, we must allocate the memory for such variables using dynamic memory management techniques, i. e., using `malloc`, `calloc` and `realloc` standard library functions. Such dynamically allocated variables last till the end of program execution. Thus, the memory allocated using these functions is not destroyed when the control returns to the calling functions. Dynamic memory management is explained in Section 11.4.7.

### Example 11.9 Returning a pointer from a function

Consider that we wish to write a function that accepts two integer numbers and returns a pointer to the smaller number. The function `bad_pmin1` given below has two value parameters (`a` and `b`) and return type of pointer to `int`. Within this function, the smaller number is first determined in variable `small` whose address is then returned as expected. However, this implementation is incorrect as it returns the address of a local variable.

```
int *bad_pmin1(int a, int b) /* INCORRECT FUNCTION */
{
    int small = (a < b) ? a : b;
    return &small; /* Wrong: can't return addr of local variable */
}
```

The function `bad_pmin2` given below is also incorrect as it returns the address of one of the function parameters which, as we know, is local to the function.

```
int *bad_pmin2(int a, int b) /* INCORRECT FUNCTION */
{
    if (a < b)
        return &a; /* Wrong: can't return addr of a param */
    else return &b; /* also wrong */
}
```

The function `pmin` given below is the correct implementation. It accepts two parameters, `pa` and `pb`, which are pointers to `int` and returns the one that points to the smaller number.

```
/* return pointer to smaller value */
int *pmin(int *pa, int *pb)
{
    if (*pa < *pb)
        return pa;
    else return pb;
}
```

Note that although `pa` and `pb` are pointer variables, they are function parameters and are local to the

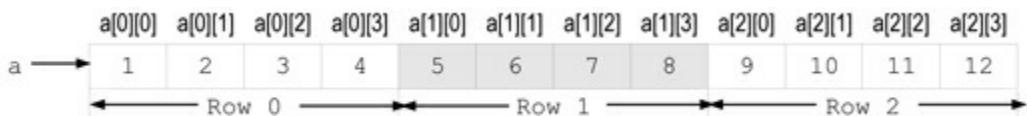
function `pmin`. Thus, returning the address of either `pa` or `pb` (in case we need to return a pointer to a pointer to smaller value) is also incorrect.

### 11.4.3 Matrices and Pointers

Consider an integer matrix declared and initialized as shown below.

```
int a[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
```

All the elements of this matrix are allocated in a contiguous block of memory in a row major order, i. e., row-by-row as illustrated in Fig 11.3.



**Fig. 11.3** Memory allocation for a matrix

As with a vector, the matrix name is a pointer to the first element of the matrix, i. e., element `a[0][0]`. However, note that the matrix name is not a pointer variable and it does not have any memory allocated to it. Thus, the total memory required for a matrix of type `T` having  $m$  rows and  $n$  columns can be calculated as `m * n * sizeof(T)`.

Note that the matrix name is a constant whose value cannot be modified. Thus, the following statements are invalid.

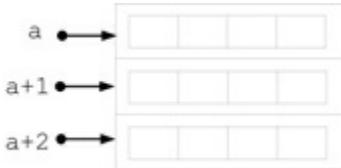
```
a++;  
a += 10;  
a = b; /* b is another matrix */
```

#### Accessing Matrix Elements by Using Array Name as a Pointer

As we know, the expression `x[i]` to access the  $i$ th element of vector `x` can be equivalently written using pointer notation as `*(x+i)`. Thus, the expression `a[i][j]` to access the  $ij$ th element of matrix `a` can be equivalently written as `*(*(a+i)+j)`. Observe that this expression is obtained by applying twice the construct used for a vector<sup>3</sup>. The address of `a[i][j]` is thus given as `*(a+i)+j`. These results are summarized below.

$$\begin{aligned} a[i][j] &\equiv *(*(a+i)+j) \\ \&a[i][j] &\equiv *(a+i)+j \end{aligned}$$

To understand these expressions. Consider again the matrix declared as `int a[3][4]`. Due to the left-to-right associativity of the `[]` operators, this is equivalent to `(a[3])[4]`, i. e., `a` is an array having three elements each of which is also an array having four elements, as depicted in Fig 11.4.



**Fig. 11.4 Matrix as an array of one-dimensional arrays**

The array name is a pointer to the first row and the expression  $a+i$  is a pointer to the  $i$ th row. Dereferencing it as  $*(a+i)$  gives us the element contained in the  $i$ th row, which is an array having 4 elements. As an array (name) is actually a pointer to its beginning,  $*(a+i)$  is obviously a pointer to the first element of this array. Thus, the expression  $*(a+i)+j$  gives a pointer to the  $j$ th element in this matrix (in the  $i$ th row). Thus, dereferencing it as  $*(*(a+i)+j)$  gives the  $ij$ th element, i. e.,  $a[i][j]$ .

Note that the expression  $a+i$  which is a pointer to the  $i$ th row and  $*(a+i)$  which is a pointer to the first element in the  $i$ th row, both point to the same memory location, i. e., they have same value. However, they are different as the first one is a row pointer which, when incremented, advances to the next row, whereas the second one is an element pointer which, when incremented, advances to the next element in that row.

If the element being accessed is in the first row and/or first column, we can rewrite the element access expression as shown below.

$**a$	Element in first row, first column
$**(a+i)$	Element in $i$ th row, first column
$*(a+i+j)$	Element in first row, $j$ th column

#### Example 11.10 Matrix manipulation using matrix name as a pointer

##### a) Print a matrix

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        printf("%2d ", *(*(a+i)+j)); /* a[i][j] */
    printf("\n");
}
```

This program segment uses nested **for** loops to print a matrix (or its portion) of size  $m \times n$ . Observe that instead of the subscript notation ( $a[i][j]$ ), the array elements are printed using the equivalent pointer expression  $*(*(a+i)+j)$ .

##### b) Read a matrix

```

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        scanf("%d", *(a+i)+j ); /* &a[i][j] */
}

```

This code segment reads a matrix of size  $m \times n$  from the keyboard. Note that the expression `&a[i][j]` is replaced by its pointer equivalent, `*(a+i)+j`.

### Accessing Matrix Elements Using Another Pointer Variable

As we have just seen, the expression to access a matrix element using its name as a pointer is quite involved. We can improve code readability and efficiency by using a separate pointer variable that steps through the matrix.

There are two possible approaches to implement this. In first approach, we need to initialize a pointer to the beginning of a row to be processed and increment this pointer after each element is processed. This approach is particularly suitable for dynamic arrays in which each row is allocated a separate block of memory. In another approach, we can take advantage of the fact that the entire matrix is allocated a contiguous block of memory. We can thus initialize a pointer to the beginning of a matrix and increment it each time an element is processed.

#### Example 11.11 Process a matrix using a separate pointer variable

The function `imat_print2` given below prints an integer matrix using the first approach mentioned above. It uses a pointer variable `pa` to point to the element to be printed. For each row being printed, this pointer is initialized to the first element of that row using the expression `*(a+i)`. The pointer is incremented after each element is printed using the expression `*pa++`.

```

void imat_print2(int a[][4], int m, int n)
{
    int i, j;
    int *pa;

    for (i = 0; i < m; i++) {
        pa = *(a+i);      /* init int pointer to point to a[i][0] */

        for (j = 0; j < n; j++)
            printf("%2d ", *pa++); /* access elem & incr pointer */
        printf("\n");
    }
}

```

The function `mat_print3` given below uses the second approach in which the pointer variable `pa` is

first initialized to point to the beginning of the matrix (i. e., element  $a[0][0]$ ) using the expression  $*a$ . Then a nested **for** loop is used to print the matrix. Note how the pointer is incremented after each element is printed using expression  $*pa++$ .

```
void imat_print3(int a[][4], int m, int n)
{
    int i, j;
    int *pa = *a;          /* init int pointer to point to a[0][0] */

    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("%2d ", *pa++);
        printf("\n");
    }
}
```

Finally, note that we can initialize pointer **pa** using typecast as shown below.

```
int *pa = (int *) a; /* init int pointer to point to a[0][0] */
```

### Passing a Matrix to a Function Using a Pointer

As we know, we need not specify the number of rows when a matrix is passed to a function. Thus, the **mat\_func** function given below specifies that a matrix having unknown number of rows, each having four elements, will be passed as the actual argument.

```
void mat_func(int a[][4], int m, int n);
```

We can pass this matrix using equivalent pointer declaration as **int (\*a)[4]**. Here, the parentheses surrounding **\*a** are essential. We read this declaration as follows: **a** is a pointer to an array having four elements of type **int**. Since **a** is a pointer, we can pass an array having several elements, each of which is a four-element array, i. e., a matrix of size  $n \times 4$ . Thus, the declaration of **mat\_func** function given above can be equivalently rewritten as

```
void mat_func(int (*a)[4], int m, int n);
```

Note that within the function, we can access the elements of this matrix using the usual subscript notation. The **imat\_print4** function to print a matrix of size  $n \times 4$  is given below.

```
void imat_print4(int (*a)[4], int m, int n)
{
    int i, j;
```

```

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
        printf("%2d ", a[i][j]);/* access elem using subscript */
    printf("\n");
}

```

#### 11.4.4 Multidimensional Arrays and Pointers

We can extend the discussion in last section to multidimensional arrays. For example, consider the declarations of a three-dimensional array of size  $2 \times 3 \times 4$  given below.

```
int a[2][3][4];
```

We can interpret **a** as an array having three elements each of which is a matrix of size  $3 \times 4$ . Thus, contiguous memory is allocated for three matrices **a[0]**, **a[1]** and **a[2]**.

As we know, array name **a** is a pointer to its beginning. Actually, it is a pointer to a matrix of size  $3 \times 4$ , i.e.,  $(^*a)[3][4]$ . We can access the *ijk*th element of this matrix using pointer notation as  $*(^*(^*(a+i)+j)+k)$ . We can pass this array to a function and access its elements using pointer notation as illustrated in the **iarr3d\_print** function given below.

```

void iarr3d_print(int (*a)[3][4], int m, int n, int p)
{
    int i, j, k;
    int *pa = **a; /* init int pointer to point to a[0][0][0] */

    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < p; k++)
                printf("%2d ", *pa++); /* access elem & incr ptr */
            printf("\n");
        }
        printf("\n\n");
    }
}

```

Note that the pointer **pa** is initialized to **\*\*a**. As **a** is a pointer to array **a[0]** of size  $3 \times 4$ , **\*a** is a pointer to vector **a[0][0]** having four elements and dereferencing it once more as **\*\*a**, we get a pointer to element **a[0][0][0]**. Alternatively, we can write this initialization as

```
int *pa = (int *) a; /* init int pointer to point to a[0][0][0] */
```

### 11.4.5 Pointer to a Pointer

We know that a pointer can point to a value of any type. Thus, it can point to a pointer as well. Such a pointer to a pointer is depicted in Fig 11.5a, in which pointer **ppa** points to pointer **pa**, which in turn points to variable **a**. These pointers can be initialized as shown below.



**Fig. 11.5 Pointer to a pointer**

```
int a, *pa, **ppa;  
pa = &a;           /* pointer pa points to variable a */  
ppa = &pa;         /* pointer ppa points to pointer pa */
```

These declarations can be alternatively written in a more readable form as

```
int a;  
int *pa = &a;  
int **ppa = &pa;
```

Now we can use pointer **ppa** to access the value of variable **a**. As we know, the expression **\*ppa** gives the value pointed by pointer to **ppa** which is the value of pointer **pa** (i. e., the address of variable **a**). Thus, to access the value of variable **a**, we have to dereference pointer **ppa** once more, as in **\*\*ppa**. Thus, the expression **\*\*ppa = 10** assigns value 10 to variable **a** and **++\*\*ppa** increments its value. Note that the variable **a** can be accessed using pointer **pa** as well. Thus, we can also set variable **a** to 10 using expression **\*pa = 10**.

Note that we can extend this concept further. The declaration

```
int ***pppa = &ppa;
```

declares **pppa** as a pointer to a pointer to a pointer to variable **a** and initializes it to point to variable **ppa** declared earlier. This pointer is illustrated in Fig 11.5b. To access variable **a** using pointer **pppa**, we must dereference it three times. Thus, **\*\*\*pppa = 10** assigns value 10 to variable **a**.

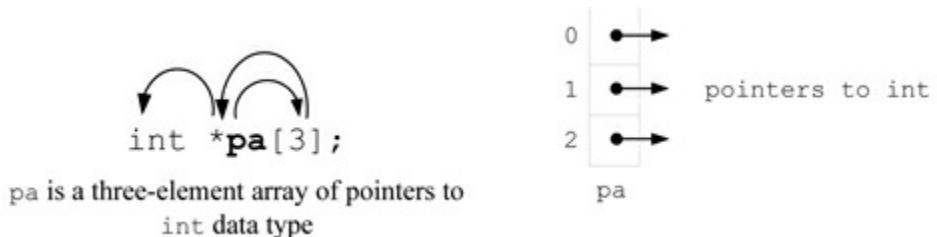
You might be wondering what is the use of such pointers to other pointers. However, as we will see, they are used to create dynamic multidimensional arrays. For example, a pointer to pointer is used to create a dynamic matrix, and a pointer to a pointer to a pointer is used to create a dynamic three-dimensional array.

### 11.4.6 Array of Pointers

Recall that we can declare an array of any data type. Thus, we can declare an array of pointers as well. Consider the declaration given below:

```
int *pa[3];
```

Since the [ ] operator has higher precedence than \* operator, **pa** is an array (having 3 elements) of type pointer to **int** as illustrated in Fig 11.6.



**Fig. 11.6 An array of pointers**

As we already know, the elements of array **pa** are not initialized and they have garbage values. Thus, we cannot dereference them unless we initialize them to point to data of appropriate type. For example, we can initialize the *i*th pointer to point to a variable of **int** type as **pa[i] = &a**. After this initialization, we can access the value pointed by pointer **pa[i]** as **\*pa[i]**.

However, usually dynamic memory allocation functions, namely, **malloc** and **calloc**, are used to allocate memory for each pointer and obtain a two-dimensional array. While using an array of pointers, the number of rows is fixed (decided at compile time, as it is an array), whereas the use of a pointer to a pointer allows the number of rows to be decided at run-time.

#### 11.4.7 Dynamic Memory Management

When a C program is compiled, the compiler allocates memory to store different data elements such as constants, variables (including pointer variables), arrays and structures. This is referred to as **compile-time or static memory allocation**. There are several limitations in such static memory allocation:

1. This allocation is done in memory exclusively allocated to a program, which is often limited in size.
2. A static array has fixed size. We cannot increase its size to handle situations requiring more elements. As a result, we will tend to declare larger arrays than required, leading to wastage of memory. Also, when fewer array elements are required, we cannot reduce array size to save memory.
3. It is not possible (or efficient) to create advanced data structures such as linked lists, trees and graphs, which are essential in most real-life programming situations.

The C language provides a very simple solution to overcome these limitations: **dynamic memory allocation** in which the memory is allocated at run-time, i. e., during the execution of a program. Dynamic memory management involves the use of pointers and four standard library functions,

namely, `malloc`, `calloc`, `realloc` and `free`. The first three functions are used to allocate memory, whereas the last function is used to return memory to the system (also called freeing/deallocating memory). The pointers are used to point to the blocks of memory allocated dynamically.

When called, a memory allocation function allocates a contiguous block of memory of specified size from the **heap** (the main memory of a computer) and returns its address. This address is stored in a pointer variable so as to access that memory block.

The memory allocated dynamically remains with the program as long as we do not explicitly return it to the system using the `free` function. Thus, when such memory is no longer required in our program, we should return it to the system. This is an important responsibility of a programmer, failing in which means that this memory will not be available to any program running on that machine including subsequent executions of our program. Thus, ill-designed programs (that do not free all allocated memory) will continue to *eat up* heap space, progressively reducing the computer's data processing ability. This is not desirable on any computer/device. Further execution of such programs will eventually lead to system hangs due to unavailability of memory and the system must be rebooted. Such a reboot may not be feasible on computers used in critical applications, e. g., servers such as Google, Yahoo, MSN, etc. Hence, a programmer should be carefully free all the allocated memory blocks.

The heap is managed by the operating system and is allocated (on demand) to running programs. The heap is much larger than the program's local memory used to hold program data. Thus, we can create much larger arrays and other data structures in the heap. Note that the allocation on the heap is not in a sequence, i. e., the memory blocks can be allocated anywhere. As a result, a heap memory is usually fragmented. The memory manager in the operating system decides the optimal location for allocation of a particular memory block.

### Standard library functions for dynamic memory management

Recall that the C language provides four functions for dynamic memory management, namely, `malloc`, `calloc`, `realloc` and `free`. These functions are declared in `stdlib.h` header file. They are summarized in Table 11.2 and are described below.

**Table 11.2 Dynamic memory management functions declared in the `stdlib.h` header file**

Function	Typical call	Description
<code>malloc</code>	<code>malloc(sz)</code>	Allocate a block of size $sz$ bytes from memory heap and return a pointer to the allocated block
<code>calloc</code>	<code>calloc(n, sz)</code>	Allocate a block of size $n \times sz$ bytes from memory heap, initialize it to zero and return a pointer to the allocated block
<code>realloc</code>	<code>realloc(blk, sz)</code>	Adjust the size of the memory block $blk$ allocated on the heap to $sz$ , copy the contents to a new location if necessary and return a pointer to the allocated block
<code>free</code>	<code>free(blk)</code>	Free block of memory $blk$ allocated from memory heap

The `malloc`, `calloc` and `realloc` functions allocate a contiguous block of memory from *heap*. If memory allocation is successful, they return a pointer to allocated block (i. e., starting address of the block) as a `void` (i. e., a typeless) pointer; otherwise, they return a `NULL` pointer.

### ***The `malloc` function***

The `malloc` (*memory allocate*) function is used to allocate a contiguous block of memory from heap. If the memory allocation is successful, it returns a pointer to the allocated block as a `void` pointer; otherwise, it returns a `NULL` pointer. The prototype of this function is given below.

```
void *malloc(size_t size );
```

The `malloc` function has only one argument, the *size* (in bytes) of the memory block to be allocated. Note that `size_t` is a type, also defined in `stdlib.h`, which is used to declare the sizes of memory objects and repeat counts. We should be careful while using the `malloc` function as the allocated memory block is uninitialized, i. e., it contains garbage values.

As different C implementations may have differences in data type sizes, it is a good idea to use the `sizeof` operator to determine the size of the desired data type and hence the size of the memory block to be allocated. Further, we should typecast the `void` pointer returned by these functions to a pointer of appropriate type. Thus, we can dynamically allocate a memory block to store 50 integers as shown below.

```
int *pa; /* pointer to the memory block to be allocated */
. . .
pa = (int *) malloc(50 * sizeof(int)); /* alloc memory */
```

This is often sufficient to allocate a memory block, particularly in small toy-like programs. However, in coding applications, it is a good idea to ensure, before continuing with program execution, that the memory allocation was successful, i. e., the pointer value returned is not `NULL` as shown below.

```
pa = (int *) malloc(50 * sizeof(int)); /* alloc memory */
if (pa == NULL) {
    printf("Error: Out of memory ...\\n");
    exit(1); /* Error code 1 is used here to indicate
               out of memory situation */
}
/* continue to use pa as an array here onwards */
```

Note that once memory is allocated, this dynamically allocated array can be used as a regular array. Thus, the *i*th element of array `pa` in the above example can be accessed as `pa[i]`.

Note that we can combine the memory allocation statement with the `NULL` test to make the code

concise (sacrificing readability) as shown below.

```
if ((pa = (int *) malloc(50 * sizeof(int))) == NULL) {  
    printf("Error: Out of memory ...\\n");  
    exit(1);  
}
```

If we use this approach to create concise programs, we are likely to make mistakes at least in the beginning. Hence, observe carefully the use of parentheses in the `if` statement. You will soon realize that it is not as difficult as it appears to be. Since the assignment statement has lower precedence than the equality operator, the memory allocation statement is first included in a pair of parentheses and then compared with the `NULL` value within the parentheses of `if` statement as illustrated below.

The diagram illustrates the precedence of operators in the `if` statement. A horizontal bracket groups the entire expression `(pa = (int *) malloc(50 * sizeof(int)))`. Another horizontal bracket groups the assignment operator `=` and its right-hand side `NULL`. A vertical bracket groups the entire comparison expression `((pa = (int *) malloc(50 * sizeof(int))) == NULL)`.

```
if ((pa = (int *) malloc(50 * sizeof(int))) == NULL) {
```

### ***The `calloc` function***

The `calloc` function is similar to `malloc`. However, it has two parameters that specify the number of items to be allocated and the size of each item as shown below.

```
void *calloc( size_t n_items, size_t size );
```

Another difference is that the `calloc` initializes the allocated memory block to zero. This is useful in several situations where we require arrays initialized to zero. It is also useful while allocating a pointer array to ensure that the pointers will not have garbage values.

### ***The `realloc` function***

The `realloc` (`reallocate`) function is used to adjust the size of a dynamically allocated memory block. If required, the block is reallocated to a new location and the existing contents are copied to it. Its prototype is given below.

```
void *realloc( void *block, size_t size );
```

Here, `block` points to a memory block already allocated using one of the memory allocation functions (`malloc`, `calloc` or `realloc`). If successful, this function returns the address of the reallocated block; or `NULL` otherwise.

### ***The `free` function***

The `free` function is used to deallocate a memory block allocated using one of the memory allocation functions (`malloc`, `calloc` or `realloc`). The deallocation actually returns that memory block to

the system heap. The prototype of `free` function is given below.

```
void free ( void *block );
```

As already discussed, when a dynamically allocated block is no longer required in the program, we must return its memory to the system.

#### 11.4.8 Dynamic Arrays

An array is a powerful and easy-to-use data structure provided in the C language. We know that arrays provide easy access to their elements and entire arrays can be manipulated easily using loops. However, there are some drawbacks/limitations of arrays:

1. *Inability to resize an array at run-time*: As the memory for arrays is allocated at compile time, it is not possible to change the array size during program execution. Thus, if the declared array size is insufficient in some situation, the executable program is almost useless. In such situations, we need to modify the source code suitably and recompile the program. This is either impossible (if the source code is not available) or very difficult (if the executable is to be distributed to a large number of persons). Also, it is not possible to reduce the array size to save memory in situations where fewer array elements are required.
2. *Memory wastage*: As array size cannot be increased at run-time, a programmer tends to declare arrays much larger than usually required to cater to some unforeseen situations. This leads to wastage of memory in most of the executions.

To overcome these limitations, we can implement the **dynamic arrays** using the dynamic memory allocation feature of the C language. This allows a programmer to allocate only the required memory for arrays. For example, if a class has only 35 students, an array to store names of the students in that class will be allocated to hold only 35 elements. On the other hand, while using static memory allocation, we might use much larger arrays, say having 100 elements, leading to large wastage of memory.

The general steps for working with a dynamic array of type `T` are as follows:

1. A pointer variable of suitable type (say `T*` for a vector, `T**` for a matrix, etc.) is declared to point to the array, the memory for which is to be allocated at run-time.
2. During program execution, the desired array size is determined (say through user interaction, file I/O or using some calculations).
3. The required memory is dynamically allocated to the pointer, preferably using a user-defined memory allocation function (e. g., `ivec_alloc` for `int` vector, `imat_alloc` for `int` matrix, etc.), that in turn uses the `malloc` or `calloc` functions.
4. The dynamic array can now be used in the usual manner. Thus, its elements can be accessed using subscript operators (e. g., `a[i]` for a vector, `a[i][j]` for a matrix, etc.) and the array can be

processed using loops. Also, we can pass these arrays to functions, although there are some differences for two and higher dimensions.

- 5. If we wish, we can change the size of this array, preferably using a user-defined function (say `ivec_realloc`, `imat_realloc`, etc.), that in turn uses the `realloc` standard library function and continue to work with it.
- 6. When this dynamic array is no longer required in the program, we should return it to the system, using the `free` function for vectors and a user-defined function (e. g., `imat_free`) for matrices and multidimensional arrays.

As we have just seen, the dynamic allocation of an array requires quite a bit of code. If the program requires a large number of dynamic arrays, which is quite possible even in a small application, we will end up with a lot of repetitive code for memory allocation. We can avoid this repetitive code and make our programs concise and more readable by defining functions for allocation/deallocation of such arrays.

### Dynamic Vectors

As we know, a pointer to type `T` is analogous to an array of type `T`. A pointer can be used to represent a vector, as illustrated in Fig 11.7. When a pointer is declared, the memory is allocated only for the pointer variable. The memory for the array elements is usually allocated separately using dynamic memory allocation functions, namely, `malloc` or `calloc`.



**Fig. 11.7** Representing a vector using a pointer

The function `ivec_alloc`, that dynamically allocates an integer vector containing  $n$  elements and returns a pointer to the allocated memory block, is given below. Note that the function return type is `int *`.

```
int *ivec_alloc(int n)
{
    int *tmp = (int *) malloc(n * sizeof(int));
    if (tmp == NULL) {
        printf("Error: Out of memory ...\\n");
        exit(1);
    }
    return tmp;
}
```

This function accepts an array size ( $n$ ) and allocates memory for the array, i. e.,  $n * \text{sizeof}(\text{int})$

bytes, to a temporary integer pointer `tmp`. This pointer is returned to the calling function. However, if the memory allocation is unsuccessful, it displays an error message and exits with error code 1. Thus, we require only a single line call to allocate a dynamic array as shown below.

```
int *a;  
.  
.  
a = ivec_alloc(100); /* allocate 100 element int array */
```

Note that the pointer variable `tmp` is local to the function and will be destroyed when the function returns. However, the address of the allocated memory block is returned to the calling function, and we can continue to use the block as a vector (through pointer `a` in above example) until we explicitly free its memory using `free` function.

Finally note that the above function is written specifically for the allocation of an integer vector and we need to write a separate function for the vector of each type required in the program. For this, we should first change the function name, say `fvec_alloc` for `float` type, `cvec_alloc` for `char` type, `sivec_alloc` for `short int` type, etc. Also, we should replace each `int` keyword (other than that in the function parameter declaration, `int n`) with the desired type. For example, function `fvec_alloc` for allocation of a `float` vector is given below.

```
float *fvec_alloc(int n)  
{  
    float *tmp = (float *) malloc(n * sizeof(float));  
    if (tmp == NULL) {  
        printf("Error: Out of memory ...\\n");  
        exit(1);  
    }  
    return tmp;  
}
```

### Program 11.1 Read a dynamic vector and print it in reverse order

A program to read a dynamic vector of type `float` and print it in reverse order is given below. It first reads the array size (`n`) from the keyboard and allocates a dynamic vector of that size using the `fvec_alloc` function. Then a `for` loop is used to read array elements from the keyboard and another `for` loop is used to print the array in reverse order. Observe that the elements of the dynamic array are accessed using the usual subscript notation, `a[i]`. Finally, the memory allocated to the array is freed and the program ends. Note that the program includes the `stdlib.h` header file.

```
#include <stdio.h>  
#include <stdlib.h>  
float *fvec_alloc(int n);
```

```

int main()
{
    float *a;      /* pointer for dynamic array */
    int i, n;

    printf("Enter vector size: ");
    scanf("%d", &n);
    a = fvec_alloc(n);      /* allocate vector */

    /* read a vector */
    printf("Enter vector elements: ");
    for (i = 0; i < n; i++)
        scanf("%f", &a[i]);

    /* print a vector */
    printf("Given vector in reverse order: ");
    for (i = n - 1; i >= 0; i--)
        printf("%5.2f ", a[i]);
    printf("\n");

    free(a);      /* free vector */
    return 0;
}

/* include definition of fvec_alloc function here */

```

The program output is given below.

```

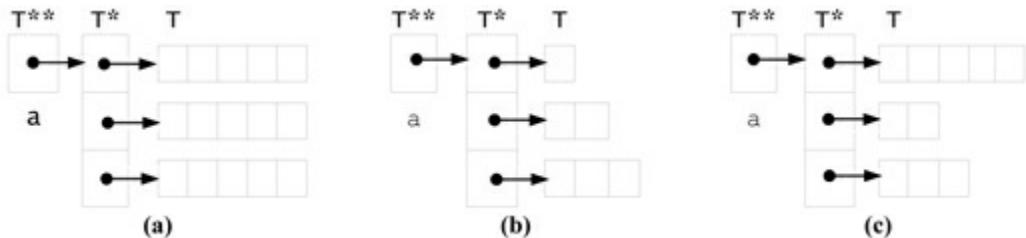
Enter vector size: 5
Enter vector elements: 1.2 2.3 3.4 4.5 5.6
Given vector in reverse order: 5.60 4.50 3.40 2.30 1.20

```

### Dynamic Two-Dimensional Arrays Using a Pointer to Pointer

Since a pointer to type T is analogous to an array of type T, a pointer to a pointer ( $T^{**}$ ) is analogous to an array of type  $T^*$ , i. e., an array of pointers to type T. Each element in this array of pointers can be used further to point to an array of type T. Thus, a pointer to a pointer can be used to represent two-dimensional arrays, as illustrated in Fig 11.8. Observe that we have a lot of flexibility in allocating memory for two-dimensional arrays. Thus, we can allocate a rectangular matrix (Fig 11.8a), a lower/upper triangular matrix (Fig 11.8b) and a ragged matrix in which each row can have different number of elements (Fig 11.8c). The use of a pointer to a pointer in conjunction with dynamic memory allocation is advantageous over a static matrix as it allows the allocation of a two-dimensional array of

desired size and shape, avoiding wastage of memory.



**Fig. 11.8** Representing two-dimensional arrays using a pointer to a pointer: (a) Regular matrix (b) Lower triangular matrix (c) Ragged matrix

Note that when a pointer to a pointer is declared, the memory is allocated only for the pointer variable. The memory for the pointer array and the two-dimensional array elements is usually allocated separately using dynamic memory allocation functions, namely, `malloc` or `calloc`.

A simplified function `imat_alloc` given below allocates a regular integer matrix of size  $m \times n$  to variable `tmp` of type `int **` and returns this pointer to the calling function.

```
int **imat_alloc(int m, int n)
{
    int i;

    int **tmp = (int **) malloc(m * sizeof(int *));
    for(i = 0; i < m; i++)
        tmp[i] = (int *) malloc(n * sizeof(int));

    return tmp;
}
```

Note that the function return type is a pointer to a pointer to `int`, i. e., `int **` and the function returns a value of correct type (as `tmp` is of type `int **`). First, `malloc` allocates memory for  $m$  pointers (i. e.,  $m * \text{sizeof}(\text{int } *)$ ) and assigns the address of this block, after typecasting to `int **`, to variable `tmp`. Then a `for` loop is used to allocate memory for each row. The `malloc` function now allocates memory for  $n$  integers (i. e.,  $n * \text{sizeof}(\text{int})$ ) and assigns the address of this block, after typecasting to `int *`, to the  $i$ th pointer `tmp[i]`. Finally, the value of `tmp` is returned.

As you have probably noticed, the error checking code has been omitted in `imat_alloc` function to keep the things simple and focus on memory allocation logic. Although this is okay for small programs, we should include the error handling code in actual applications. The modified function is given below.

```
/* allocate the memory for a dynamic matrix */
```

```

int **imat_alloc(int m, int n)
{
    int i;

    /* first allocate memory for pointer array */
    int **tmp = (int **) malloc(m * sizeof(int *));
    if (tmp == NULL) {
        printf("Error: Out of memory ...\\n");
        exit(1);
    }

    /* allocate memory for matrix rows */
    for(i = 0; i < m; i++) {
        tmp[i] = (int *) malloc(n * sizeof(int));
        if (tmp[i] == NULL) {
            printf("Error: Out of memory ...\\n");
            exit(1);
        }
    }
    return tmp;
}

```

This function can be called from any function to allocate a matrix as shown below.

```

int **a;
a = imat_alloc(m, n);

```

Note that now we cannot simply use the `free` function (as in `free(a)`) to free the memory allocated to a matrix as it will only free the memory allocated to the pointer array. The function `imat_free` to free an integer matrix is given below. It accepts two parameters: a pointer to the matrix and the number of rows. It first uses a `for` loop to free the memory allocated to matrix rows and then frees the memory allocated to the pointer array.

```

void imat_free(int **a, int m)
{
    int i;

    /* free the memory allocated to matrix rows */
    for (i = 0; i < m; i++)
        free(a[i]);

    free(a); /* free the memory allocated to pointer array */
}

```

## Program 11.2 Matrix addition using dynamic matrices

A program for adding integer matrices is given below. It uses dynamic matrices. The functions `imat_alloc` and `imat_free` are used for memory management. Their definitions are omitted to save space. The functions `imat_read`, `imat_print` and `imat_add` are used for operations on matrices.

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

int **imat_alloc(int m, int n);
void imat_free(int **a, int m);
void imat_read(char *msg, int **a, int m, int n);
void imat_print(char *msg, int **a, int m, int n);
void imat_add(int **a, int **b, int **c, int m, int n);

int main()
{
    int **a, **b, **c; /* pointers for dynamic matrices */
    int m, n;

    printf("Enter matrix size: ");
    scanf("%d %d", &m, &n);

    /* allocate matrices */
    a = imat_alloc(m, n);
    b = imat_alloc(m, n);
    c = imat_alloc(m, n);

    /* read matrices from keyboard, add them and print result */
    imat_read("Enter matrix a", a, m, n);
    imat_read("Enter matrix b", b, m, n);
    imat_add(a, b, c, m, n);
    imat_print("Addition matrix c", c, m, n);

    /* free matrices */
    imat_free(a, m);
    imat_free(b, m);
    imat_free(c, m);

    return 0;
}
/* imat_alloc() and imat_free() function defs omitted to save space */
```

```

/* read a matrix from keyboard */
void imat_read(char *msg, int **a, int m, int n)
{
    int i, j;

    printf("%s:\n", msg);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
    }
}

/* print a matrix */
void imat_print(char *msg, int **a, int m, int n)
{
    int i, j;

    printf("%s:\n", msg);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }

    printf("\n");
}

/* add two matrices */
void imat_add(int **a, int **b, int **c, int m, int n)
{
    int i, j;

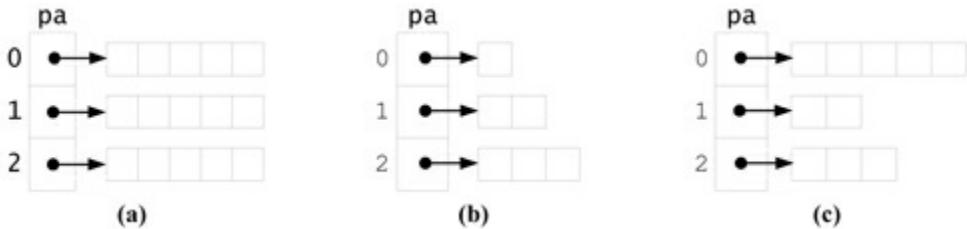
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
}

```

## Dynamic Two-Dimensional Arrays Using an Array of Pointers

Each element in one-dimensional array of pointers can be used to point to an array of type T. Thus, an array of pointers can be used to represent a two-dimensional array as illustrated in Fig 11.9. An array of pointers is also useful to represent an array of variable length strings (e. g., the command-line

arguments) as discussed in the next chapter.



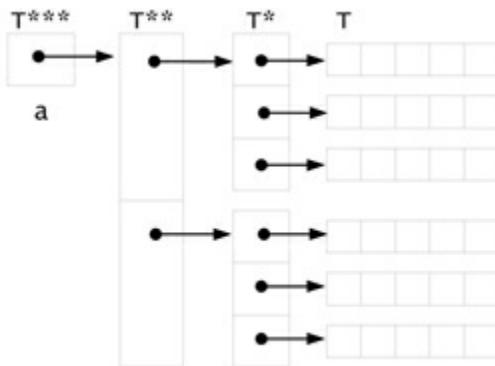
**Fig. 11.9** Representing two-dimensional arrays using an array of pointers: (a) Regular matrix (b) Lower triangular matrix (c) Ragged matrix

Note that as in the case of a pointer to a pointer, we have a lot of flexibility in allocating two-dimensional arrays as well. However, the maximum number of rows is limited by the size of the pointer array and cannot be changed. Hence, a pointer to a pointer is preferred over an array of pointers for implementation of dynamic two-dimensional arrays.

Also note that the memory allocated to array elements is not part of the pointer array. It is usually allocated separately using dynamic memory allocation functions, namely, `malloc` and `calloc`.

### Dynamic Multidimensional Arrays

We can extend the above concept to allocate memory for multidimensional arrays. For example, we can use a pointer to a pointer to a pointer to represent a three-dimensional array as shown in Fig 11.10 for an array of size  $2 \times 3 \times 5$ . The declaration for such a pointer is given below.



**Fig. 11.10** Representing a three-dimensional array using a pointer to a pointer to a pointer

```
int ***a;
```

As in case of the two-dimensional array, the memory is allocated only for this pointer variable. The remaining memory for pointer arrays and array elements is usually allocated using dynamic memory allocation functions.

A function `i3darr_alloc` to allocate memory for a three-dimensional array of size  $m \times n \times p$  is given below (the error handling code is omitted to keep the code simple).

```
/* allocate memory for dynamic 3-D array.
Error handling code is omitted to save space */
int ***i3darr_alloc(int m, int n, int p)
{
    int i, j;

    /* allocate memory for pointer array of size m */
    int ***tmp = (int ***) malloc(m * sizeof(int **));

    /* allocate memory for m matrices */
    for(i = 0; i < m; i++) {
        /* allocate memory for pointer array of size n */
        tmp[i] = (int **) malloc(n * sizeof(int *));

        /* allocate memory for n rows in each matrix */
        for (j = 0; j < n; j++)
            tmp[i][j] = (int *) malloc(n * sizeof(int));
    }

    return tmp;
}
```

A function `i3darr_free` to free the memory allocated to a three-dimensional array using the above function is given below.

```
/* free memory allocated to dynamic 3-D array */
void i3darr_free(int ***a, int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++) { /* free memory for m matrices */
        for (j = 0; j < n; j++) /* free memory for n rows of each mat
            free(a[i][j]);
        free(a[i]);
    }
    free(a);
}
```

Note that we can also represent a three-dimensional array using a two-dimensional array of pointers. For example, the declaration given below declares `pm` as a two-dimensional array of size  $10 \times 20$  of pointers to `int` type.

```
int *pm[10][20];
```

This array can be used to represent a dynamic three-dimensional array of size  $10 \times 20 \times p$ . However, the size of matrix `pm` restricts the maximum array size. Moreover, if we need smaller arrays, this approach leads to wastage of memory. The use of a pointer to a pointer to a pointer is advantageous as it enables the use of such arrays of desirable size.

#### 11.4.9 Pointer to a Function

We can declare a **pointer to a function** using the following syntax.

```
ret_type (*func_name)(param_list)
```

where `func_name` is a pointer to a function, `param_list` is a comma-separated list of function parameter declarations and `ret_type` is the return type of the function. Note that the parentheses surrounding `*func_name` are essential; otherwise, the pointer becomes associated with `ret_type`. For example, in declaration

```
int (*test)(float, float)
```

`test` is a pointer to a function that accepts two parameters of type `float` and returns an integer value. However, in declaration

```
int *test(float, float)
```

`test` is a function that accepts two parameters of type `float` and returns a pointer to `int`.

We can assign a function to a function pointer and call this assigned function through a function pointer as illustrated in the example given below. A pointer to a function is also useful to write a polymorphic function, as discussed in Section 11.4.10.

#### Example 11.12 Using function pointers

Consider that we wish to print dates in different formats. Let us write different functions for each desired format. For example, the function `print_date1` and `print_date2` given below print the date `15/12/11` in the formats `15/12/11` and `December 15, 2011`, respectively.

```
/* print date in dd/mm/yy format */
void print_date1(int dd, int mm, int yy)
{
    printf("%d/%d/%d\n", dd, mm, yy);
}
/* print date in month_name dd, yyyy format */
void print_date2(int dd, int mm, int yy)
```

```

{
    static char *month[] = {
        "", "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "December"
    };

    if (yy < 100)
        yy += 2000;
    printf("%s %d, %d\n", month[mm], dd, yy);
}

```

Note that in the second function, `month` is a static array of character pointers. It is used to store the names of the months. Also note that if a two-digit value is provided for year `yy`, it is converted to four-digit value by adding 2000 to it.

Now, to understand how these functions can be called through a function pointer, consider the `main` function given below.

```

int main()
{
    void (*print_date)(int, int, int); /* function pointer */

    print_date = print_date1;
    print_date(15, 12, 11);

    print_date = print_date2;
    print_date(15, 12, 11);

    return 0;
}

```

This function first declares `print_date` as a pointer to a function that accepts three integer values and returns no value. Then it assigns function `print_date1` to function pointer `print_date` and calls this function through the function pointer. Similarly, it calls function `print_date2` through the function pointer `print_date`.

#### 11.4.10 Polymorphic Functions using **void** Pointer

The word **polymorphic** means having multiple forms. It is derived from two Greek words: *poly* meaning many and *morph* meaning form. We can write different types of polymorphic functions. These include functions that perform different computations, produce different output, or work with different data types.

## Polymorphic Functions Having Different Computation or Output

To write a **polymorphic function** that performs different computation or has different output, we can declare a function pointer as one of the parameters of this function. For example, we can write a polymorphic function *pmfunc* as

```
ret_type pmfunc (param_decl, void (*func_ptr)(param_decl) )
```

When this function is called, we can pass as an argument a function that matches with *func\_ptr*. This function is then called through *func\_ptr*. Since we can pass different functions that perform computations differently, the polymorphic function *pmfunc* will either perform a different computation or produce a different output. This is illustrated in the example given below.

### Example 11.13 Writing polymorphic functions that performs different computation

Consider that we wish to calculate interest on fixed deposit. The interest may be calculated as simple interest or compound interest. We can write two functions, namely, `simple_interest` and `compound_interest` as shown below.

```
float simple_interest(float deposit, int years, float intr_rate)
{
    return deposit * years * intr_rate / 100;
}

float compound_interest(float deposit, int years, float intr_rate)
{
    return deposit * pow((1 + intr_rate / 100), years) - deposit;
}
```

Now we can define a function named `interest` that calculates either simple interest or compound interest as shown below.

```
float interest(float deposit, int years, float intr_rate,
              float (*calc_interest)(float, int, float))
{
    return calc_interest(deposit, years, intr_rate);
}
```

To calculate simple interest, this function can be called as shown below.

```
int main()
{
    float intr = interest(1000, 3, 10.5, simple_interest);
```

```
        printf("Interest: %0.2f", intr);
}
```

## Polymorphic Functions that Operate on Arrays of Different Data Types

The requirements for defining a polymorphic function that operates on vectors are given below:

1. The array to be operated upon should be passed using a **void** pointer.
2. The number of array elements and the size of each array element should be passed.
3. The operation(s) dependent on array data type should be performed using separate functions. The polymorphic function should accept a pointer to this function as a parameter. For example, while performing searching and sorting operations, comparison of array elements will vary with the type of data. Thus, **bsearch** and **qsort** functions use a parameter **fcmp** which is a pointer to the comparison function.
4. The function for data type specific operations should use **void** pointers to pass data values to be operated upon. For example, the parameter **fcmp** representing comparison function in **bsearch** function uses two parameters of type **const void \***.

### Example 11.14 A polymorphic function to print a vector of any type

Now let us write a polymorphic function **vec\_print** to print a vector of any type. The code to print a vector typically uses a **printf** function call within a **for** loop. Thus, code to print vectors of different types, say **int**, **float**, etc., will differ only in the **printf** function call. Thus, we can use a separate function to print an array element of the desired type. For example, functions **print\_int** and **print\_float** to print an element of type **int** and **float**, respectively, are given below.

```
void print_int(const void *a)
{
    printf("%d ", *(int *) a);
}

void print_float(const void *a)
{
    printf("%0.2f ", *(float *) a);
}
```

Observe that a **void** pointer is used to pass an element to be printed to these functions. This pointer is typecast to the desired type before it is dereferenced and the value pointed by it is printed.

The definition of **vec\_print**, a polymorphic function that prints an array of any type is given below.

```

void vec_print(const char *msg, const void *a, int n, size_t sz,
              void (*print_elem)(const void *))
{
    int i;

    printf("%s: ", msg);
    for(i = 0; i < n; i++)
        print_elem(a + i * sz);
    printf("\n");
}

```

The first parameter `msg` is the message to be printed before the array elements are printed. Next three parameters, namely, `a`, `n` and `sz`, specify the array to be operated upon. The last parameter, `print_elem`, is a pointer to a function that accepts a constant void pointer as a parameter and returns no value.

Observe how the `print_elem` function is called inside the `for` loop. A pointer to the *i*th array element is obtained using the expression `a + i * sz`. We can alternatively write this function more efficiently by updating pointer `a` after each element is printed as shown below.

```

for(i = 0; i < n; i++) {
    print_elem(a);
    a += sz;
}

```

The main function that initializes two arrays and prints them using the `vec_print` function is given below.

```

int main()
{
    int a[] = {5, 3, 4, 1, 8, 7, 10, 2, 6, 9};
    float x[] = {1.1, 5.5, 3.3, 2.2, 4.4};

    vec_print("int array", a, 10, sizeof(a[0]), print_int);
    vec_print("float array", x, 5, sizeof(x[0]), print_float);
    return 0;
}

```

The output obtained by running this function is given below.

```

int array: 5 3 4 1 8 7 10 2 6 9
float array: 1.10 5.50 3.30 2.20 4.40

```

### 11.4.11 Complex Declarations Involving Pointers

The declarations involving pointers, arrays and functions can be quite complex. To understand these declarations, remember that the [ ] and ( ) operators have higher precedence than the \* operator. Several declarations involving pointers and arrays are given in Table 11.3.

The first three declarations in Table 11.3 are useful for representing two-dimensional arrays. As we know, `int **pp` declares `pp` as a pointer to a pointer to integer, `int *ap[]` declares `ap` as an array of pointers to integer and `int (*pa)[]` declares `pa` as a pointer to an array of integers.

**Table 11.3** Complex declarations involving pointers and arrays

Declaration	Meaning
<code>int **pp</code>	Pointer to pointer to integer
<code>int *ap[]</code>	Array of pointers to integer
<code>int (*pa)[]</code>	Pointer to an array of integer
<code>int ***ppp</code>	Pointer to a pointer to a pointer to integer
<code>int **app[]</code>	Array of pointer to pointer to integer
<code>int (**ppa)[]</code>	Pointer to a pointer to an array of integer
<code>int *(*pap)[]</code>	Pointer to an array of pointers to integer
<code>int (*apa[])[]</code>	Array of pointers to array of integer

The next five declarations can be used for representing three-dimensional arrays. The declaration `int ***ppp` declares `ppp` as a pointer to a pointer to a pointer to an integer, `int **app[]` declares `app` as an array of pointers to a pointer to integer, `int (**ppa)[]` declares `ppa` as a pointer to a pointer to an array of integers. The declaration `int *(*pap)[]` declares `pap` as a pointer to an array of pointers to integer and `int (*apa[])[]` declares `apa` as an array pointers to an array of integers.

Seveal declarations involving pointers, arrays and functions are given in Table 11.4. The declaration `int *fp()` declares `fp` as a function that returns a pointer to an integer and `int **fpp()` declares `fpp` as a function that returns a pointer to a pointer to integer.

The declaration `int (*pf)()` declares `pf` as a pointer to a function that returns an integer, `int (**ppf)()` declares `ppf` as a pointer to a pointer to a function that returns an integer, and `int (*pfp)()` declares `pfp` as a pointer to a function that returns a pointer to integer.

**Table 11.4** Complex declarations involving pointers, arrays and functions

Declaration	Meaning
-------------	---------

<code>int *fp()</code>	Function which returns a pointer to integer
<code>int **fpp()</code>	Function with return value pointer to a pointer to integer
<code>int (*pf)()</code>	Pointer to a function with return value integer
<code>int (**ppf)()</code>	Pointer to a pointer to a function with return value integer
<code>int *(*pfp)()</code>	Pointer to function with return value pointer to integer
<code>int (*apf[])()</code>	Array of pointers to functions with return value integer
<code>int (*fpa())[]</code>	Function with return value pointer to array of integers
<code>int (*fpf())()</code>	Function with return value pointer to function, which returns an integer

The declaration `int (*apf[])()` declares `apf` as an array of pointers to functions that return integer values, `int (*fpa())[]` declares `fpa` as a function that returns a pointer to an array of integers, and `int (*fpf())()` declares `fpf` as a function that returns a pointer to a function that returns an integer value.

## Exercises

- Select the correct option for the following questions:
  - `p, &p`
  - `*p, p`
  - `*p, &p`
  - `*v, &v`
- Consider that `a` is an array having 10 integers (each requiring two bytes) with values 10, 20, ..., 100 and `pa` and `pb` are pointers to its first and last element, respectively. What is the value of `pb - pa`?
  - 90
  - 18
  - 9
  - Error
- If pointer `p` points to element `a[2]` of a float array (requiring four bytes for each element), which of the following expression can be used to increment element `a[3]`?

- i.  $*p++$
  - ii.  $++*(p+4)$
  - iii.  $*++p$
  - iv.  $++*++p$
- d. If pointer  $p$  points to an array element, which of the following expression provides the value of that element and also increments the pointer to point to the next value?
- i.  $(*p)++$
  - ii.  $*++p$
  - iii.  $*(p++)$
  - iv. None of these
- e. Which of the following operations can be performed on pointer pointing to elements of same array?
- i. Addition
  - ii. Subtraction
  - iii. Multiplication
  - iv. All of these
2. State whether the following statements are true or false. Also, rewrite the incorrect statements correctly.
- a. A pointer to a variable of type  $T$  requires  $\text{sizeof}(T)$  bytes for its storage.
  - b. The number of elements between two pointers, say  $pa$  and  $pb$ , pointing to an array of type  $T$  is given as  $(pb - pa)/\text{sizeof}(T)$ .
  - c. A pointer to an ordinary variable cannot be incremented as in case of a pointer to an array element.
  - d. When a pointer pointing to an array element is incremented, it points to the next array element irrespective of the size of array elements in bytes.
3. Determine the output of the following program segments.
- a. 

```
int a = 10;
int *b = &a;
printf("%d ", --*b);
printf("%d", *b--);
```
  - b. 

```
int a = 2, b = 10;
int *pa = &b, pb = &a;
*pa /= *pb;
```

```

*pb *= *pa;
printf("%d %d %d %d", a, b, *pa, *pb);

c. int a[] = {1, 2, 3, 4, 5};
int *pa = a, *pb = &a[5];
printf("%d ", --pb - ++pa);

d. int a[] = {1, 2, 3, 4, 5};
int *pa = &a[3];
printf("%d ", *--pa);
printf("%d ", *pa--);
printf("%d ", *--pa);

e. int a[10], *p, i;
for(i = 0; i < 10; i++)
    a[i] = i+1;
p = a;
for(i = 10; i > 6; i--, p++)
    printf("%d ", *p++);

f. int a[] = {1, 2, 3, 4, 5};
int *pa;
for(pa = a; pa < a + 5; ++pa)
    (*pa)++;
while(pa > a)
    printf("%d ", *--pa);

```

l. Write the code for the following functions:

- Function `vol_sa` that accepts the radius of a sphere and return it's volume and surface area.
- Function `rotate3` that accepts three integers (say, `a`, `b`, `c`), assigns values of `a` to `b`, `b` to `c` and `c` to `a` and returns these modified values.
- Function `to_feet_inch` that accepts length in meters, converts it into feet and inches and returns both the values to the calling function such that the feet value should be returned as function value.
- Function `swap_ptr` that interchanges two integer pointers.

## Exercises (Advanced Concepts)

- Answer the following questions in brief:
  - Write the expression to access elements `a[0][0]`, `a[2][0]` and `a[0][3]` using pointer notation.
  - Give a declaration to store at the most 10 strings using an array of pointers.

- c. What is a polymorphic function? Give suitable example.
  - d. State the functions for dynamic memory allocation along with their purpose.
  - e. State the differences between a pointer to a pointer and an array of pointers.
  - f. What precaution should be taken while returning a pointer from a function.
2. Select the correct option for the following questions:
- a. If `pa` is a pointer to a `long double`, what is the value of the expression `sizeof(*pa)`? (Assume 4-byte pointers).
    - i. 4
    - ii. 8
    - iii. 10
    - iv. None of the above
  - b. Which of the following expression can be used to access element `a[3][4]`?
    - i. `**((a+3)+4)`
    - ii. `*(*(a+4)+3)`
    - iii. `**(a+3)+4`
    - iv. `*(*(a+3)+4)`
  - c. If `pppx` is a pointer to a pointer to a pointer to variable `x` of type `int`, how do we increment the value of `x`?
    - i. `***pppx++`
    - ii. `***++pppx`
    - iii. `++***pppx`
    - iv. `*(**pppx)++`
  - d. Which among following functions is used to free the memory allocated using `realloc` function?
    - i. `free`
    - ii. `rfree`
    - iii. `refree`
    - iv. None of these
  - e. A lower triangular matrix of `float` type is dynamically allocated using an array of pointers. What is the minimum memory required if the matrix has 4 rows? (Assume 2-byte pointers)
    - i. 64 bytes

- ii. 40 bytes
  - iii. 56 bytes
  - iv. 48 bytes
3. State whether the following statements are true or false.
- a. We can convert a pointer to `char` to a pointer to `long double`.
  - b. A function can return a pointer to an element of an array passed to it as an argument.
  - c. While allocating memory for a ragged matrix, we will require less memory when using an array of pointers compared to that required by using a pointer to a pointer.
  - d. We should never return a pointer to a variable declared within a function.
4. Write the code for the following functions.
- a. Accept an array of integers and return a pointer to its largest element.
  - b. Accept two strings, `str1` and `str2`, as parameters and return a pointer to a position where string `str2` is found in `str1` and `NULL` otherwise.
  - c. Accept an array of pointers to `int` type and dynamically allocate a lower triangular array having  $n$  rows.
  - d. Allocate memory for a three-dimensional array in which individual matrices have different sizes. Accept a one-dimensional array that specifies the number of matrices and their sizes, and dynamically allocate a three-dimensional array.
  - e. Print a complex number in either `a+ib` or `(a, b)` form using polymorphic functions implemented using function pointers.
  - f. Return a pointer to the string containing the name of the month given as a parameter.
5. Determine the output of the following program segments.
- a. 

```
int a = 10;
int *pa = &a;
int **ppa = &pa;
printf("%d ", **ppa);
printf("%d ", ++**ppa);
```
  - b. 

```
float a = 10;
float *pa = &a;
int *pb = (int *) pa;
printf("%d", sizeof(pa) ==
      sizeof(pb));
```
  - c. 

```
char *s[] = {"Hi", "Bye"};
printf("%c", **s);
printf("%c", *s[0]);
```

```
printf("%c", *(s+1));
printf("%c", s[1][2]);
d. int *pa, i, sum = 0;
pa = (int *) calloc(10, sizeof(int));
for(i = 1; i < 10; i++)
    sum += ++*pa++;
printf("%d", sum);
```

5. Give the declarations for the following.

- a. Pointer to an array of **double**
  - b. Pointer to an array of pointers to **float**
  - c. Function which returns a pointer to **char**
  - d. Pointer to function with return value pointer to **int**
  - e. Array of pointers to functions with return value **unsigned int**
- 

<sup>1</sup> The default 2-byte pointers in Turbo C/C++ are called *near* pointers, because they can point to variables within a 64-KB memory segment. The *far* pointers, on the other hand, require 4 bytes of memory but are capable of pointing to a variable in other memory segments, i. e., anywhere in the memory.

<sup>2</sup> *lvalue* is an expression which can appear on the left hand side of an assignment operator.

<sup>3</sup>  $a[i][j] \equiv (a[i])[j] \equiv (*a)[i][j] \equiv *(*a+i)+j$

# 12 Strings

This chapter presents the operations we can perform on character strings. Strings are usually processed using `while` loops, so we first study this in detail. Then, we examine how to write user-defined functions to process strings. The C library provides a rich set of functions to process strings and we take a closer look at these functions.

The *Advanced Concepts* section covers several topics on string manipulation, such as their pitfalls, writing more involved string manipulation functions, nesting of string manipulation functions, advanced library functions, use of dynamic memory allocation to save memory and command-line arguments.

## 12.1 Processing Strings

### 12.1.1 Processing Strings Using Loops

We know that a string is a sequence of characters terminated with a null character and is stored in an array of characters. Several operations on strings involve performing the same or similar operation(s) on each character in a string. Some examples are given below:

1. To display a string on the screen, each character in it is displayed on the screen.
2. To copy a string to another string, each character in it is copied to the target string.
3. To determine the length of a string, we count the characters in it, one by one.
4. To convert a string to uppercase representation, each lowercase letter in it is replaced by the corresponding uppercase letter.

Thus, string operations usually involve performing the desired operation(s) on each character in it using a loop. In this section, we study how to perform operations on strings.

As strings in C are null-terminated, the number of characters in a given string is not readily known. Also, we cannot directly tell the position of the last character in the string. Hence, most operations on strings usually proceed from the first character to the last. A `while` loop can thus be used to perform operations on strings, as shown below.

```
i = 0;  
while (str[i] != '\0') {
```

```

/* perform desired operation on str[i] */
...
i++;
}

```

The loop counter `i` is first initialized to 0. It thus points to the first character in string `str`. The body of the `while` loop is executed as long as `str[i]`, i. e., the character at the *i*th position, is not a null character. Within the body of the `while` loop, the desired operation is performed on `str[i]`, after which the loop counter is incremented. Note that we can equivalently write the above `while` loop by omitting the test simply as `while(str[i])`.

As the `while` loop and `for` loop are equivalent, we can rewrite the above code concisely using a `for` loop, as shown below.

```

for (i = 0; str[i] != '\0'; i++) {
    /* perform desired operation on str[i] */
    ...
}

```

However, the `do . . while` loop may not be suitable in some situations as its body is executed at least once which may not be correct while operating on a null string.

### Example 12.1 Operations on strings using loops

The standard C library provides several functions to perform basic operations on strings. These include functions to read a string from the keyboard (`gets` ), display a string (`puts` ), determine the length of a string (`strlen` ), copy a string (`strcpy` ), concatenate two strings (`strcat` ), convert a string to uppercase (`strupr` ), compare two strings (`strcmp` ), etc.

In this example, we study how such basic operations are performed on strings using loops. These examples use character arrays named `str`, `a`, `b`, `c`, etc. to store strings. These arrays and other variables are assumed to be declared at the appropriate places. It is also assumed that the strings have been initialized or read from the keyboard using either `gets` or `scanf` function.

#### a) Determine length of a string

The length of a string is defined as the number of character in it excluding the null terminator. The program segment given below determines the length of a string.

```

len = 0;
while (str[len] != '\0')
    len++;

```

The character counter `len` is first initialized to 0 and a `while` loop is then used to determine the

string length. The body of the loop has only one statement that increments the counter `len`. After execution of the `while` loop is complete, the variable `len` contains the string length. Note that we may rewrite the above code using a `for` loop, as shown below:

```
for(len = 0; str[len] != '\0'; len++)
;
```

or even more concisely as

```
for(len = 0; str[len]; len++)
;
```

Observe that the only statement in the body of the `while` loop has now become the update expression in the `for` loop. Thus, no statement is required within the body of the `for` loop. In such situations, it is good programming practice to include a *null statement* on a line by itself, as shown above.

### b) Display a string on the screen

To display a string on the screen, we need to display each character in it (using either `putchar` or `putch` function) until we reach the null terminator, which is not to be displayed. A program segment to display a string `str` is given below.

```
i = 0;
while (str[i] != '\0') {
    putchar(str[i]);
    i++;
}
```

### c) Convert a string to uppercase

The program segment given below converts a string to uppercase representation.

```
for(i = 0; str[i] != '\0'; i++) {
    if (str[i] >='a' && str[i] <= 'z') /* lowercase letter? */
        str[i] += 'A' - 'a';           /* convert it to uppercase */
}
```

It uses a `for` loop to test each character in the given string and if it is a lowercase letter, it is replaced by the corresponding uppercase letter. Note that a character is a lowercase character if it is greater than or equal to 'a' and less than or equal to 'z' and to replace a lowercase letter with its uppercase representation, we add 'A' - 'a' (i. e., -32) to it or subtract 32 from it.

We can simplify this code using the `toupper` function provided in the `ctype.h` header file, which converts a character to uppercase. The test for lowercase character is not required as the

`toupper` function converts a lowercase letter to uppercase and other characters are returned without any change. The modified code is given below.

```
for(i = 0; str[i] != '\0'; i++)
    str[i] = toupper(str[i]);
```

#### d) Copy a string

To copy a string to another string, we have to copy each character in the source string to the target string, including the null terminator. It is assumed that the target array has enough space to accommodate all the characters being copied to it. The program segment given below copies string `b` to string `a`.

```
i = 0;
while (b[i]      != '\0') {
    a[i] = b[i];
    i++;
}
a[i] = '\0'; /* terminate string a with a null character */
```

The `while` loop is set up as before. The main operation within the loop is to copy the  $i$ th character from source string `b` to target string `a`. Observe that the `while` loop does not copy the null terminator to the target string. Hence, we have copied it after the `while` loop. Forgetting to write a null terminator leads to difficult-to-trace bugs in the program and we should always be careful about it.

We now present two alternatives code segments that eliminate the need for a separate statement to write a null terminator to the target string. For this, we have to first copy a character from the source string to the target string and then test whether it is a null terminator or not. Thus, a `do . . while` loop can be used in this situation, as shown below.

```
i = 0;
do
    a[i] = b[i];      /* copy a char, including null terminator */
while (b[i++] != '\0');
```

Observe that the loop variable `i` is incremented using the postfix increment operator in the loop test. It can be verified that the code works correctly even for a null string.

Another implementation using a `while` loop is given below, in which the loop test is modified to copy the  $i$ th character to the target string before it is actually tested.

```
i = 0;
while ((a[i] = b[i]) != '\0') /* copy character and test it */
    i++;
```

As the inequality operator (`!=`) has higher precedence than the assignment operator, the parentheses surrounding the assignment expression `a[i] = b[i]` are essential. Observe that this implementation is more elegant than the `do .. while` implementation. This `while` loop can be interpreted as follows: `while a[i]`, which is assigned the value of `b[i]`, is not equal to null, increment the value of variable `i`.

#### e) Concatenate two strings

String concatenation involves appending a string at the end of another string. For this, we have to first locate the null terminator in the target string and then, starting from this position, copy all the characters from the source string including the null terminator. The program segment given below uses `while` loops to concatenate string `b` to string `a`.

```
/* locate null terminator in string a */
i = 0;
while (a[i] != '\0')
    i++;

/* copy string b to string a starting from position i */
j = 0;
while (b[j] != '\0') {
    a[i] = b[j];
    i++;
    j++;
}
a[i] = '\0'; /* append null character */
```

Using the coding style in the previous example, we can rewrite the code to copy string `b` as

```
/* copy string b to string a starting from position i */
j = 0;
while ((a[i] = b[j]) != '\0')
    i++, j++;
```

Note that the assignment `a[i] = b[j]` has been moved to the loop test, eliminating the last statement to copy the null terminator. Also, the statements to increment the values of variables `i` and `j` have been combined using a comma operator, eliminating the braces. We can make this code more concise by eliminating the `!=` operator and including the increment operators in the loop test, as shown below.

```
/* copy string b to string a starting from location i */
j = 0;
while(a[i++] = b[j++])
```

;

#### f) Read a string from the keyboard

Reading a string from the keyboard is slightly different from the earlier examples in that the user cannot enter a null character to mark the end of a string. Hence, we can use the newline or space character to mark the end of a string input. A program segment to read a string `str` from the keyboard until the *Enter* key is pressed is given below.

```
i = 0;
ch = getchar(); /* read first character */
while (ch != '\n') {
    str[i] = ch;
    ch = getchar();
    i++;
}
str[i] = '\0'; /* terminate string with a null character */
```

Initially, the loop variable `i` is initialized to 0 and the first character is read from the keyboard into character variable `ch`. Then a `while` loop is set up to continue until a newline is entered. In each iteration of the loop, character `ch` is assigned to `str[i]`, the next input character is read from the keyboard and the loop variable `i` is incremented. Finally, a null terminator is stored in the string after the `while` loop is completed.

Note that the `getchar` function is called twice, before the loop as well as inside it. Such operations can be moved inside the loop test to obtain the same effect and avoid duplication. The modified program segment is given below.

```
i = 0;
while ((ch = getchar()) != '\n') {
    str[i] = ch;
    i++;
}
str[i] = '\0'; /* terminate string with a null character */
```

The above implementations use a character variable `ch` to hold the input character before it is stored in the string. We can eliminate this variable and read the input character directly in the *i*th position of the string, as shown below.

```
i = 0;
while ((str[i] = getchar()) != '\n')
    i++;
str[i] = '\0'; /* terminate string with a null character */
```

This is a bit tricky (but correct) code. Note that the newline character entered from the keyboard is stored in the `str` array, though it is not supposed to be. However, this does not cause any problem as the last statement overwrites it with a null terminator.

Although these implementations accept a string as expected, there is still a problem that they accept as many characters as the user will enter without taking into account the size of the array `str`. Hence, as a final improvement, the modified program segment given below limits the maximum number of input characters to `max_len`.

```
i = 0;
while (i < max_len && (str[i] = getchar()) != '\n')
    i++;
str[i] = '\0';
```

Note that due to strict left-to-right evaluation of operands of the `&&` operator, the test `i < max_len` is performed before a character is read from the keyboard.

#### Program 12.1 Count the number of letters, digits, white spaces and other characters

Write a program to read a string from the keyboard and count the number of letters, digits, white spaces and other characters in it.

**Solution:** Let us use an array of characters (`str`) to store the string and variables `nletter`, `ndigit`, `nspace` and `nother` to count the letters, digits, white spaces and other characters, respectively. The program given below first reads a string from the keyboard using the `gets` function and then uses a `while` loop to determine the desired counts. An `if-else-if` statement is used within the `while` loop to test the *i*th character and update the respective counter. Note the use of variable `ch` to minimize access to the array element (possibly improving its efficiency) as well as to improve the readability of the code. Finally, the program prints the counts.

```
/* Count letters, digits, whitespaces and other chars in a given string
#include <stdio.h>

int main()
{
    char str[81];
    int nletter, ndigit, nspace, nother; /* char counts */
    int i;

    printf("Enter a line of text:\n");
    gets(str);

    /* count characters in string str */
    nletter = ndigit = nspace = nother = 0; /* init counts */
```

```

i = 0;
while (str[i] != '\0') {
    char ch = str[i];
    if (ch >= 'A' && ch <= 'Z' ||
        ch >= 'a' && ch <= 'z')
        nletter++;
    else if (ch >= '0' && ch <= '9')
        ndigit++;
    else if (ch == ' ' || ch == '\n' || ch == '\t')
        nspace++;
    else nother++;
    i++;
}
/* print counts */
printf("Letters: %d \tWhite spaces: %d", nletter, nspace);
printf(" Digits : %d \tOther chars : %d\n", ndigit, nother);
return 0;
}

```

The output of the program is given below.

```

Enter a line of text:
God helps them who help themselves.
Letters: 28      White spaces: 4      Digits : 0      Other chars : 2

```

We can now rewrite this code to make it concise and to improve its readability. This is done using a `for` loop, instead of a `while` loop, and the character classification functions (`isletter`, `isdigit`, `isspace`) from the `ctype.h` header file. The variable `ch` is also eliminated.

```

/* count characters in string str */
nletter = ndigit = nspace = nother = 0; /* init counts */
for (i = 0; str[i] != '\0'; i++) {
    if (isletter(str[i]))
        nletter++;
    else if (isdigit(str[i]))
        ndigit++;
    else if (isspace(str[i]))
        nspace++;
    else nother++;
}

```

## 12.1.2 Writing Functions for String Processing

In the last subsection, we studied how to perform basic string operations using loops. In most real-life programs, we will need to perform string operations on more than one string. Hence, it is a good idea to write functions to perform such operations.

A string processing function will accept one or more strings passed to it as arguments. Each string argument is accepted in a parameter of type `char []` or `char *`, the latter being more commonly used. Note that if the function modifies such a parameter, the argument string is also modified.

In some cases, however, it is not desirable to modify the argument string. If the function does not need to (or is not supposed to) modify the argument string, it is good programming practice to accept such string arguments in parameters of type `const char *`. Such parameters cannot be modified in the function and any attempts, intentional or otherwise, to modify such parameters will be caught by the compiler.

As we know, the standard C library provides several functions to manipulate strings. The declarations of these functions are provided in the `string.h` header file. These functions are covered in Section 12.2. Before we develop a function, it is a good idea to check if it is already available in the library. *Do not reinvent the wheel*. Moreover, the library functions have been tested thoroughly and generally have efficient implementations.

### Example 12.2 Writing functions for string processing

In this example, we study how to write functions for processing strings. Some examples given below deviate from the rule of not reinventing the wheel and rewrite some of the library functions mainly to understand how such functions are written and also to have a better insight into their operation. To avoid the accidental reuse of standard library function names (which are of the form `strxxx`), we have named our functions `str_xxx`.

#### a) Function to determine the length of a string

The `str_len` function given below uses the logic applied in Example 12.1a to determine the length of a given string and return it as an integer value. Since the argument string is not to be modified in this function, it is accepted in a parameter of type `const char *`.

```
int str_len(const char *a)
{
    int i = 0;

    while (a[i] != '\0')
        i++;
    return i;
}
```

#### b) Function to copy a string

In `str_cpy` function given below, the second string (`b`) is copied to first (`a`), which is consistent with the `strcpy` function from the C standard library. As the source string `b` should not be modified in this function, it is declared to be of type `const char *`. However, the target string `a` is declared to be of type `char *` as it will be modified during the copy operation.

```
char *str_cpy(char *a, const char *b)
{
    int i = 0;
    while ((a[i] = b[i]) != '\0') /* copy char and test it */
        i++;
    return a;
}
```

Note that the modifications to the destination string, i. e., `a`, will actually be written to the corresponding argument string passed to this function, making string copy available in the calling function. Thus, the function is expected to have `void` as its return type. It may, however, come as a surprise that the function has a return type of `char *` and returns the copied string `a` explicitly to the calling function. However, returning a pointer to the result string allows this function to be called from within another function call, as shown below.

```
printf("%s", str_cpy(str1, str2));
len = strlen(str_cpy(s1, s2));
```

### c) Function to reverse a string

The function `str_rev` to perform an in-place reversal of a specified string is given below. Since the argument string is to be modified, the function accepts it in parameter `a` of type `char *`. As usual, the function returns a pointer to the result string.

```
char *str_rev(char *a)
{
    int i, j;

    /* locate last character in string a */
    for(j = 0; a[j] != '\0'; j++)
        ;
    j--; /* j points to last char in string a */

    /* reverse string by exchanging characters at pos i and j */
    for (i = 0; i < j; i++, j--) { /* i moves forward, j backward*/
        /* exchange a[i] and a[j] */
        char t = a[i];
```

```

        a[i] = a[j];
        a[j] = t;
    }

    return a;
}

```

To reverse a given string, we use the logic presented in Example 9.4e to reverse an array of numbers. However, the difference is that here we do not know the position of the last character in the string. Hence, a `for` loop is used to first locate this. When the `for` loop is complete, the variable `j` is positioned at the null terminator. Hence, it is decremented to point to the last character.

Now we use another `for` loop to perform the string reversal. Within this loop, variable `i` moves forward, starting from the first character, whereas variable `j` moves backwards, starting from the last character. For each pair of values of `i` and `j`, we exchange characters `a[i]` and `a[j]`. This process continues as long as the value of `i` is less than that of `j`.

#### d) Function to compare strings

The `str_cmp` function given below compares two strings. It accepts the strings in parameters `a` and `b` of type `const char *` as these strings should not be modified and returns an `int` value as follows: -1 if `a < b`, 1 if `a > b` and 0 if `a == b`.

```

int str_cmp(const char *a, const char *b)
{
    /* compare strings char by char as long as both strings
       contain a non-null char at position i */
    int i = 0;
    while (a[i] != '\0' && b[i] != '\0') {
        if (a[i] < b[i])
            return -1;           /* a < b */
        else if (a[i] > b[i])
            return 1;           /* a > b */
        i++;
    }
    /* both strings are equal up to character position i-1 */
    if (a[i] == '\0' && b[i] == '\0') /* both strings exhausted */
        return 0;                 /* a == b */
    else if (a[i] == '\0') /* string a exhausted */
        return -1;           /* a < b */
    else return 1;          /* string b exhausted, a > b */
}

```

To compare two strings, we compare their contents, one character at a time, starting from the first character. This comparison should stop when we reach the null terminator in either or both the strings. Thus, a `while` loop is set up, with `i` as the loop variable, to compare characters in the given strings as long as both strings contain a non-null character at position `i` (i. e., `a[i] != '\0' & & b[i] != '\0'`). We compare characters `a[i]` and `b[i]` within this loop and return `-1` if `a[i] < b[i]` and `1` if `a[i] > b[i]`.

The `while` loop terminates when we reach the end of either or both the strings. Note that since the function did not return from within the `while` loop, all the characters compared so far are equal. Thus, if we have reached the end of both the strings, they are obviously equal and we return a zero value. Otherwise, if we have reached the end of string `a`, it is the shorter of the two strings and we return `-1`. Otherwise, string `a` is the larger of the two strings and we return `1`.

The `strcmp` standard library function uses different return values which allow a concise and efficient implementation. Instead of returning either `-1` or `1` when strings differ, the function returns a negative value if `a < b` and a positive value if `a > b`. The `str_cmp` function given above has been rewritten below using these return values.

```
int str_cmp(const char *a, const char *b)
{
    int i = 0;

    while (a[i] != '\0' && b[i] != '\0') {
        if (a[i] != b[i])
            return a[i] - b[i];
        i++;
    }

    return a[i] - b[i];
}
```

The `while` loop is set up as before. The code within this loop has been modified to return the difference `a[i]-b[i]` if they are not equal. Thus, if `a[i] < b[i]`, a negative value is returned and if `a[i] > b[i]`, a positive value is returned as desired.

The code after the `while` loop has also become very concise in this implementation. It simply returns the value of `a[i] - b[i]`. Thus, if we have reached the end of both the strings, both `a[i]` and `b[i]` are null characters and the function returns zero value as expected. Otherwise, the function returns a negative value if `a < b` (as `a[i]` is a null character) or a positive value if `a > b` (as `b[i]` is a null character) as desired.

## Program 12.2 Determine frequency of letters A ... Z in a given string

Write a program to determine frequencies of letters A ... Z in a given string, ignoring case.

**Solution:** Let us write a function `str_letter_freq` to determine the frequencies of letters in a given string and function `print_freq` to display these frequencies on the screen. The complete program in which these functions are called from the `main` function is given below

```
/* Determine frequencies of letter A to Z in a given string */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void str_letter_freq(const char *str, int freq[]);
void print_freq(int freq[]);
int main()
{
    char str[] = "He came, he saw, he conquered";
    int freq[26];

    printf("Given string: %s\n", str);
    str_letter_freq(str, freq);
    print_freq(freq);

    return 0;
}

/* calculate frequency of letters in a given string */
void str_letter_freq(const char *str, int freq[])
{
    int i;

    /* initialize elements of freq array to zero */
    for(i = 0; i < 26; i++)
        freq[i] = 0;
    /* calculate freq of letters */
    for (i = 0; str[i] != '\0'; i++) {
        int ch = toupper(str[i]);      /* convert str[i] to uppercase */
        if (isupper(ch))            /* if ch is a uppercase letter */
            freq[ch - 'A']++;       /* increment count */
    }
}

/* print frequency */
void print_freq(int freq[])
{
```

```

int i;
printf("Letter frequencies:\n");
/* print letters A to Z on a line */
for(i = 'A'; i <= 'Z'; i++)
    printf("%3c", i);
printf("\n");
/* print freq counts below corresponding letters */
for(i = 0; i < 26; i++)
    printf("%3d", freq[i]);
printf("\n");
}

```

The program output is given below.

```

Given string: I came I saw I conquered
Letter frequencies:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
2 0 2 1 6 0 0 3 0 0 0 0 1 1 1 0 1 1 1 0 1 0 1 0 0 0

```

## 12.2 Library Functions for String Processing

The C language provides a large number of functions to process strings, which includes 22 standard library functions. These functions are listed in Table 12.1 (the commonly used functions are shown in boldface).

### 12.2.1 ANSI C Standard Library Functions for String Processing

Table 12.2 summarizes the ANSI C standard library functions. The commonly used string manipulation functions, namely, **strcpy**, **strcat**, **strlen** have been explained in detail in Section 4.4.3.

**Table 12.1** *String processing functions in C language*

Category	Standard library functions	Other functions
String copy	<b>strcpy</b> , <b>strncpy</b> , <b>strxfrm</b> , <b>memcpy</b> , <b>memmove</b>	<b>stpcpy</b> , <b>memccpy</b>
String concatenation	<b>strcat</b> , <b>strncat</b>	
String comparison	<b>strcmp</b> , <b>strncmp</b> , <b>strcoll</b> , <b>memcmp</b>	<b>stricmp</b> , <b>strnicmp</b> , <b>strcmpli</b> , <b>strncmplti</b> , <b>memicmp</b>
String search	<b>strchr</b> , <b>strrchr</b> , <b>strstr</b> , <b>strspn</b> , <b>strcspn</b> , <b>strpbrk</b> , <b>strtok</b> , <b>memchr</b>	
Case conversion		<b>strlwr</b> , <b>strupr</b>
Error Handling	<b>strerror</b>	<b>_strerror</b>
Miscellaneous	<b>strlen</b> , <b>memset</b>	<b>strset</b> , <b>strnset</b> , <b>strdup</b> , <b>strrev</b> , <b>setmem</b>

## String Copy Functions

The **strcpy function** copies one string to another. All the characters in the source string, including the null terminator, are copied to the target string. The function returns the target string. Its prototype is given below.

```
char *strcpy(char *dest, const char *src);
```

The **strcpy function**, on the other hand, copies  $n$  initial characters, at the most, from one string to another and returns the target string. Its prototype is given below.

```
char *strncpy(char *dest, const char *src, size_t1 n);
```

If a null terminator is encountered in the source string before  $n$  characters are copied, it is copied to *dest*. Otherwise, the null terminator is not appended to *dest*. Hence, the programmer must be careful.

The **strxfrm function** transforms a string into another string for no more than  $n$  characters. Its prototype is given below.

```
size_t strxfrm(char *dest, char *src, size_t n);
```

This function returns the number of characters in the resulting transformed string not including the null terminator.

**Table 12.2** String processing functions in the ANSI C standard library

Category	Function	Typical call	Description
Copy	strcpy	strcpy(dest, src)	Copies string <i>src</i> to <i>dest</i> ; returns pointer to <i>dest</i>
	strncpy	strncpy(dest, src, n)	Copies the first <i>n</i> characters at the most from <i>src</i> to <i>dest</i> ; returns pointer to <i>dest</i>
	strxfrm	strxfrm(s1, s2, n)	Transforms string <i>s2</i> into string <i>s1</i> for no more than <i>n</i> characters; returns length of <i>s1</i>
Concatenation	strcat	strcat(dest, src)	Concatenates string <i>src</i> to <i>dest</i> ; returns pointer to <i>dest</i>
	strncat	strncat(dest, src, n)	Concatenates the first <i>n</i> characters at the most from string <i>src</i> to <i>dest</i> ; returns pointer to <i>dest</i>
Comparison	strcmp	strcmp(s1, s2)	Compares strings <i>s1</i> and <i>s2</i> (case-sensitive)
	strncmp	strncmp(s1, s2, n)	Compare the first <i>n</i> characters at the most from strings <i>s1</i> and <i>s2</i> (case-sensitive)
Search	strchr	strchr(s, c)	Finds the first occurrence of character <i>c</i> in string <i>s</i>
	strrchr	strrchr(s, c)	Finds last occurrence of character <i>c</i> in string <i>s</i>
	strstr	strstr(s1, s2)	Searches for sub-string <i>s2</i> in string <i>s1</i>
	strspn	strspn(s1, s2)	Returns the length of the initial segment of string <i>s1</i> that consists of characters entirely from string <i>s2</i>
	strcspn	strcspn(s1, s2)	Returns the length of the initial segment of string <i>s1</i> that consists entirely of characters not in string <i>s2</i>
	strpbrk	strpbrk(s1, s2)	Finds the first occurrence of any character from string <i>s2</i> in string <i>s1</i>
	strtok	strtok(s1, s2)	Scans string <i>s1</i> for the first token not contained in string <i>s2</i>
	strlen	strlen(s)	Returns the length of string <i>s</i>
Operations with memory	memcpy	memcpy(dest, src, n)	Copies a block of <i>n</i> bytes from <i>src</i> to <i>dest</i>
	memmove	memmove(dest, src, n)	Copies a block of <i>n</i> bytes from <i>src</i> to <i>dest</i>
	memcmp	memcmp(s1, s2, n)	Compares the first <i>n</i> bytes of two string
	memchr	memchr(s, c, n)	Searches for the first occurrence of character <i>c</i> in a memory block <i>s</i> of <i>n</i> bytes
	memset	memset(s, c, n)	Sets the first <i>n</i> bytes in a block of memory to character <i>c</i>

## String Concatenation Functions

The `strcat` function concatenates string *src* to *dest*. The `strncat` function, on the other hand, appends *n* characters, at the most, of string *src* to *dest*. Their prototypes are given below.

```
char *strcat (char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

The characters from string *src* are copied to *dest* starting from the position of the null terminator in string *dest*. Note that both the functions return *dest*.

## String Comparison Functions

In addition to string copy and concatenation, we often require string comparison operations, e. g., to sort strings in alphabetical order. C language does not allow us to compare strings using the relational and equality operators (`<`, `>`, `<=`, `>=`, `==` and `!=`). Instead, the C standard library provides the `strcmp` and `strncmp` functions which perform case-sensitive string comparison, i. e., without ignoring case. The `strcmp` function compares entire strings, whereas the `strncmp` function compares at most the first  $n$  characters in the strings. These functions accept two strings as arguments and compare them, character by character, using the character codes in the machine's character set (usually the ASCII code). Recall that in ASCII code, all digits have smaller values than uppercase letters, which in turn have smaller values than the lowercase letters. Thus, the string "HELLO" is smaller than both "Hello" and "hello".

The `strcmp` and `strncmp` functions return a value as follows:  $<0$  if  $s1 < s2$ ,  $== 0$  if  $s1 == s2$  and  $>0$  if  $s1 > s2$ . Their prototypes are given below.

```
int strcmp (const char *s1, const char *s2);  
int strncmp(const char *s1, const char *s2, size_t n);
```

The `strcoll` function compares two strings according to the collating sequence set by `setlocale`. The value returned by this function is similar to that of the `strcmp` function. The prototype of this function is given below.

```
int strcoll (char *s1, char *s2);
```

In addition to these, several implementations of C language provide additional functions (`stricmp` and `strnicmp`) and macros (`strcmpli` and `strncmpli`) for case-insensitive comparison, i. e., comparison in which the character case is ignored and the uppercase and lowercase letters are considered equivalent. Thus, the strings "Hello", "HELLO", "hello", "HeLLO", etc. are all considered equal. The `stricmp` and `strnicmp` functions are similar to the `strcmp` and `strncmp` functions except that they ignore case during comparison.

### *The `strcmp` function*

A typical call to the `strcmp` function take the following form:

```
strcmp (s1, s2)
```

This function performs a case-sensitive comparison between strings  $s1$  and  $s2$  and returns an integer value as follows: negative number if  $s1 < s2$ , zero if  $s1 == s2$  and a positive number if  $s1 > s2$ . The return value indicates the lexicographic ordering of strings.

The comparison starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or the end of either or both the strings is reached.

During comparison, if a character in one string is smaller, that string is considered as the smaller string and the appropriate value is returned. However, if the characters being compared are equal, the comparison proceeds for the next character. Now, if the end of a string is encountered, that string is considered to be lexicographically smaller than the other string. However, if all characters in both strings are exhausted, with all previous characters being equal, the strings are considered to be lexicographically equal. This is illustrated below with the help of several examples.

First consider the comparison of strings "Hello" and "World". Initially, the characters 'H' and 'W' are compared. Since 'H' < 'W', string "Hello" is considered to be lexicographically smaller and the function returns a negative value. Note that the subsequent characters in strings are not compared in this case.

Next consider the comparison of strings "Help" and "Hello". As the first three characters in these strings are equal on comparison, the fourth character pair, 'p' and 'l', is compared. Since 'p' > 'l', the first string is lexicographically larger than the second string and the function returns a positive value.

Now consider the comparison of strings "Hell" and "Hello". The first four characters compare equal after which the characters in the first string are exhausted. Thus, string "Hell" is considered to be smaller and the function returns a negative value.

Finally, consider the comparison of two equal strings: "Hello" and "Hello". In this case, all the characters compare equal. After the comparison of letter 'o' in each string, the characters in both strings are exhausted. Thus, the strings are considered to be equal and the function returns a zero value.

### Example 12.3 Process strings using library functions

#### a) String comparison

The program segment given below reads two strings from the keyboard and compares them.

```
char str1[20], str2[20];
int res;

printf("Enter two strings: ");
scanf("%s %s", str1, str2);

res = strcmp(str1, str2);
if (res < 0)
    printf("String \"%s\" is smaller than \"%s\"\n", str1, str2);
else if (res > 0)
    printf("String \"%s\" is greater than \"%s\"\n", str1, str2);
else printf("String \"%s\" is smaller than \"%s\"\n", str1, str2);
```

The strings are accepted in character arrays `str1` and `str2`. Then the `strcmp` function is used to compare them (case sensitive comparison) and store the return value in variable `res`. The value of `res`

is then tested using an `if-else-if` statement and an appropriate message string is printed. The output of the program containing this code segment is shown below.

```
Enter two strings: wonder Wonderful
String "wonder" is greater than "Wonderful"
```

This output is correct as the word `wonder` starts with a lowercase `w`, whereas the word `Wonderful` starts with an uppercase `W`.

### b) Function to swap (i. e. exchange) strings

The function `str_swap` that uses a well-known three-step procedure to exchange the contents of two strings is given below.

```
void str_swap(char *str1, char *str2)
{
    char temp[50];

    strcpy(temp, str1);
    strcpy(str1, str2);
    strcpy(str2, temp);
}
```

The function accepts two strings, `str1` and `str2`, as parameters of type `char *` and exchanges their contents. Note that the `strcpy` function is used to copy the strings and a local character array is used to store string `str1` temporarily. This function can be called from the `main` function to exchange strings as shown below.

```
int main()
{
    char s1[10] = "Hi";
    char s2[10] = "Bye";

    str_swap(s1, s2);
    printf("s1: %s s2:%s", s1, s2);
    return 0;
}
```

Note that this function has several limitations. First, the function uses a very inefficient approach to exchange strings. Second, the array representing each string should have enough space to accommodate the other string being copied to it and third, the length of the strings being exchanged is limited by the size of the `temp` array.

### c) Efficient function to swap (i. e. exchange) strings

As we know, a string is represented as a pointer to `char`. Fig. 12.1 illustrates a more efficient approach to exchange two strings that overcomes the problems explained above. In this approach, instead of exchanging the contents of strings, we simply exchange the string pointers.



**Fig. 12.1** Efficient approach to swap strings: (a) two strings (b) after exchanging the string pointers

To exchange string pointers, we need to pass pointers to these pointers. Thus, the parameters to function `pstr_swap` are of type `char **`, i. e., pointer to pointer to `char`. Within the function, `*str1` and `*str2` refer to string pointers. We exchange them using a temporary variable `temp`.

```
void pstr_swap(char **str1, char **str2)
{
    char *temp = *str1;
    *str1 = *str2;
    *str2 = temp;
}
```

When this function is called from the `main` function, we pass the addresses of strings `s1` and `s2` as shown below.

```
int main()
{
    char s1[] = "Hi";
    char s2[] = "Bye";

    str_swap(&s1, &s2);
    printf("s1: %s s2:%s", s1, s2);
    return 0;
}
```

## String Search Functions

Another category of operations that programmers often have to perform on strings is to search for occurrences of specific characters and sub-strings. The C standard library provides a rich set of functions to perform such search operations. These include `strchr`, `strrchr`, `strstr`, `strspn`, `strcspn`, `strpbrk`, and `strtok`.

The **strchr** and **strrchr** functions find the first and last occurrence, respectively, of a specified character in a string. The **strstr** function, on the other hand, finds the first occurrence of one string in another string.

The **strspn** function returns the length of the initial segment of a string that consists of characters entirely from another string. The **strcspn** function, on the other hand, returns the length of the initial segment of a string that consists entirely of characters not in another string. The **strpbrk** function is used to search in one string the first occurrence of any character in the other string. The **strtok** function is very useful and convenient for separating the tokens in a given string. It considers a string as a sequence of zero or more text tokens, separated by spans of one or more characters specified in another string. The **strspn**, **strcspn**, **strpbrk**, and **strtok** functions are discussed in the *Advanced Concepts* section.

### ***The **strchr** and **strrchr** functions***

The **strchr function** finds the first occurrence of a specified character in a string, whereas the **strrchr function** finds the last occurrence of a specified character in a string. If the specified character is found, the function returns a pointer to that character; otherwise, it returns a **NULL** pointer. The prototypes of these functions are given below.

```
char *strchr (const char *s, int c);
char *strrchr(const char *s, int c);
```

Combined with a loop, we can use these functions to count the occurrences of a specific character, to replace all occurrences of a specific character with some other character, etc.

### ***The **strstr** function***

It is also possible to search for a sub-string in a given string. The **strstr function**, whose prototype is given below, finds the first occurrence of sub-string **s2** in string **s1**.

```
char *strstr(const char *s1, const char *s2);
```

If substring **s2** is found, the function returns a pointer to it; otherwise, it returns a **NULL** pointer. We can combine this function with a loop to find all the occurrences of one string in another. However, replacing one or more occurrences of a sub-string with another string is not as easy as replacing a character unless the replacement string is of same length as the sub-string being replaced. Unfortunately, the C library does not provide any function to replace a sub-string with another string.

#### **Example 12.4 Searching Strings**

- a) Search a character in a string

The program segment given below reveals whether a C expression (provided as a string) is a parenthesized expression or not, i. e., whether it contains any parentheses or not.

```
char expr[] = "(a + b) / (c -d)";
if (strchr(expr, '(') != NULL)
    printf("Parenthesized expression\n");
else printf("Not a parenthesized expression\n");
```

Note that the `strchr` function is called from the test expression of an `if-else` statement. If a left parenthesis '(' is found in string `expr`, the function returns a pointer to it, i. e., a non-null pointer and the message `Parenthesized expression` is displayed; otherwise, the `strchr` function returns a `NULL` pointer and the message `Not a parenthesized expression` is displayed.

As a non-null pointer will be considered as a true expression, the above `if-else` statement can be written as follows:

```
if (strchr(expr, '('))
    printf("Parenthesized expression\n");
else printf("Not a parenthesized expression\n");
```

#### b) Search a sub-string in a given string

The example given below uses the `strstr` function to search a string in another string and display an appropriate message.

```
char quote[] = "An apple a day keeps doctor away";

if (strstr(quote, "apple"))
    printf("Search string found\n");
else printf("Search string not found\n");
```

This example is very similar to the previous example of searching for a character in a string and is self-explanatory.

#### c) Convert a date string from the dd-mm-yyyy format to the dd/mm/yyyy format

The program segment given below converts a date string from the *dd-mm-yyyy* format to the *dd/mm/yyyy* format.

```
char date[] = "8-12-2006";
char *ptr = strchr(date, '-'); /* ptr points to first '-' in date */
while (ptr != NULL) { /* while there is a '-' in date string */
    *ptr = '/'; /* replace it with a '/' */
    ptr = strchr(ptr, '-'); /* point ptr to next '-' in date str */
```

```
}
```

Initially, the `strchr` function is used to locate the position of the first '-' character in `date` and its address is assigned to pointer `ptr`. Then a `while` loop is set up to replace all '-' characters with '/'. Within the loop, the character '-' (pointed to by `ptr`) is first replaced by '/' and `ptr` is moved to the next '-' character as long as `ptr` is not null, i. e., as long as there is another '-' character in the string.

We can rewrite this code to replace characters in a concise manner, as shown below.

```
ptr = date;
while ((ptr = strchr(ptr, '-')) != NULL) /* while there is a '-' */
    *ptr = '/'; /* replace it with a '/' */
```

We can further simplify this code by eliminating the comparison with `NULL` as shown below:

```
ptr = date;
while (ptr = strchr(ptr, '-')) /* while there is a '-' */
    *ptr = '/'; /* replace it with a '/' */
```

#### d) Function to replace all occurrences of a character in a string with another character

In the previous example, we learnt how to convert a date string in the *dd-mm-yyyy* format to the *dd/mm/yyyy* format. If such date conversions are required often, it is a good idea to write a function for it. In fact, we may come across other similar situations where all the occurrences of a specific character in a given string must be replaced by another character. The function `str_replace_char` given below does just that.

```
char *str_replace_char(char *str, char ch, char rep_ch)
{
    char *ptr = str;
    while (ptr = strchr(ptr, ch)) /* while there is a ch */
        *ptr = rep_ch; /* replace it with a '/' */
    return str;
}
```

This function accepts a string and two characters `ch` and `rep_ch`. All the occurrences of character `ch` in string `Str` are replaced by character `rep_ch`. The function returns the modified string so that we can call this function from other function calls as illustrated below.

```
char date[] = "8-12-2006
char msg[] = "I came I saw I conquered";

puts(str_replace_char(date, '-', '/'));
puts(str_replace_char(msg, ' ', '\t'));
```

## 12.3 Advanced Concepts

### 12.3.1 Nesting String Manipulation Functions

We know that strings are basically character arrays and are passed to a function using the pass by reference mechanism. Thus, if we modify a string parameter in a called function, the corresponding argument string in the calling function is modified too, i. e., the modifications to the parameter strings are available in the calling function. Thus, there is no need to explicitly return the result string from a string manipulation function. However, several string manipulation functions return a pointer to the result string. Consider the prototypes of `strcpy`, `strcat`, `strlwr` and `strupr` functions given below.

```
char *strcpy(char *dest, const char *src);
char *strcat(char *dest, const char *src);
char *strlwr(char *s);
char *strupr(char *s);
```

It is easy to understand that returning a pointer to the result string allows the result of one function call to be passed to another by nesting string manipulation function calls. This makes the program concise. For example, given the first name (`fname`) and last name (`lname`) of a person, we can use `strcpy` and `strcat` functions to obtain the full name and print it using the `printf` function, as shown below.

```
strcpy(full_name, fname);
strcat(full_name, " ");
strcat(full_name, lname);
printf("Your full name: %s\n", full_name);
```

Here, the values returned by `strcpy` and `strcat` are ignored. We can use these return values to write these function calls concisely, as shown below:

```
strcat(strcat(strcpy(full_name, fname), " "), lname);
```

Execution of this code starts from the innermost function call (`strcpy`) which copies string `fname` to `full_name` and returns a pointer to `full_name`. This pointer is then passed as the first argument to the inner `strcat` function, which appends a space to it (to `full_name`) and returns a pointer to it. This pointer (`full_name`) is finally passed as the first argument to the outer `strcat` function which appends last name to it.

Note that the above function calls can even be combined with the `printf` function call used to print the concatenated string, as shown below:

```
printf("Your full name: %s\n",
      strcat(strcat(strcpy(full_name, fname), " "), lname));
```

Here, the pointer returned by the outer `strcat` function (which is a pointer to `full_name` ) is passed as an argument to the `printf` function.

### 12.3.2 Avoiding Pitfalls in String Processing

We should be extremely careful while working with strings. In particular, we should not exceed the target array size and we should not forget the null terminators. These issues are discussed below.

#### Do Not Exceed Target Array Size

We already know that while writing a string in an array, we should not exceed the target array size. C compilers usually do not give any error in such situations. Such strings are likely to overwrite some other data in the program, causing unpredictable behaviour. Hence, it is the programmer's responsibility to ensure that the target array size is not exceeded while processing strings. Consider the program segment given below.

```
char str1[] = "Hello friends, See this magic!";
char str2[] = "Magic string";
char str3[5];

strcpy(str3, str1); /* copy str1 to str3 */
printf("str2: %s", str2);
```

This program segment copies string `str1` to `str3` and then prints another string `str2` which has been initialized to "Magic string". We obviously expect the program to print the output as "Magic string". However, when executed using Dev-C++, the program printed unexpected output: See this magic! which is actually a part of string `str1`. Note that the compiler did not report any error when the program was compiled and executed.

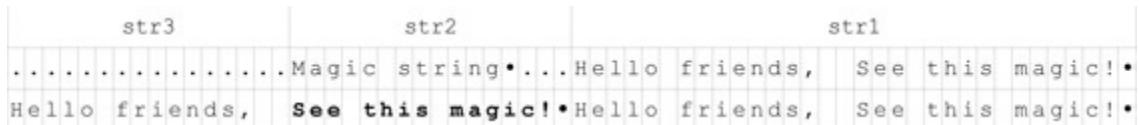
The program demonstrated this unexpected behaviour because we copied more characters to string `str3` than could be accommodated in it. Note that `str1` has 32 characters including the null terminator. However, there was provision for only five characters in the target string `str2`. Thus, when `str1` was copied to `str3`, it extended into the `str2` array, overwriting its contents. That's how the string "See this magic", which is part of `str1` was printed when we printed `str2`.

Next, the initializer of string `str1` was modified as "Hello friends" and the program was executed again. Now we know that this string also does not fit in array `str3` and the string copy operation will surely overwrite string `str2`. Thus, we now expect the program to display some portion of string `str1`. However, this time the program displayed the string Magic string, the original value of string `str2`. Yet another surprise!

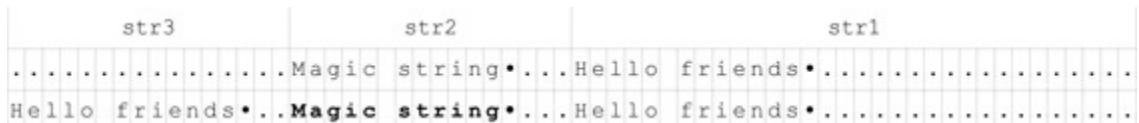
If this is the kind of surprises we get in such a small program, imagine what would happen in more complex programs or real-life applications. Moreover, since the compiler does not give any warning or error, debugging the program is going to be very difficult and time consuming. So remember: *Never*

*exceed the array size while processing strings.*

We can easily understand the behaviour of our program if we understand how memory is allocated to these arrays in Dev-C++. Dev-C++ allocates memory to character arrays in multiples of 16 bytes and the arrays in this example are allocated memory in the order: **str3**, **str2** and **str1**. Memory allocation and the effect of program execution on **str2** is shown below for the first situation discussed above. The unused character in a string is indicated by '.' and a null terminator by '•'.



The effect of second situation discussed above is illustrated below.



Finally, note that the behaviour of this program is compiler dependent. In Turbo C 2.0, it prints the string correctly but a memory exception error is reported and Turbo C is closed.

### Do Not Forget Null Terminators

Forgetting null terminators at the end of a string is another major problem which is also likely to give unpredictable results. C compilers do not give any warning or error in these situations as well. Hence programmers should take utmost care while writing strings. Consider the program segment given below.

```
char str1[] = "Hello friends, See another magic!";
char str2[] = "Magic string";
int i;

/* copy string str2 to str1 */
i = 0;
while (str2[i] != '\0') {
    str1[i] = str2[i];
    i++;
}
printf("str1: %s", str1);
```

This program segment copies string **str2** to **str1**. When this program is executed, we expect the text "str1: Magic string" to be displayed. However, the program displayed the output shown below:

str1: Magic strings, See another magic

This is because we intentionally did not include the null terminator in the copied string. Let us now modify the declaration of string `str1` as

```
char str[10];
```

and execute the program again. Now we get the output as

```
str1: Magic string@
```

### 12.3.3 Working with Words in a String

A text string usually comprises several words. In this section, we study how to process words in a given string. This includes various operations, namely, word counting, string capitalization and word separation.

#### Count the Number of Words in a String

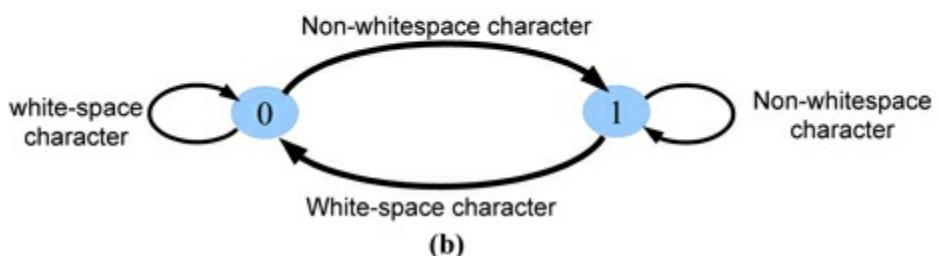
We assume that a word is a sequence of any non-whitespace character separated by whitespace (i. e., space, tab, newline, etc.). Thus, the string "hello 0123 \*%\$#@" contains three words. The counting of words is illustrated in Fig. 12.2.

Let us use variable `nword`, with zero initial value, as the word counter. Let us also use a `while` loop to process the given string, one character at a time and increment the word counter for each word encountered. Now the only question is how to identify words.

Let us use flag `in_word` to indicate whether the current character in the string is a part of a word or not. This flag has two values: 1 indicating that we are inside a word and 0 otherwise. Fig. 12.2b shows the state transition diagram which displays the two states corresponding to the values of `in_word`. The state with `in_word = 0` is the starting state as we are outside any word before the word count begins.

	M	a	y		G	o	d		B	l	e	s	s		Y	o	u	.	\0
in_word:	0	1		0	1		0	1		0	1		0	1		0	1		0
nword:	0	1		2		3		4											

(a)



**Fig. 12.2 Counting the words in a string: (a) Values of `in_word` flag and `nword` counter during processing of string, (b) State transition diagram**

In each state, we have two types of inputs that make a transition from one state to another —a whitespace character and a non-whitespace character. Thus, there are four transitions to be considered, as shown by arcs. The transitions shown by thick arcs are of interest as they cause a change in the *current state*:

- 1. `in_word` is 0 and the next character is not a whitespace. This marks the beginning of a new word. Hence, we set `in_word` to 1 and increment the word count.
- 2. `in_word` is 1 and the next character is a whitespace. This marks the end of the current word. Hence, we reset `in_word` to 0.

The function `str_wcount1` to count the words in a given string is given below.

```
/* count words in a given string */
int str_wcount1(const char *a)
{
    int nword = 0, in_word = 0, i;

    for(i = 0; a[i] != '\0'; i++) {
        if (in_word == 0) { /* if we are not in a word */
            if (!isspace(a[i])) { /* if a[i] not whitespace */
                in_word = 1; /* we are in a word now */
                nword++; /* increment word count */
            }
        }
        else if (isspace(a[i])) /* in a word & a[i] is whitespace */
            in_word = 0; /* we are outside word now */
    }

    return nword;
}
```

We can rewrite the above function as shown below.

```
/* count words in a given string */
int str_wcount2(const char *a)
{
    int nword = 0, in_word = 0, i;

    for(i = 0; a[i] != '\0'; i++) {
        if (isspace(a[i])) { /* if a[i] is a whitespace */
```

```

        if (in_word)          /* if we are in a word */
            in_word = 0;      /* we are outside any word */
    }
    else if (in_word == 0) { /* not whitespace & not in word */
        in_word = 1;        /* we are in a word now */
        nword++;           /* increment word count */
    }
}

return nword;
}

```

Note that when we encounter a whitespace, we are not in a word any longer. Thus, it is not necessary to test whether we are in a word to reset `in_word` to 0. The modified `for` loop is given below.

```

for(i = 0; a[i] != '\0'; i++) {
    if (isspace(a[i]))      /* if a[i] is a whitespace */
        in_word = 0;        /* we are outside any word */
    else if (in_word == 0) { /* not whitespace & not in word */
        in_word = 1;        /* we are in a word now */
        nword++;           /* increment word count */
    }
}

```

### Convert a String to Capitalized Case

The C standard library provides two case conversion functions, `strupr` and `strlwr`, to convert a string to uppercase and lowercase, respectively. However, it does not provide a function to convert a string to *capitalized case*, in which the first letter of each word is an uppercase letter and the remaining letters are in lowercase, as in `This String Illustrates Capitalized Case`".

To convert a string to capitalized case, we need to identify the word boundaries in it and then convert the case of the letters depending on their position (convert the first letter in each word to uppercase and the remaining letters to lowercase).

The `str_wcount` function in the previous example counts the number of words in a given string. We can modify it to convert the string to capitalized case, as shown below.

```

/* convert a given string to capitalized case */
char *str_cap(char *a)
{
    int in_word = 0, i;

    for(i = 0; a[i] != '\0'; i++) {
        if (in_word == 0) {

```

```

        if (!isspace(a[i])) {
            in_word = 1;
            a[i] = toupper(a[i]);
        }
    }
    else {
        if (isspace(a[i]))
            in_word = 0;
        else a[i] = tolower(a[i]);
    }
}

return a;
}

```

Observe that when we encounter a new word (i. e., statement `in_word = 1`), we convert the letter being processed (`a[i]`) to uppercase using the `toupper` function. The second and subsequent letters in a word are identified by a thin arc transition at state 1 in Fig. 12.2b, for which we have to convert the letter being processed to lowercase. This is done by adding an `else` clause to the `if (isspace(a[i]))` statement.

### Program 12.3 Separate words in a given string

Write a program to separate the words in a given string.

0	W	i	s	h	\0				
1	y	o	u	\0					
2	a	l	l	\0					
3	t	h	e	\0					
4	b	e	s	t	\0				
5									
6									

**Fig. 12.3** Separating the words in a string "Wish you all the best" using a two-dimensional array of size  $7 \times 10$

**Solution:** Let us write function `str_words` to accept a string and separate the words in it. The number of words in a given string as well as the number of characters in each word are not known before hand. To keep the code simple, let us assume that the given string contains at the most `MAX_WORD` words and each word in turn contains at the most `MAX_CHAR` characters. Thus, the function `str_words` can return the words in the given string in a two-dimensional array of size `MAX_WORD x MAX_CHAR` of type `char`, as shown in Fig. 12.3.

Observe the null terminators at the end of each word. Note that the operations involved in separating the words are similar to those used in the `str_wcount` function used to count the words in a given string. Hence, we modify the `str_wcount` function to separate the words and write them to the two-dimensional array parameter.

To simplify the things further, we have assumed that the given string does not contain any punctuation symbols and that the words are separated by whitespaces. Although this is a very restrictive assumption, it helps us focus on the process of word separation. In Program 12.4, we study how the words in a text containing punctuation symbols can be separated using the C standard library functions. The complete program is given below.

```
/* Separate words from a given string */
#include <stdio.h>
#include <string.h>
#include <conio.h>

#define MAX_WORD 50
#define MAX_CHAR 20

int str_words(const char *a, char words[][MAX_CHAR]);
void init_arr(char words[][MAX_CHAR]);

int main()
{
    char *quote = "A friend in need is a friend indeed";
    char words[MAX_WORD][MAX_CHAR];
    int nwords, i;

    nwords = str_words(quote, words);

    printf("Given string: %s\n\n", quote);
    printf("%d words in given string:\n", nwords);
    for(i = 0; i < nwords; i++)
        printf(" %s\n", words[i]);
    return 0;
}

/* separate words in a given string */
int str_words(const char *a, char words[][MAX_CHAR])
{
    int in_word; /* flag - inside a word or outside it? */
    int w, c;     /* position in words array for next char from string
    int i;
```

```

w = -1;
in_word = 0; /* we are not in a word */
for(i = 0; a[i] != '\0'; i++) {
    if (in_word == 0) { /* we are not in a word */
        if (!isspace(a[i])) { /* cur char is not a white space */
            in_word = 1; /* we are in a new word now */
            c = 0;
            words[+w][c] = a[i];/* write first char of word in
                                 words array */
        }
    }
    else if (isspace(a[i])) { /* in a word & whitespace encountered */
        in_word = 0; /* we are outside word now */
        words[w][++c] = '\0'; /* write null terminator to word */
    }
    else words[w][++c] = a[i]; /* add next char to words array */
}
words[w][++c] = '\0'; /* write null terminator to last word */
return w + 1; /* number of words in given string */
}

```

The variables `w` and `c` in `str_words` function are used to indicate the position in `words` array where the next non-whitespace character from a given string will be written. The variable `w` is initialized to `-1`. When a new word is encountered, the values of variables `W` and `C` are updated to point to the next word in `words` array (`w` is incremented and `C` is reset to `0`). Also, when subsequent characters in a word are encountered, the variable `C` is incremented. The transition from one word to the next is important as we have to write a null terminator to mark the end of the current word in `words` array. When string processing is complete, a null terminator is written at the end of the last word in `words` array. Finally, the function returns the number of words in the given string. The program output is given below.

```

Given string: A friend in need is friend indeed
8 words in given string:
A
friend
in
need
is
a
friend
indeed

```

This `str_words` function has a limitation in that it will not work correctly when the limits of

`MAX_WORD` and `MAX_CHAR` are exceeded. We can modify the function to check these limits before writing the next character into `words` array. Thus, we require four `if` statements to print an error message and exit the program if the limits are exceeded. To avoid repetitive coding and to keep the code short, we can define a function to check the array limits and safely write a character to `words` array, as shown below.

```
void add_char_to_words(char ch, char words[][MAX_CHAR], int w, int c)
{
    if (w >= MAX_WORD) {
        printf("Error: Word limit exceeded ...\\n");
        exit (1);
    }

    if (c >= MAX_CHAR) {
        printf("Error: char limit in word %d exceeded ...\\n", w+1);
        exit(1);
    }
    words[w][c] = ch;
}
```

This function will now replace all the statements in `str_words` that write the next character to `words` array. For example, the statement

```
words[w][++c] = a[i];
```

will be replaced by

```
add_char_to_words(a[i], words, w, ++c);
```

#### 12.3.4 ANSI Functions for Advanced String Processing

This section discusses advanced standard library functions for string processing. These include the string search functions `strspn`, `strcspn`, `strpbrk`, `strtok`, `mem...` functions (`memcpy`, `memmove`, `memcmp`, `memchr` and `memset`) and error handling function (`strerror`).

##### String Search Functions

The **`strspn` function**, whose prototype is given below, returns the length of the initial segment of string `s1` that consists of characters entirely from `s2`.

```
size_t strspn(const char *s1, const char *s2);
```

Thus, the function call `strspn("5678+1234", "01234567890")` returns 4, the length of string

"5678" which is composed entirely of characters in string "0123456789".

The **strcspn function**, on the other hand, returns the length of the initial segment of string *s1* that consists entirely of characters not in *s2*.

```
size_t strcspn(const char *s1, const char *s2)
```

Thus, the function call `strcspn("5678+12*23", "+-*%")` returns 4, the length string "5678" which is composed entirely of characters not in string "+-\*%".

The **strupbrk function** is used to search in one string the first occurrence of any character in the other string. Its prototype is given below.

```
char *strupbrk(const char *s1, const char *s2);
```

This function scans string *s1* for the first occurrence of any character in string *s2*. If a character is found, it returns a pointer to it; otherwise, a null pointer is returned.

The **strtok function** is very useful and convenient for separating the tokens in a given string. Its prototype is given below.

```
char *strtok(char *s1, const char *s2);
```

This function considers string *s1* as a sequence of zero or more text tokens, separated by spans of one or more characters specified in string *s2*, e. g., in the function call

```
strtok(" One Two, 3/4;Five-6", " ,;")
```

the first string will be interpreted as having four tokens ("One", "Two", "3/4" and "Five-6") as they are separated by one or more separator characters specified in the second string.

The usage and working of this function is slightly more involved compared to other functions in the library. Each call to `strtok` separates one token from the specified string. Thus, we will usually use the `strtok` function in conjunction with a loop to separate all the tokens.

To separate the tokens in a given string, say *s1*, we first call the `strtok` function with string *s1* specified as its first argument. If string *s1* contains at least one token separated by the characters from the second argument string, the function writes a null character immediately after the first token and returns a pointer to the beginning of this token. Thus, the function call given above writes a null character after the first token, i. e., "One", as in "one\0Two, 3/4 Five-6" and returns a pointer to character 'O' in token "One". Now we can use this pointer to operate on the first token. Remember that the characters after the null terminator are not part of the token string.

To separate each subsequent token in this string, we call the `strtok` function with the NULL pointer as the first argument. If a token is available in the string, the function writes a null terminator immediately after it and returns a pointer to its beginning. Thus, after all the tokens are separated, the

contents of the first string in the above example will be "One\0 Two\0 3/4\0Five-6".

When no token is available in the specified string, the `strtok` function returns a `NULL` pointer which is usually used to terminate the token separation process.

Note that the separator string (`s2`) may be different for each subsequent call to the `strtok` function. This introduces a lot of flexibility in the interpretation of tokens in strings and also allows the `strtok` function to be useful in widely varying situations.

Since the modifications to the string being tokenized (i. e., string `s1`) cannot be undone easily, it is a good idea to pass a copy of the string rather than the string itself. The `strupr` function discussed shortly can be used to create a duplicate of a string for this purpose.

However, we have to remember to return the memory allocated to this string copy using the `free` function.

#### Program 12.4 Determine number of words and average word length in a given text string

Write a program to determine the number of words and average word length in a given string containing English text.

**Solution:** We may want to know the number of words in a given string and the average word length. The English text contains words (i. e., sequences of letters) separated by spaces, newlines and punctuation marks such as period, comma, semicolon, colon, quotation marks, exclamation, question mark, etc. It is quite a difficult job to separate the words from these punctuation marks. However, this task is greatly simplified when we use the powerful `strtok` function provided in the standard C library. The program given below determines the number of words in a given string and the average word length.

```
/* determine number of words and average word length in a given string
#include <stdio.h>
#include <string.h>

int main()
{
    char str[] = "Alexander said, \"I came, I saw, I conquered!\"";
    char punct_str[] = ".,:;!?'\""; /* punctuator list */
    char *tmp_str; /* copy of given string */
    char *wptr; /* pointer to a word */
    int nword, nchar; /* no of words and no of chars in all words */
    float avg_len; /* average word length */

    printf("Given string: %s\n", str);
    puts("\nWords in a given string:");
}
```

```

nword = nchar = 0;
tmp_str = strdup(str);           /* copy of given string */
wptr = strtok(tmp_str, punct_str); /* get ptr to first word */
while (wptr != NULL) {
    nword++;                   /* increment word count */
    nchar += strlen(wptr);     /* update character count */
    printf("%s ", wptr);       /* display current word */
    wptr = strtok(NULL, punct_str); /* get ptr to next word */
}

avg_len = (float) nchar / nword; /* calc average word length */
printf("\n\nTotal words: %d\n", nword);
printf("Average word length: %4.1f\n", avg_len);

free(tmp_str);      /* release memory allocated to tmp_str */
return 0;
}

```

The main function initializes character array `str` with a string literal. A character array `punct_str` is used as a string of punctuation characters and is initialized with the string literal " .;!:?"". This string is used by `strtok` to separate the words in string `str`. Since the `strtok` function modifies the string being processed, we create a duplicate of this string using the `strdup` function and operate on this string. A character pointer `tmp_str` is used to point to this string.

The `main` function first displays string `str`. Initially, the counters `nword` and `nchar` are initialized to 0 and the given string is duplicated and `tmp_str` is made to point to it. The `strtok` function is used to separate first word from string `tmp_str`. The character pointer `wptr` is used to point to this word. Then a `while` loop is set up to process the remaining words in string `tmp_str`.

In each iteration of the loop, first the `nword` and `nchar` counters are updated. Then the current word pointed to by `wptr` is displayed and the `strtok` function is called to separate the next word from string `tmp_str`. The program output is given below.

```

Given string: Alexander said, "I came, I saw, I conquered!"

Words in a given string:
Alexander said I came I saw I conquered

Total words: 8
Average word length: 4.0

```

## mem... Functions

The ANSI standard library provides several functions to operate with blocks of memory. These include `memcpy`, `memmove`, `memcmp`, `memchr` and `memset`.

The **memcpy** and **memmove** functions copy a block of  $n$  bytes from  $src$  to  $dest$ . Their prototypes are given below.

```
void *memcpy (void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```

If  $src$  and  $dest$  overlap, the behaviour of the `memcpy` function is undefined. However, the `memmove` function correctly copies the bytes in the overlapping locations. Note that both the functions return  $dest$ .

The `memcmp` function compares the first  $n$  bytes of two strings and returns an integer value, as in the `strcmp` function. Its prototype is given below.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

The **memchr** function searches the first  $n$  bytes of a memory block for a specified character. If the character is found, it returns a pointer to its first occurrence; otherwise, it returns null pointer. Its prototype is given below.

```
void memchr(const void *s, int c, size_t n);
```

The **memset** function sets the first  $n$  bytes of array  $s$  to character  $c$ . Its prototype is given below.

```
void *memset(void *s, int c, size_t n);
```

Note that the functionality provided by these `mem...` functions is available in various `str...` functions (`strncpy`, `strncmp`, `strchr` and `strnset`). Thus, a beginner can avoid using these functions and use the `str...` functions instead. However, `strnset` is not an ANSI function.

## Error Handling Function

The **strerror** function takes an error number and returns a pointer to the error message string associated with that error. Its prototype is given below.

```
char *strerror(int errnum);
```

### 12.3.5 Non-ANSI Functions for String Processing

Besides the ANSI-Standard functions discussed in the previous section, C compilers provide several other functions for string processing. Some of these are summarized in Table 12.3 and explained below.

**Table 12.3** Non-ANSI string processing functions in C library

Category	Function	Typical Call	Description
String copy	stpcpy	stpcpy( <i>dest, src</i> )	Copies a string; returns a pointer to the null character in the copied string
	stricmp	stricmp( <i>s1, s2</i> )	Compares strings <i>s1</i> and <i>s2</i> ignoring case
String comparison	strnicmp	strnicmp( <i>s1, s2, n</i> )	Compares at the most first <i>n</i> characters from strings <i>s1</i> and <i>s2</i> ignoring case
	strcmpi	strcmpi( <i>s1, s2</i> )	Macro with the same function as stricmp
	strnncmpi	strnncmpi( <i>s1, s2, n</i> )	Macro with the same function as strnicmp
Case conversion	strlwr	strlwr( <i>s</i> )	Converts a string to lowercase; returns a pointer to it
	strupr	strupr( <i>s</i> )	Converts a string to uppercase; returns a pointer to it
Error handling	_strerror	_strerror( <i>s</i> )	Builds a customized error message string
Operations with memory	memccpy	memccpy( <i>dest, src, c, n</i> )	Copies a block of <i>n</i> bytes from <i>src</i> to <i>dest</i> ; copying stops as soon as character <i>c</i> is copied
	memcmp	memcmp( <i>s1, s2, n</i> )	Compares the first <i>n</i> bytes of two strings, ignoring case
	setmem	setmem( <i>s, n, v</i> )	Sets first <i>n</i> bytes of memory block <i>s</i> to value <i>v</i>
Miscellaneous	strset	strset( <i>s, c</i> )	Sets all characters in a string to <i>c</i>
	strnset	strnset( <i>s, c, n</i> )	Sets first <i>n</i> characters of string <i>s</i> to <i>c</i>
	strrev	strrev( <i>s</i> )	Reverses a string and returns a pointer to it
	strdup	strdup( <i>s</i> )	Duplicates a string and returns pointer to it

## String Copy and Comparison Functions

The **stpcpy** function is similar to the **strcpy** function except that it returns a pointer to the null character in the target string. Its prototype is given below.

```
char *stpcpy(char *dest, const char *src);
```

The **stricmp** and **strnicmp** functions are similar to the **strcmp** and **strncmp** functions except that these functions compare strings ignoring case, i. e., lowercase and uppercase letters will be considered equivalent. The prototypes of these functions are given below.

```
int stricmp(const char *s1, const char *s2);
int strnicmp(const char *s1, const char *s2, int n);
```

The **strcmpi** and **strnncmpi** are macros that call the **stricmp** and **strnicmp** functions. They are provided for compatibility with other compilers. The declarations of these macros are similar to those of the corresponding functions and are given below.

```
int strcmpi(const char *s1, const char *s2);
```

```
int strncmpi(const char *s1, const char *s2, int n);
```

### Case Conversion Functions

The **strlwr function** converts a string to lowercase, whereas the **strupr function** converts a string to uppercase. All letters in the string are converted to their lowercase or uppercase representation, whereas other characters remain unaffected. The prototypes of these functions are given below.

```
char *strlwr(char *s);
char *strupr(char *s);
```

### Operations with Memory

The **memccpy function** is similar to the **memcpy** function except that while copying first  $n$  bytes of a block of memory from  $src$  to  $dest$ , the copying stops if character  $C$  is copied. The prototype of this function is given below.

```
char *memccpy(void *dest, const void *src, int c, size_t n);
```

The **memicmp function** compares the first  $n$  bytes of memory blocks  $s1$  and  $s2$ , ignoring case. It is similar to the **memcmp** function except that it ignores case. Its prototype is given below.

```
void memicmp(const void *s1, const void *s2, size_t n);
```

The **setmem** function sets the first  $n$  bytes of a memory block to a specified value. The prototype of this function is given below.

```
void setmem(void *dest, unsigned n, char v);
```

### Miscellaneous and Error Handling Functions

The **strset and strnset functions** set the characters in a string to a specified value. The **strset** function sets all the characters in string  $s$  to  $c$ , whereas the **strnset** function sets the first  $n$  characters in  $s$  to  $c$ . These functions return a pointer to string  $s$ . The prototypes of these functions are given below.

```
char *strset(char *s, int c);
char *strnset(char *s, int c, size_t n);
```

The **strrev function** reverses all the characters in a given string except the null terminator. Its prototype is given below.

```
char *strrev(char *s);
```

The **strupr function** creates a duplicate copy of a specified string. Its prototype is given below.

```
char *strdup(const char *s);
```

The **strdup** function allocates memory for string copy, copies string *S* to that memory and returns its address. It is the programmer's responsibility to free this memory (using the **free** function). However, if the required memory could not be allocated, i. e., if string duplication failed, the function returns a **NULL** value.

The **\_strerror** function builds a customized error message. Its prototype is given below.

```
char *_strerror(const char *s);
```

If *S* is null, the function returns a pointer to the most recently generated error message. Otherwise, it returns the most recently generated system error message preceded by the custom error message *s*.

### 12.3.6 Dynamic Memory Allocation

Thus far, we have used arrays to store strings. This approach leads to simple code but usually requires more memory as we have to make a provision to accommodate larger string likely to be encountered. Despite making this provision, there are usually some situations in which the strings encountered are larger than the arrays used and the program simply fails. The solution to this problem is to use dynamic memory allocation. However, this usually leads to more complex programs.

We have studied in last chapter that the C standard library provides three functions to allocate memory at run time: **malloc**, **calloc** and **realloc**. The **malloc** function allocates a block of memory of specified size and returns a pointer to it. The **calloc** function allocates a block of memory (*n* × *size*), initializes it to 0 and returns a pointer to it. The **realloc** function adjusts the size of the allocated memory block by copying its contents to a new location if necessary.

It is the programmer's responsibility to ensure that the memory allocated at run time is explicitly returned to the system using the **free** function; otherwise, memory leaks, where the memory which is not freed is unavailable for use until the system is rebooted, may occur.

#### Example 12.5 Using dynamic memory allocation for strings

##### a) Read a string from keyboard

The function **dstr\_read** given below reads a string from the keyboard into array **buf**, stores it in dynamically allocated memory and returns a pointer to it.

```
char *dstr_read(char *msg)
{
    char *str;
    char buf[100];           /* buffer */
    printf("%s", msg);
```

```

scanf("%s", buf); /* read string in buffer */

str = (char *) malloc(strlen(buf) + 1);
if (str == NULL) {
    printf("Error: out of memory ...\\n");
    exit(1);
}
strcpy(str, buf);
return str;
}

```

The function first reads a string from the keyboard into array **buf** of size 100. Then it allocates the exact memory required for this string using the **malloc** function and copies the string from the buffer to it. Finally, it returns a pointer to the string. As the size of buffer **buf** is set to 100, this function can be used to read strings such as the name of a person, email id, a line in an address, etc.

This function is used in the **main** function to read the first and last names of a person. Observe the calls to the **free** function to return to the system the memory used by strings **fname** and **lname**.

```

int main()
{
    char *fname, *lname;
    printf("Enter your name:\\n");
    fname = dstr_read("First name: ");
    lname = dstr_read("Last name: ");
    printf("Hi, %s %s\\n", fname, lname);

    free(fname);
    free(lname);
    return 0;
}

```

### b) Function to copy a string to another dynamic string

The function **dstr\_cpy** given below copies a given string to another whose memory is dynamically allocated and returns a pointer to the copy. The function first allocates the required memory for the string, copies the given string to that memory and returns its address.

```

char *dstr_cpy(char *str)
{
    char *temp;
    if ((temp = (char *) malloc(strlen(str) + 1)) == NULL) {
        printf("Error: out of memory ...\\n");

```

```

        exit(1);
    }
    strcpy(temp, str);
    return temp;
}

```

## Using an Array of Pointers to Store Strings

In several applications, we may be required to process an array of strings. For example, the names of students, employees, subjects, countries, etc. As a string is stored in an array of characters, we can use a two-dimensional array of characters to store an array of strings. For example, we can use a two-dimensional array named **fruits** of size  $4 \times 10$  to store the names of four fruits, as shown in Fig. 12.4a. Now, the  $i$ th fruit name can be accessed using the expression **fruits[i]**. Thus, the code given below prints the names of the fruits contained in the **fruits** array.

```

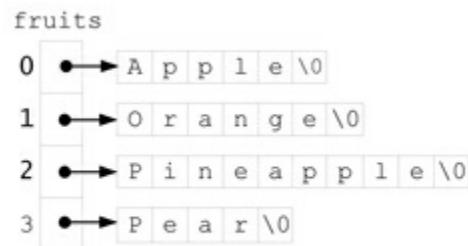
printf("Fruits: ");
for(i = 0; i < 4; i++)
    printf("%s ", fruits[i]);
printf("\n");

```

We know that each row in a two-dimensional array has the same number of elements. However, the strings in C language can be of arbitrary length. Thus, a two-dimensional array of characters is not suitable for storing an array of strings. It limits the maximum string length and/or wastes substantial memory. Thus, we observe a lot of wasted space in

fruits	
	0 A p p l e \0 1 O r a n g e \0 2 P i n e a p p l e \0 3 P e a r \0
0	A p p l e \0
1	O r a n g e \0
2	P i n e a p p l e \0
3	P e a r \0

(a)



(b)

**Fig. 12.4 Representing an array of strings: (a) using two-dimensional array of char (b) using array of pointers**

Fig. 12.4a (shown in grey). Also, the name **Watermelon** cannot be stored in this array, thereby severely limiting its use. Increasing the number of columns to accommodate longer fruit names further leads to wastage of memory. This problem is more serious when large variations in string lengths occur.

We also know that the memory for arrays is allocated at compile time (i. e., static memory allocation). Thus, we have to reserve memory for the maximum number of strings to be stored in that array. This leads to further wastage of memory when we actually have to store fewer strings.

A much better solution to this problem is to declare an array of pointers as

```
char *fruits[4];
```

The memory required to store the name of each fruit is dynamically allocated (using the `malloc` function), as shown in Fig. 12.4b. This data structure overcomes all the drawbacks of two-dimensional array representation. First, there is no wastage of memory (at the end of strings) as only the required memory is allocated for each string. Second, each string can be of arbitrary length and long strings do not increase memory wastage, as in case of two-dimensional arrays. Also, we can make a provision for a large number of strings (by increasing the size of the pointer array), without actually allocating memory for them. This further avoids memory wastage.

However, there is a small overhead to achieve these benefits. First, we require additional memory to store the pointers (2 or 4 bytes per pointer). Second, some execution time is required to allocate memory before a string is actually stored. Also, writing such programs involving dynamic memory allocation makes the code more complex.

Finally, note that we can initialize an array of pointers to `char` using strings, as illustrated in the example given below.

```
char *fruits[] = {"Apple", "Orange", "Pineapple", "Pear"};
```

This statement causes memory to be allocated as shown in Fig. 12.4b. The size of the pointer array is automatically calculated by the compiler. It is also possible to make a provision for more numbers of strings by specifying the array size as shown below.

```
char *fruits[10] = {"apple", "Orange", "Pineapple", "Pear"};
```

In this case, the first four pointers are initialized to point to specific strings and the remaining pointers are initialized to 0, i. e., `NULL` pointer. Note that no memory is allocated for the remaining strings.

### Program 12.5 Separate words from a given string to an array of pointers

In Program 12.3, we separated the words in a given string into a two-dimensional array of type `char`. That code has two problems. First, it fails if the number of words in the given string exceeds the limit `MAX_WORD` or the number of characters in any word exceed the limit `MAX_CHAR`. Another problem is that it uses more memory to separate words. Let us now rewrite the code to overcome some of these drawbacks.

The drawbacks of extra memory usage and word length limit are easily overcome by using dynamic memory allocation for each word, as shown below. When a new word is available, the function `str_words` given below uses the `dstr_cpy` function explained above to store a new word in dynamically allocated memory and store its pointer in the `words` array.

```
/* separate words in a given string in an array of pointers */
```

```

int str_words(const char *a, char *words[MAX_WORD])
{
    char buf[100];
    int in_word;
    int w, c;
    int i;

    w = -1;
    in_word = 0; /* we are not in a word */

    for(i = 0; a[i] != '\0'; i++) {
        if (in_word == 0) { /* if not in a word */
            if (!isspace(a[i])) { /* if cur char is not a white space */
                in_word = 1; /* we are in a word now */
                w++;
                c = 0;
                buf[c] = a[i]; /* write first char in a word to buf*/
            }
        }
        else if (isspace(a[i])) { /* in a word, whitespace encountered
            in_word = 0; /* we are outside word now */
            buf[++c] = '\0'; /* write null terminator to word */
            words[w] = dstr_cpy(buf);
        }
        else buf[++c] = a[i]; /* write next char in a word */
    }
    buf[++c] = '\0'; /* write null terminator to last word */
    words[w] = dstr_cpy(buf);

    return w + 1;
}

```

This implementation does not overcome the limitation of maximum number of words. We can achieve it by reallocating memory to `words` array when its capacity is exceeded. This is left as an exercise.

### 12.3.7 Command-Line Arguments

When a C program is complied, we obtain an executable file. This file can be executed from the command line. For example, when we compile the `hello.c` file in the MS Windows environment, we obtain an executable file named `hello.exe` which can be executed from the command prompt as

C>hello

We may wish to send additional information to a program from the command line. For example, to copy a file, we specify the source and destination file names in the **cp** command in Linux OS as in

```
$cp dest file src file
```

Similarly, to delete a directory and all its contents recursively, we use the **rm** command as

```
$rm -r -f dir name
```

where the **-r** switch indicates recursive operation and **-f** switch indicates forceful (without prompt) deletion.

Such values specified in a command line after the command word are called **command-line arguments**. They can be used to provide data to a program, specify file names on which operations should be performed, specify command-line switches (e. g., **-r** and **-f** in the above example) to alter the behaviour of a program, etc. Thus, command-line arguments are used to generalize the behaviour of a program.

We can make a provision to accept command-line arguments in a program. For this, we need to specify two parameters in the **main** function as

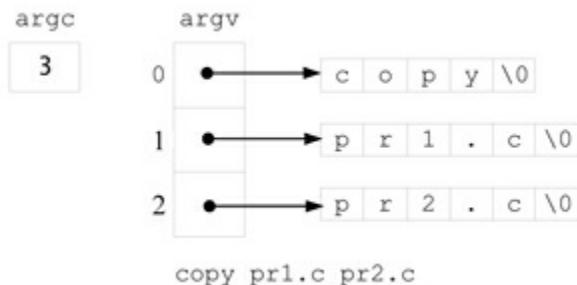
```
int main (int argc, char *argv[])
```

Here, **argc** (argument count) is an integer parameter that specifies the number of command-line arguments and **argv** (argument values) provides the actual arguments. Note that **argv** is an array of pointers to **char** type. It stores pointers to strings representing command-line arguments, as shown in Fig. 12.5. The arguments entered on the command line are available in the **argc** and **argv** parameters in the **main** function during program execution.

Note that the command-line arguments are stored as strings. Thus, if numeric values are passed to a program as command-line arguments, they should be converted to numeric type using functions such as **atoi**, **atod**, etc.

## Program Execution

Depending on the OS and GUI provided, we can use one of three ways to execute a program that accepts command-line arguments: compile in GUI and run from the command line, compile and run from the GUI and compile and run from the command line.



**Fig. 12.5** The `argc` and `argv` values for a command to copy a file

In the first approach, we first compile the program in the GUI window to create an executable file and then run that file from a command window. Thus, in MS Windows, we open a command window by entering the `cmd` command in the *Start/Run...* dialog box, change to the directory containing the program executable file using the `cd` command and then type the command containing the command-line arguments.

For example, to execute a program `cla.exe` (in `E:\Bichkar\Prog` folder) that accepts any number of command-line arguments, we can type the commands as shown below. Note that the command prompt is shown in bold and should not be typed. We can use the *Tab* key to complete the file and folder names on the command line.

```
C>E:
E>cd \Bichkar\Prog
E>cla One Two Three Four
```

In the second approach, we specify the command-line arguments in the GUI itself, using a menu command (*Run/Arguments...* in TC++, *Execute/Parameters* in Dev-C++ and *Project/Set programs' arguments* in Code::Blocks while working with Projects). Note that the name of the executable file should not be entered in the dialog box provided. Once the command-line arguments are specified, we can execute the program as usual.

We can also compile and execute programs from the command line. The command to enter the program depends on the compiler (`tcc` in TC++, `cc` for GNU C compiler). The program executable file is specified using the `-o` flag followed by the file name. While using the GNU C compiler, if the output file is not specified, the default is `a.exe` in MS Windows or `a.out` in Linux. Thus, we can compile the `cla.c` program (located in `E:\Bichkar\Prog` folder) that accepts any number of command-line arguments and execute it as shown below.

```
C>E:
E>cd \Bichkar\Prog
E>cc cla.c -o cla.exe
E>cla One Two Three Four
```

## Example 12.6 Command-line arguments

### a) Display command-line arguments

The program given below displays the command-line arguments passed to it. It first displays the value of `argc` and then uses a `for` loop to display the strings contained in the `argv` array.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    printf("argc: %d\n", argc);
    for(i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);
    return 0;
}
```

This program is saved as `cla.c` and then complied to obtain the executable file `cla.exe`. The output obtained by executing this file from the command line along with some command-line arguments is given below.

```
c>cla One Two Three
argc: 4
argv[0]: cla
argv[1]: One
argv[2]: Two
argv[3]: Three
```

### b) Simple calculator using command-line arguments

A program for a simple calculator using command-line arguments is given below. It evaluates an expression of the form  $a \text{ op } b$ , where  $a$  and  $b$  are floating-point numbers and  $\text{op}$  is an operator (+ - \* /).

```
/* scal.c: simple calculator using command-line arguments */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
```

```

double a, b, ans;
char op;

if (argc != 4) {
    printf("Usage: scal val1 op val2\n");
    exit(1);
}

a = atof(argv[1]);
op = argv[2][0];
b = atof(argv[3]);

switch(op) {
    case '+': ans = a + b; break;
    case '-': ans = a - b; break;
    case '*': ans = a * b; break;
    case '/': ans = a / b; break;
    default: printf("Unknown operator\n"); break;
}
printf("%.3f %c %.3f = %.3f\n", a, op, b, ans);

return 0;
}

```

The program output to perform the addition of two numbers is given below<sup>2</sup>.

```
c>scal 1.1 + 2.2
1.100 + 2.200 = 3.300
```

## Exercises

1. Answer the following questions in brief:
  - a. While defining a function to process a string, how should we declare a string parameter that is an input-only parameter, i. e., it should not be modified in the function definition?
  - b. What precautions should we take while writing a function to copy or concatenate a string?
  - c. Which standard library functions can be used to search a character in a given string?
  - d. Name the standard library functions for searching within a string.
  - e. What is the difference between the `strcpy` and `strncpy` functions?

2. Identify the errors if any in the following function definitions and rewrite them correctly:

```
a. /*copy string s2 to s1 */
void str_copy(char s1[], char s2[], int n)
{
    int i;
    for(i = 0; i < n; i++)
        s1[i] = s2[i];
}

b. /* copy a string s2 to s1 */
void str_copy(char s1[], char s2[])
{
    int len = strlen(s2); /* det length of string s2 */
    for(i = 0; i < len; i++)
        s1[i] = s2[i];
}

c. /* return number of digits in a given string */
int str_digits(char *str)
{
    int count = 0, i = 0;
    while(str[i] != '\0') {
        if (str[i] >= '0' || str[i] <= '9')
            count++;
        i++;
    }
    return count;
}
```

3. Write C programs for the following problems.

- Accept a string and count the number of vowels and consonants in it.
  - Display a given string by replacing each letter with the next one, except letters z and Z, which should be replaced by letters a and A, respectively. Thus, the string "for loop is very powerful" should be displayed as gps mppq jt wfsz qpxfsgvm.
4. Define functions for the following operations on strings:
- Determine the length of the initial portion of a given string containing only letters.
  - Count the number of words in a given string.
  - Change the case of characters in a given string.
  - Test whether a given string is a palindrome or not, i. e., whether it reads the same in the forward and backward direction, as in "never odd or even" and "Don't nod".

- e. Convert a given two-digit number to word.

## Exercises (Advanced Concepts)

1. Answer the following questions in brief:
  - a. What is the difference between the `strcpy` and `stpcpy` functions?
  - b. Why does a string processing function such as `strcpy` return a pointer to the result string?
  - c. State the advantages of an array of pointers to `char` over a two-dimensional array of `char` to store an array of strings.
  - d. Name the parameters passed in the `main` function to accept command-line arguments. Also state the types of these parameters.
2. Determine the purpose of the following program segment:

```
char *digit_str[] = {"Zero", "One", "Two", "Three", "Four", "Five",
                     "Six", "Seven", "Eight", "Nine", "Ten"};  
  
int num, d, digits[5];  
  
scanf("%d", &num);
for (d = 0; num > 0; num /= 10)
    digits[d++] = num % 10;  
  
for (d--; d >= 0; d--)
    printf("%s ", digit_str[digits[d]]);
```

3. Write C programs for the following problems.
  - a. Accept a sentence and print the words in reverse order, e. g., the string "C is a very easy programming language" should be printed as `language programming easy very a is C`
  - b. Replace each consonant in a string with the next one except letters z and Z, which should be replaced by letters b and B, respectively. Thus, the string "Programming in C is fun" should be modified as `Qsohsanniph ip D it gup`
4. Define functions for following operations on strings:
  - a. Replace all occurrences within a substring with another string.
  - b. Convert a given integer number (represented using `long int`) to word (use Indian conventions: thousand, lakh and crore).
  - c. Convert a given string representation of a number to an integer number that can be represented in

a **long int** (use international conventions: thousand, million and billion).

---

**<sup>1</sup> size\_t** is an integer type used for memory object sizes and repeat counts. It is defined in several header files including **stdio.h**, **string.h** and **stdlib.h**. This type is used to indicate the return type in **strlen** function and to specify the number of characters to be operated in a string in several other functions such as **strncpy**, **strncat**, **strcmp**, etc.

**<sup>2</sup>** While using GNU C compiler (in Dev-C++ or Code::Blocks), the program gives wrong answer for multiplication operation

# 13 Structures

This chapter covers *structures* which are collections of data of the same or different type. When combined with arrays and pointers, they enable us to represent very complex data.

We begin with the basics of structures, which includes the declaration of a structure, definition of structure variables, accessing their elements and performing operations on them. Then we study how structure variables can be passed to and returned from functions. Structures containing arrays are covered next followed by nested structures, i. e., structures containing other structures.

The *Advanced Concepts* section presents several concepts that include pointer to a structure, array of structures, combinations of structures and arrays to represent complex data structures and structures containing pointer members.

## 13.1 Structures

We have seen that an array is a powerful feature in C language that allows us to group data items of the same type. However, we often require related data items of different types to be grouped together. Consider the examples given below:

1. Process the information of a library book (accession number, title, author, publisher, price, purchase date, issue status, etc.)
2. Process the information of a student (roll number, name, date of birth, e-mail address, marks in several subjects, result, percentage marks, etc.)
3. Process the information of an inventory item (item code, name, price, quantity, etc.)

We cannot use arrays in such situations as the individual data items are of different types. The C language provides a very powerful feature, called **structure**, that allows us to group related data items of the same or different types.

### 13.1.1 Declaring Structures

To use a structure in our program, we have to first *declare* it. The general format for the declaration of a structure is given below.

```
struct tag {
```

```
member-declarations ;  
};
```

where **struct** is a C keyword that introduces the structure declaration and *tag* is the name of the structure being declared. The rules for naming a structure tag are the same as those for a variable name. We can define multiple structures within a scope. However, each of them must have a distinct name. Note the semicolon after the closing brace '}'. This is one of the very few places such where a semicolon is required.

The variables declared within the curly braces in the above declaration are called the **members**, **fields** or **components** of a structure. As mentioned earlier, the members of a structure may be of the same or different types. Thus, the *member-declarations* may consist of one or more declaration statements each of which takes the usual form:

```
type member_list ;
```

where *type* is a data type and *member\_list* is a comma-separated list of structure members (i. e., variables). The members of a structure must have distinct names. However, a member variable may have the same name as that of the structure tag. The member variables in different structures may also have same names. Moreover, the name of a structure tag or member variable may be same as that of a non-member variable. As good programming practice, we usually give distinct names for structure tags, structure members and non-member variables. However, we may occasionally reuse the member names.

In this section, we consider the structure members of built-in data types such as **char**, **int**, **float**, etc. The C language also allows arrays, structures and pointers to be used as members of a structure. These topics are discussed later in this chapter.

### Example 13.1 Declaring structures

#### a) Structure to represent a date

Consider the declaration of structure **date** given below.

```
struct date {  
    int dd, mm, yy;  
};
```

This structure has three members, namely, **dd**, **mm** and **yy**, all of type **int**. They represent the day of month, month and year, respectively. We often declare each member variable on a separate line. This allows us to write comments for each member variable, as shown below.

```
struct date {  
    int dd; /* day of the month */  
    int mm; /* month */
```

```
    int yy; /* year */  
};
```

Observe that we have used a structure to represent a date although all the members are of the same data type. The use of an array to represent a date will lead to programs that are difficult to understand because the components of a date will have to be identified from array subscripts, e. g., `d[0]` as day of month, `d[1]` as month and `d[2]` as year. Moreover, differences in representation of dates in different parts of world (`dd/mm/yy` or `mm/dd/yy`) will cause further difficulties in understanding and modifying the program. On the other hand, the use of a structure allows us to refer to the components of a date by appropriate member names, as in `d.dd`, `d.mm` and `d.yy`, making the programs easy to understand.

#### b) Structure to represent a complex number

A complex number  $x + iy$  has two parts, a real part  $x$  and an imaginary part  $y$ . Though we can use two independent variables, say `re` and `im`, to represent the real part and imaginary part, respectively, it is much more convenient to declare a structure `complex` having two members (`re` and `im`), as shown below.

```
struct complex {  
    float re, im;  
};
```

#### c) Structure to represent a circle

Consider the declaration of structure `circle` given below which has four members: `x`, `y` and `radius` of type `int` and `filled` of type `char`.

```
struct circle {  
    int x, y; /* coordinates of center */  
    int radius;  
    char filled; /* fill status Y/N */  
};
```

#### d) Structure and member name usage

Consider the declarations given below to illustrate the usage of names:

```
struct point {  
    char point; /* struct name point used as a member */  
    int x, y;  
};  
struct circle {
```

```

int x, y, radius; /* members of point struct (x & y) reused */
char filled;
};

float x, y;           /* struct members x & y used as variables */
char point;           /* struct name point used as variable */

```

First observe that the structure name `point` is also used as a member variable in the same structure. In addition, the members `x` and `y` are used in both the structures. Further note that the structure name `point` and members `x` and `y` are also defined as scalar variables.

### 13.1.2 Defining Structure Variables

The declaration of a structure only lists its members along with their types. It does not allocate any memory for member variables. The structure name is thus similar to a data type. Memory is allocated only when we define variables of this structure type.

Once a structure is declared, we can define variables of that type using the following syntax:

```
struct tag var_list;
```

where `tag` is the structure name and `var_list` is a comma-separated list of variable names. A structure variable name must be distinct within a scope. Thus, it cannot be the same as that of any other variable in that scope. However, it can be the same as that of a structure tag. For example, we can declare variables `d` and `today` of type `struct date` just declared as follows:

```
struct date d, today;
```

Now both the variables `d` and `today` have three members, namely, `dd`, `mm` and `yy` of type `int`.

We can also define variables at the end of a structure declaration, between the closing brace and the semicolon, using the format given below.

```
struct tag {
    member declarations;
} var_list;
```

Thus, we can declare a structure to represent complex numbers and also define variables `x` and `y` as complex numbers using the declaration given below.

```
struct complex {
    float re, im;
} x, y;
```

In this case, variables `x` and `y` are of type `struct complex` and each of them has two members `re` and `im` of type `float`.

Note that it is also possible to omit the structure name in a structure declaration. Such structures are called **anonymous structures**. This is particularly useful when we want to declare all the required variables of this structure type in this anonymous structure declaration and do not have to use the structure name in the subsequent program (e. g. to pass or return values to and from a function). Thus, the above declaration can also be written as shown below.

```
struct {  
    float re, im;  
} x, y;
```

Finally, note that we can declare scalar variables as well as arrays of structure types. In this section, we consider the scalar structure variables only. Arrays of structures are discussed in the Advanced Concepts Section.

### 13.1.3 Accessing Structure Members – the Dot Operator

Each structure variable has member variables as specified in the declaration of the corresponding structure. To access the individual members of a structure variable, C language provides the **structure member operator** ( . ), which is also called as the **dot operator**. A member of a structure variable can be accessed by writing the structure variable name followed by the dot operator and the name of desired member as shown below.

```
struct_var.member_name
```

Thus, the members of variable `today` of type `struct date`, declared in the previous section, can be accessed as `today.dd`, `today.mm` and `today.yy`. Note that we can use a structure member only in conjunction with a structure variable. Using a structure member on its own is an error that beginners should avoid. For example, using `dd`, `mm` and `yy` without the name of the structure variable is an error.

Once we access a structure member as shown above, we can perform various operations on it, e. g., we can use it in an expression, assign a value to it, pass it as an argument to a function and so on. Note that the dot operator has higher precedence than most other operators. Thus, when we access a structure member within an expression, we do not need to enclose it within parentheses.

To increment (decrement) a value of a structure member, we can use either the prefix or postfix increment (decrement) operator as follows:

```
++ struct_var.member_name  struct_var.member_name ++  
-- struct_var.member_name  struct_var.member_name --
```

Note that in these expressions, the dot operator is bound first to its operands followed by the `++` (`--`) operator. Thus, the increment (decrement) operator operates on the member variable and not the structure variable.

We often require the address of a member variable, e. g., to read its value using the `scanf` function. We can obtain it using the *address-of* operator (`&`) as shown below.

```
&struct_var.member_name
```

In this case, the dot operator is correctly bound to its operands first followed by the address-of operator. Thus, the address-of operator gives us the address of the member variable and not the structure variable.

### Example 13.2 Performing operations on member variables

A pixel on the computer screen can be represented by  $x$  and  $y$  coordinates (integer values). The code given below declares structure `pixel` and defines several variables of this type.

```
struct pixel {
    int x, y;
} left_top, right_bottom, center;

struct pixel left_bottom, p;
```

Note that three structure variables, namely, `left_top`, `right_bottom` and `center` are defined in the structure declaration itself and the variables `left_bottom` and `p` are declared subsequently.

The top-left corner of the screen has coordinates  $(0, 0)$ . The  $x$  coordinates increase from left to right and  $y$  coordinates increase from top to bottom. Assuming screen resolution of  $640 \times 480$  pixels (i. e., VGA), we can assign values to `left_top` and `right_bottom` as shown below.

```
left_top.x = left_top.y = 0;
right_bottom.x = 639;
right_bottom.y = 479;
```

We can initialize `left_bottom` from the `left_top` and `right_bottom` pixels as follows:

```
left_bottom.x = left_top.x;
left_bottom.y = right_bottom.y;
```

The coordinates of the `center` pixel can be determined from the `right_bottom` pixel as

```
center.x = (right_bottom.x + 1) / 2;
center.y = (right_bottom.y + 1) / 2;
```

We can also check whether a given pixel `p` is on the screen or not as follows:

```
int on_screen;
```

```

if (p.x >= 0 && p.x <= right_bottom.x && /* check x-coordinate */
    p.y >= 0 && p.y <= right_bottom.y)      /* check y-coordinate */
    on_screen = 1;
else on_screen = 0;

```

We can pass the member variables as actual arguments to a function. Thus, we can print the *x* and *y* coordinates of pixel center using the `printf` function as

```
printf("Coordinates of center: x=%d y=%d\n", center.x, center.y);
```

or we can use the `scanf` function to read the coordinates of pixel *p* from the keyboard as

```
scanf("%d %d", &p.x, &p.y);
```

Further, if we have a function `put_pixel` that accepts the *x* and *y* coordinates of a pixel and displays that pixel on the screen, we can call it to display the `center` pixel as follows:

```
put_pixel(center.x, center.y);
```

Finally, note that as the names of structure variables can be same as that of a structure member or a structure tag, the following declarations are valid.

```

struct pixel x, y;
struct pixel pixel;

```

The members of structure variable *x* are referred as *x.x* and *x.y* and the members of structure variable *y* are referred as *y.x* and *y.y*.

#### 13.1.4 Structure Initialization

The C language allows initialization of a structure variable, the syntax for which is similar to the initialization of an array as shown below:

```
struct tag var = { expr1, expr2, ... };
```

where *tag* is the name of the structure already defined and *var* is a variable of type `struct tag`. The initializer is a comma-separated list of expressions of appropriate type enclosed in curly braces. The first member of structure variable *var* is initialized with the value of *expr1*, the second member is initialized with *expr2* and so on. The type of each expression must be in accordance with the corresponding member variable; otherwise, conversions are applied to convert its value to the desired type. However, if such a conversion is not possible, the compiler will report an error.

It is not necessary to initialize all members of a structure variable. If we specify fewer values than required, the remaining members are initialized with the default value, which is zero for arithmetic variables and `NULL` for pointer variables. However, if we specify more values than required, the

compiler will issue an error message.

Although we can initialize multiple variables in a single initialization statement as shown below, it is better to initialize only one variable at a time.

```
struct tag var1 = { expr, expr, ...},  
    var2 = { expr, expr, .}, ... ;
```

We can combine structure declaration and initialization of structure variables as shown below.

```
struct tag {  
    member-declarations;  
} var1 = { expr, expr, ... },  
var2 = { expr, expr, ...}, ... ;
```

Further, we can mix the definition and initialization of structure variables. Thus, in the statement given below, three variables are defined but only one is initialized. However, it is better to avoid such code as it affects program readability.

```
struct tag var1, var2 = {expr1, expr2, ... }, var3 ;
```

We can also initialize a structure variable using variables of same type as shown below.

```
struct tag var1 = var2 ;
```

Finally, note that we cannot initialize the members of a structure in its definition. Thus, the following format is incorrect.

```
/* incorrect format for initialization of a structure */  
struct tag {  
    type1 var1 = expr1; /* wrong */  
    type2 var2 = expr2; /* wrong */  
    ...  
};
```

### Example 13.3 Initialization of structure variables

Consider the declaration of structure **date** and initialization of variable **sem1\_start**:

```
struct date {  
    int dd, mm, yy;  
} sem1_start = {7, 6, 2005};
```

The members **dd**, **mm** and **yy** of structure variable **sem1\_start** are initialized with the values 7, 6 and 2005, respectively. We can now initialize more variables of type **struct date** as follows:

```
struct date sem1_end = {5, 12, 2005}, sem1_exam = {20, 11, 2005};
```

Now consider the following initializations:

```
struct date new_year = {1, 1};  
struct date childrens_day = {14, 11};
```

These statements initialize the variable `new_year` to January 1 and `childrens_day` to November 14. As the value for member `yy` has not been specified, it is initialized to zero in both variables.

Finally, note that the code given below is incorrect.

```
struct date {  
    int dd = 1;      /* wrong code - structure members can not be */  
    int mm = 1;      /* initialized in the declaration */  
    int yy = 2005;   /* of a structure */  
};
```

### 13.1.5 Structure Assignment

To copy (or assign) an entire structure variable to another of the same type, we need to copy the values of member variables one at a time. However, there is a simpler way to do this. The C language allows a structure variable to be assigned to another variable of the same type using an assignment of the form

```
var1 = var2;
```

This statement assigns the value of each member of structure variable `var2` to the corresponding member of variable `var1`. Thus, the code given below first initializes a complex number `a` and then assigns it to another complex variable `b`:

```
struct complex {  
    float re, im;  
} a = {3, 4};  
  
struct complex b;  
b = a;
```

### 13.1.6 Other Operations on Structures

We have seen that the C language provides a rich set of operators to perform operations on variables of built-in data types. It also provides standard library functions to perform input and output operations on such variables.

However, the C language provides very few operations on structure variables; these include member access, initialization and assignment. For other operations such as addition, subtraction, increment,

decrement, comparison, input, output, etc., we will have to write our own code. Usually, we define functions to perform such operations on specific types of structures as discussed in the next section.

### Program 13.1 Addition of complex numbers

Write a program to add of two complex numbers.

**Solution:** Let us use structure **complex** to represent a complex number and write a simple program containing only the **main** function. Thus, we can define structure **complex** either in the **main** function or before it as a global definition.

The addition of two complex numbers involves the addition of the real and imaginary parts of the two numbers. Let us use structure variables **x** and **y** to represent the complex numbers being added and the structure variable **z** to represent the result.

In the **main** function, we declare structure **complex** and variables **x**, **y** and **z**. Then we read two complex numbers from the keyboard, add them and print the result. The program is given below.

```
/* Addition of complex numbers: uses local structure definition */
#include <stdio.h>

int main()
{
    struct complex {
        float re, im;
    } x, y, z;

    /* read complex numbers */
    printf("Enter real and imaginary parts of\n");
    printf(" complex number x: ");
    scanf("%f %f", &x.re, &x.im);
    printf(" complex number y: ");
    scanf("%f %f", &y.re, &y.im);

    /* add complex numbers */
    z.re = x.re + y.re;
    z.im = x.im + y.im;

    /* print addition */
    printf("z.re = %4.2f z.im = %4.2f\n", z.re, z.im);
    return 0;
}
```

As mentioned earlier, we can also declare structure **complex** before the **main** function. This is

particularly vital if we want to use this structure in other functions besides `main`. However, we may prefer to declare the variables `x`, `y` and `z` as local variables within the `main` function as shown below.

```
/* Addition of complex numbers: uses global structure definition */
#include <stdio.h>

struct complex {
    float re, im;
};

int main()
{
    struct complex x, y, z;
    /* rest of the code same as above */
}
```

### Program 13.2 Determine validity of a given date (using structure and functions)

Write a program to determine whether a given date is valid or not.

**Solution:** A program to check the validity of a given date using functions is presented in Program 9.6. In this example, we modify that program using structure `date` to represent a date. However, we use functions `date_valid`, `month_days` and `is_leap` without any modifications. As these functions accept the individual components of a date as parameters, they do not require access to the definition of structure `date`. Thus, in the program given below, we declare structure `date` within the `main` function. The code for the functions `date_valid`, `month_days` and `is_leap` is omitted to save space.

```
/* Date validity using functions and structure */
#include <stdio.h>

int date_valid(int d, int m, int y);
int month_days(int m, int y);
int is_leap(int y);

int main()
{
    struct date {
        int dd, mm, yy;
    } d;

    printf("Enter date (dd mm yyyy): ");
    scanf("%d %d %d", &d.dd, &d.mm, &d.yy);
```

```

if (date_valid(d.dd, d.mm, d.yy))
    printf("Date is valid\n");
else printf("Date is invalid\n");
}

/* include code for date_valid, month_days and is_leap functions
from Program 9.6 */

```

Note that variable `d`, which is used to represent a date, is declared in the definition of structure `date`. Also, observe how the members of variable `d` are accessed in the `scanf` and `date_valid` function calls.

## 13.2 Structures and Functions

In the previous section, we learnt how the individual members of a structure variable can be passed as arguments to a function. However, it is tedious to pass an entire structure to a function by passing its individual members, particularly when the structure contains several members. The C language allows an entire structure variable to be passed as an argument to a function. It also allows a structure variable to be returned as a function value.

Note that a structure is passed by value. When a function is called, the entire structure argument is copied to the corresponding structure parameter. Thus, modifications to the members of the structure parameter (inside the called function) will not be reflected in the corresponding argument in the called function. We can sometimes take advantage of the fact that the structure parameter is local to the function and can be modified freely, if required.

As mentioned before, we can also return a structure as a function value. This returned structure is usually assigned to a structure variable in the called function or passed to another function.

The structure copy operation during a function call and return can be time-consuming and may result in inefficient programs, particularly when the structure is large. We can avoid the copying of structures and improve program efficiency by using pointers to structures, as explained in Section 13.5.1.

### 13.2.1 Passing a Structure to a Function

The C language allows an entire structure variable to be passed as an argument to a function. The general format of a function that takes a single structure parameter is shown below.

```

ret_type func_name ( struct tag param )
{
    ...
}

```

where *tag* is the structure name and *param* is the function parameter name. If the members of structure *tag* are named *member1*, *member2*, ..., the members of parameter *param* can be accessed within the function as *param.member1*, *param.member2*, ... . We can use these members in expressions, pass them to other functions as arguments or even assign new values to them.

The function *func\_name* given above can be called from some function as shown below.

```
struct tag arg;  
...  
func_name( arg );
```

Here, variable *arg* is defined as a variable of type **struct tag** and then function *func\_name* is called with *arg* as the argument for the formal parameter *param*. Observe that the parameter *param* and argument *arg* are of the same type.

A function can have more than one structure parameter. The general format for the definition of such a function is given below where the formal parameters *param1*, *param2*, ... may be of the same or different structure types.

```
ret_type func_name( struct tag1 param1, struct tag2 param2, ... )  
{  
    ...  
}
```

Besides one or more structure parameters, a function can have any number of other parameters which may be scalar variables or arrays.

#### Example 13.4 Passing structures to a function

##### a) Magnitude of a complex number

A function to calculate the magnitude of a given complex number is given below.

```
/* magnitude of a complex number */  
float complex_mag(struct complex a)  
{  
    return (float) sqrt(a.re * a.re + a.im * a.im);  
}
```

This function declares parameter **a** of type **struct complex** (declared in Example 13.1b). It calculates and returns the magnitude of the given complex number using its members, i. e., **a.re** and **a.im**. As the **sqrt** function returns a value of type **double**, we have used an explicit typecast to avoid a compiler warning. This function can be called from the **main** function as follows:

```

void main()
{
    struct complex a = {1, 2};
    float mag;

    mag = complex_mag(a);
    ...
}

```

### b) Day of the year for a given date

We can determine the day-of-the-year for a given date dd/mm/yy by first adding the days in months 1 to mm - 1 and then adding value of dd to it. The function `day_of_year` given below uses a parameter of type `struct date` (declared in Example 13.1a).

```

/* determine day-of-year for a given date */
int day_of_year(struct date d)
{
    static int mdays[] = {
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    int m, tot_days;

    /* calculate total days upto mm-1 months */
    tot_days = 0;
    for (m = 1; m < d.mm; m++)
        tot_days += mdays[m];

    /* adjust tot_day value for a leap year */
    if (d.mm > 2 && is_leap(d.yy))
        tot_days++;

    return tot_days + d.dd;
}

```

This function declares a parameter `d` of type `struct date` and returns the day of the year as an integer value. The function uses a static array `mdays` to store the number of days in each month, with value 28 for February. A `for` loop is first used to count the number of days (`tot_days`) in months 1 to mm - 1 ignoring leap year. This value is then adjusted for a leap year if the given month is greater than 2 (i. e., March to December). It is assumed that the function `is_leap` is available. Finally, the value of `d.dd` is added to `tot_days` to determine the day of the year and is returned.

We can eliminate the `for` loop in the above function by using an array `mdays_sum` that contains the total number of days up to each month instead of the `mdays` array used above as shown below.

```
/* determine day-of-year for a given date */
int day_of_year(struct date d)
{
    /* total days up to each month, ignoring leap year */
    static int mdays_sum[] = {
        0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334
    };

    int tot_day = mdays_sum[d.mm-1];
    if (d.mm > 2 && is_leap(d.yy)) /* adjust for leap year */
        tot_day++;
    return tot_day + d.dd;
}
```

Note that assuming 28 days for February, there are 59 days up to February, 90 days up to March and so on. Also, note that we do not require the total for the last month, i. e., December.

We can rewrite the code for calculating the day of the year concisely using the conditional operator as shown below. Note that the parentheses surrounding conditional expression in the `return` statement are essential.

```
/* determine day-of-year for a given date */
int day_of_year(struct date d)
{
    static int mdays_sum[] = {
        0, 31, 59, 90, 120, 151, 182, 212, 243, 273, 304, 334
    };

    return mdays_sum[d.mm-1] + (d.mm > 2 && is_leap(d.yy)? 1 : 0)
        + d.dd;
}
```

## General Structure of a Program Using Structures and Functions

We have seen that a function can have one or more structure parameters. The declarations of these structures must be available to it. Moreover, they should be available to function prototypes which are usually written before the `main` function. Thus, we can declare all the structures at the beginning of a program, usually after the `#defines` but before the function prototypes. Such declarations are called global declarations. The general structure of a program using structures and functions is given below.

```
#includes
```

#defines  
structure declarations  
function prototypes  
**main** function  
other function definitions

### Program 13.3 Compare two dates

Write a program, using structures and functions, to compare two dates.

**Solution:** Consider the problem of comparison of two valid dates  $d_1$  and  $d_2$ . There are three possible outcomes of this comparison:  $d_1 == d_2$  (dates are equal),  $d_1 > d_2$  (date  $d_1$  is greater, i. e., occurs after  $d_2$ ) and  $d_1 < d_2$  (date  $d_1$  is smaller, i. e., occurs before  $d_2$ ). Let us write a function that accepts two dates as structures **d1** and **d2** of type **struct date** and returns 0 if the dates are equal, 1 if **d1** is later than **d2** and -1 if date **d1** is earlier than **d2**.

The given dates are equal if all the components (**dd**, **mm** and **yy**) of two dates are equal. Also, date **d1** is greater if any one of the following conditions is satisfied:

1. **d1.yy** > **d2.yy**,
2. **d1.yy** == **d2.yy** and **d1.mm** > **d2.mm**,
3. **d1.yy** == **d2.yy** and **d1.mm** == **d2.mm** and **d1.dd** > **d2.dd**

A complete program that uses the **date\_cmp** function to compare two given dates using an **if-else-if** statement and prints the appropriate message in the **main** function is given below.

```
/* compare two dates and print appropriate message */
#include <stdio.h>

struct date {
    int dd, mm, yy;
};

int date_cmp(struct date d1, struct date d2);
void date_print(struct date d);

int main()
{
    struct date d1 = {7, 3, 2005};
    struct date d2 = {24, 10, 2005};
```

```

date_print(d1);

int cmp = date_cmp(d1, d2);
if (cmp == 0)
    printf(" is equal to ");
else if (cmp > 0)
    printf(" is greater i.e. later than ");
else printf(" is smaller i.e. earlier than ");
date_print(d2);

return 0;
}

/* compare given dates d1 and d2 */
int date_cmp(struct date d1, struct date d2)
{
    if (d1.dd == d2.dd && d1.mm == d2.mm && d1.yy == d2.yy)
        return 0;
    else if (d1.yy > d2.yy || d1.yy == d2.yy && d1.mm > d2.mm ||
              d1.yy == d2.yy && d1.mm == d2.mm && d1.dd > d2.dd)
        return 1;
    else return -1;
}

/* print a given date */
void date_print(struct date d)
{
    printf("%d/%d/%d", d.dd, d.mm, d.yy);
}

```

The program output is given below.

```
7/3/2005 is smaller i.e. earlier than 24/10/2005
```

### 13.2.2 Structure as a Function Value

The C language allows a single structure variable to be returned as the value of a function. The general format of such a function is given below.

```

struct tag func_name (param_list )
{
    struct tag s;
    ...
}
```

```
    returns;  
}
```

This function must return a structure variable of appropriate type (`struct tag`). Also recall that the function may contain more than one `return` statements.

### Example 13.5 Structure as a functions value

#### a) Function to read a complex number

A function `complex_read` to read a complex number from the keyboard is given below.

```
/* read a complex number from keyboard */  
struct complex complex_read()  
{  
    struct complex a;  
    scanf("%f %f", &a.re, &a.im);  
  
    return a;  
}
```

This function has no parameters and it returns a value of type `struct complex`. A local variable `a` of type `struct complex` is used to store the complex number entered from the keyboard. The values of `a.re` and `a.im` are read from the keyboard and the value of structure variable `a` is returned. This function can be called from some other function, say `main`, as shown below.

```
int main()  
{  
    struct complex x;  
    x = complex_read();  
    ...  
}
```

#### b) Function for addition of complex numbers

Program 13.1 used structure `complex` but performed the addition of two complex numbers in the `main` function itself. Assuming that `complex` is defined as a global structure, we can define a function for adding two complex numbers as shown below.

```
/* addition of two complex numbers */  
struct complex complex_add(struct complex a, struct complex b)  
{
```

```

    struct complex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c;
}

```

This function has two parameters **a** and **b** of type **struct complex** and it returns a value of type **struct complex**. Within the function body, the real and imaginary parts are added in a local variable **c** (also of type **struct complex**) which is then returned.

Note that the parameters **a** and **b** in the above function are value parameters and can be modified without affecting the corresponding arguments in the function call. Thus, we can eliminate the local variable **c** within this function and calculate the addition in parameter **a** itself as shown below.

```

/* addition of two complex numbers */
struct complex complex_add(struct complex a, struct complex b)
{
    a.re += b.re;      /* perform addition in variable a itself */
    a.im += b.im;
    return a;          /* return addition */
}

```

### c) Function to increment a given valid date

In Program 9.7, we learnt how to increment a given valid date. Here, we define a function **date\_incr** that accepts a date (assumed to be valid) and returns the incremented date as the function value. Note that the function parameter **d** of type **struct date** is passed by value and is local to the function. Thus, we have used it to determine next date and avoided the use of a separate local variable.

```

/* increment given valid date and return it */
struct date date_incr(struct date d)
{
    static int mdays[] = {0,31,28,31,30,31,30,31,31,30,31,30,31,30,31};
    if (++d.dd > mdays[d.mm] + (d.mm==2 && is_leap(d.yy) ? 1 : 0)) {
        d.dd = 1;
        if (++d.mm > 12) {
            d.mm = 1;
            ++d.yy;
        }
    }

    return d;
}

```

### 13.3 Structure Containing Arrays

The C language allows an array to be used as a structure member. This enables us to group related structure members and is also useful for declaring character strings as structure members. This section explains how we can effectively use such structures. This includes

1. Declaration of structures containing arrays
2. Initialization of structures containing arrays
3. Accessing member arrays and passing them to functions
4. Accessing individual elements of member arrays
5. Using structure that contain arrays as function parameter and return value

#### 13.3.1 Declaring Structures Containing Arrays

A structure may contain one or more array members, each of a different type and size. The array members are declared using the usual syntax. We can declare ordinary (scalar) members and array members in any order. However, remember that the names of the members of a structure must be distinct.

##### Example 13.6 Declaring structures containing arrays

###### a) Structure declaration for a person's data

Consider the declaration of structure `person` given below that uses a character array `name` as a member to store a person's full name, and two other members to store the person's gender and age.

```
struct person {  
    char name[60]; /* person's name */  
    char gender;   /* M: male, F: female */  
    int age;  
};
```

We often expect a person's name to be displayed in some specific order, e. g., last name followed by first name and middle name. This cannot be ensured if we use a single character array to store the person's name. Hence, we can use three arrays, say, `lname`, `fname` and `mname`, to store these components. We can also store the salutation (such as Mr., Ms., Mrs., Dr., Shri, Smt., etc.) in another array named `salut`. Thus, the structure `person` is modified as shown below.

```
struct person {  
    char salut[5]; /* salutation */  
    char lname[20]; /* last name */
```

```

char fname[20]; /* first name */
char mname[20]; /* middle name */
char gender;     /* M: male, F: female */
int age;
};

```

Although one character position is required to store the null terminator for the string data, the sizes of the member arrays (`salut`, `lname`, `fname` and `mname`) are adequate for most, if not all, persons. We can increase these array sizes, but it leads to large memory wastage, particularly when we declare an array of this structure to store the data of several persons. As multiple members can be declared in a single declaration, we can rewrite the above declaration as shown below.

```

struct person {
    char salut[5];                      /* salutation */
    char lname[20], fname[20], mname[20]; /* person's name */
    char gender;                         /* M: male, F: female */
    int age;
};

```

Now we can declare variables of this structure as shown below:

```
struct person boss, secretary, client;
```

#### b) Structure declaration for the HSC examination details of a student

Consider the declaration of structure `hsc_stud` given below that can be used to store the HSC examination details of a student.

```

struct hsc_stud {
    int seat_no;                      /* exam seat number */
    char salut[5];                    /* salutation */
    char lname[20], fname[20], mname[20]; /* student's name */
    int marks[6];                     /* marks in six subjects */
    int tot_marks;                   /* total marks */
    float perce_marks;               /* percentage marks */
    char result[5], class[25];        /* exam result and class obtained */
};

```

The arrays used in this structure are as follows: character array `salute` for salutation, character arrays `lname`, `fname`, and `mname` to store student's name, integer array `marks` to store the marks in six subjects and character arrays `result` and `class` to store the examination result and class obtained, respectively. Other members are used to store the exam seat number, total marks and percentage marks. Note that the order of declaration of these members is quite logical.

### c) Structures representing size-aware vector and matrix

As we know, built-in arrays in C language do not store information about the number of elements in it. Hence, a programmer has to be very careful while processing arrays, particularly during operations like insert and delete. This example overcomes this problem by declaring structures to represent a **vector** and a **matrix** that store the size information along with the array data.

First consider the declaration of structure **vector** that contains two data members: **len** to store array length (i. e., number of elements in it) and an integer array named **data** having 20 elements to store the elements of the vector.

```
struct vector {  
    int len;  
    int data[20];  
};
```

Next, consider the declaration of structure **matrix** given below that contains three members: **rows**, **cols** and **data**. The **rows** and **cols** members represent the number of rows and columns in the matrix and a two-dimensional array named **data** is used to store the matrix elements. Note that the maximum matrix size is specified using symbolic constants **ROWS** and **COLS**.

```
#define ROWS 10  
#define COLS 10  
  
struct matrix {  
    int rows, cols;  
    int data[ROWS][COLS];  
};
```

#### 13.3.2 Initializing Structures Containing Arrays

To initialize a structure containing arrays, we combine the syntax for the initialization of a structure and an array. Thus, in the format for initialization of a structure given below

```
struct tag var = { expr1, expr2, ... };
```

we replace the initialization expression corresponding to the array member with an array initializer list enclosed in curly braces. Also remember that an array of characters can be initialized using a string constant.

Consider the declaration of structure **special\_nos** used to store a list of special numbers and the initialization of variable **prime\_nos** given below.

```
struct special_nos {
```

```

int n;
char name[20];
int data[20];
};

struct special_nos prime_nos = {
    10, "Prime numbers", {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
};

```

Observe that `name` is initialized using a character string, whereas `data` is initialized using array initializer that contains the first ten prime numbers.

Note that other rules for initialization of a structure discussed in Section 13.1.4 are applicable for the initialization of a structure containing arrays as well. In addition, we need not provide values for all elements of contained arrays. If we omit values for some array elements (at the end of the array), the compiler will provide default values for them.

### Example 13.7 Initializing structures containing arrays

#### a) Initialize the information of an employee

```

struct employee {
    int emp_id;          /* employee id */
    char name[60];       /* person's name */
    char dept[20];       /* work department */
    char desg[20];       /* designation */
    int leaves[13];      /* number of leaves in each month */
} e1 = {127, "K. M. Seth", "Mech Engg", "Lecturer", {0, 1, 0, 3}};

struct employee me = {
    102, "R. S. Bichkar", "E&TC Engg", "Professor", {0, 1, 2, 0}
};

```

This example declares a structure `employee` to store information about an employee of an organization and initializes two variables named `e1` and `me`. Observe that the three character arrays, `name`, `dept` and `desg`, are initialized using character strings. Also observe that since `leaves[0]` will be unused, the `leaves` array has 13 elements and the leave data is initialized for the first three months only.

#### b) Initialization of a size-aware unit matrix

This example initializes a unit matrix of size  $3 \times 3$  using the `matrix` structure declared in Example 13.6c. Observe that the matrix size is stored in the structure variable followed by the array data. The

uninitialized elements of array data will have zero value.

```
struct matrix unit3 = {  
    3, 3,          /* rows, cols */  
    {{1, 0, 0},    /* data */  
     {0, 1, 0},  
     {0, 0, 1}}  
};
```

### 13.3.3 Accessing Member Arrays

We can access the entire array member of a structure variable using the usual construct to access a structure member as shown below:

```
struct_var.arr_member
```

We can pass this array to a function. As an array is passed by address, we can use this array as an input-output parameter, i. e., to pass information to a function and to return the processed results back to the calling function.

#### Example 13.8 Passing array members of a structure to a function

##### a) HSC result processing

Consider the following definition of a function that accepts an array of marks in six subjects and returns an integer value indicating the result: 1 if the result is *pass* and 0 otherwise.

```
/* determine HSC result */  
int hsc_result(int marks[])  
{  
    int i;  
    for(i = 0; i < 6; i++) {  
        if (marks[i] < 35)  
            return 0;      /* fail */  
    }  
    return 1;            /* pass */  
}
```

Let us declare a structure variable **stud1** of type structure **hsc\_stud** declared in Example 13.6b as shown below.

```
struct hsc_stud stud1;
```

Now we can call the `hsc_result` function and set the result in the `stud1.result` array member as shown below.

```
if (hsc_result(stud1.marks))
    strcpy(stud1.result, "Pass");
else strcpy(stud1.result, "Fail");
```

Note that we can simplify this code using the conditional operator (?:) as follows:

```
strcpy(stud1.result, hsc_result(stud1.marks)? "Pass": "Fail");
```

Alternatively, we can define function `hsc_result` with one more parameter, a character array `result`, in which the result string can be returned to the calling function as shown below.

```
/* determine HSC result */
void hsc_result(int marks[], char result[])
{
    int i;
    for(i = 0; i < 6; i++) {
        if (marks[i] < 35) {
            strcpy(result, "Fail");
            return;
        }
    }
    strcpy(result, "Pass");
}
```

This function can be called as

```
hsc_result(stud1.marks, stud1.result);
```

#### b) Using the `mat_print` function to print a size-aware matrix

Consider the initialization of a size-aware unit matrix named `unit3` from Example 13.7b declared using structure `matrix` (Example 13.6c). Also consider the prototype of function `mat_print` defined to print an integer matrix of specified size:

```
void mat_print(int a[][10], int m, int n);
```

Note that the dimensions of the arrays (member `data` in `struct matrix` and parameter `a` in function `mat_print`) have the same number of columns (10). Thus, we can use the function `mat_print` to print matrix `unit3` as shown below:

```
mat_print(unit3.data, unit3.rows, unit3.cols);
```

### 13.3.4 Accessing Elements of Member Arrays

An individual element of the member array of a structure variable can be accessed by writing array indexes after the construct used to access the member array. Thus, the elements of a one-dimensional array member can be accessed as shown below:

```
struct_var.arr_member [ expr ]
```

where *expr* is an expression that specifies the index of the array element being accessed. Similarly, the elements of a two-dimensional array member can be accessed using the expression given below.

```
struct_var.arr_member [ expr1 ][ expr2 ]
```

We can perform all operations on such an array element that we can perform on a scalar variable. Thus, we can use it in an expression, assign a value to it, pass it to a function and so on. When we use it in an expression, no additional parentheses are required as the dot operator and the array index operator have higher precedence than most other operators.

For example, the prefix increment, postfix decrement, address-of operator and negation operators can be used with the elements of a one-dimensional array member of a structure as follows:

```
++ struct_var.arr_member [ expr ]  struct_var.arr_member [ expr ]--  
& struct_var.arr_member [ expr ]  -struct_var.arr_member [ expr ]
```

#### Example 13.9 Accessing the elements of member array of a structure

##### a) Determine the total marks in the HSC examination

Assume that a structure variable named **stud** is defined using structure **hsc\_stud** (see Example 13.6b) to represent the HSC examination data of a student as shown below.

```
struct hsc_stud stud;
```

The program segment given below illustrates how the elements of **marks** member array can be accessed to determine the total marks in member **stud.tot\_marks**.

```
int i;  
stud.tot_marks = 0;  
for (i = 0; i < 6; i++)  
    stud.tot_marks += stud.marks[i];
```

##### b) Read and print a size-aware matrix

Consider the program segment given below used to read and print a matrix `m` declared using structure `matrix` (Example 13.6c). Observe how the elements of array member `data` are accessed.

```
int main()
{
    struct matrix m;
    int i, j;

    printf("Enter matrix size: ");
    scanf("%d %d", &m.rows, &m.cols);

    printf("Enter matrix elements:\n");
    for(i = 0; i < m.rows; i++) {
        for(j = 0; j < m.cols; j++)
            scanf("%d", &m.data[i][j]);
        printf("\n");
    }

    for(i = 0; i < m.rows; i++) {
        for(j = 0; j < m.cols; j++)
            printf("%4d ", m.data[i][j]);
        printf("\n");
    }
}
```

### 13.3.5 Using Structure Containing Arrays as Function Parameter and Return Value

We can use an entire structure containing arrays as a function parameter. However, modifications to the structure parameter including the member arrays will not be reflected in the corresponding arguments in the called function. This might be confusing, particularly for a beginner, since arrays are passed by reference. However, understand that we are passing an entire structure (that contains arrays) and not an array. In addition, we can also return a structure containing arrays from a function.

As we already know, passing or returning a structure containing arrays will of course be very inefficient due to the involved structure copy operation. We should instead use the efficient *pointer-to-structure* mechanism. Finally, note that if we pass only the contained array member to a function, it will be passed by reference.

#### Example 13.10 Structure containing arrays as function parameter and return value

Consider the definition of function `vector_print` given below used to print a size-aware vector declared using structure `vector` (see Example 13.6c).

```
void vector_print(struct vector v)
{
    int i;

    for (i = 0; i < v.len; i++)
        printf("%d ", v.data[i]);
    printf("\n");
}
```

Observe how the vector length (`len`) and the individual elements of the `data` array are accessed within the function using the *dot* operator. This function can be called as shown below.

```
struct vector v = {5, {1, 3, 5, 7, 9}};
vector_print(v);
```

## 13.4 Nested Structures

The C language permits structures to be nested, in which a structure can contain structure members. This concept can be easily extended to declare nested structures of higher level. The use of such nested structures facilitates representation and processing of complex data.

### 13.4.1 Declaration of Nested Structures

Consider that we wish to represent the following information of a person: salutation, full name, gender and date of birth. This can be done in a straight-forward way as shown below.

```
struct person {
    char salut[5];
    char lname[20], fname[20], mname[20];
    char gender;
    int dob_dd, dob_mm, dob_yy;
};
```

However, this is an awkward implementation as the date of birth is represented using three fields. This approach will soon be tedious and cluttered if we also wish to store information about more dates. A better way is to use nested structures and define date of birth as a nested structure member as shown below.

```
struct date {
    int dd, mm, yy;
};

struct person {
```

```
char salut[5];
char lname[20], fname[20], mname[20];
char gender;
struct date dob;
};
```

Observe that the structure **date** is declared before it is used in structure **person** to declare member **dob**. Now consider a variable **p** of type **struct person**:

```
struct person p;
```

The members of contained structures can be accessed using the usual syntax that involves the use of the dot (.) operator. Thus, the **dd**, **mm** and **yy** fields of the date of birth of this person (**p.dob**) can be accessed as **p.dob.dd**, **p.dob.mm** and **p.dob.yy**, respectively.

It is also possible to declare structure **date** within the declaration of structure **person** as shown below.

```
struct person {
    char salut[5];
    char lname[20], fname[20], mname[20];
    char gender;
    struct date {
        int dd, mm, yy;
    } dob;
};
```

Although structure **date** is declared inside **person**, we can subsequently use it to declare members in other structures, structure variables as well as function parameters and return types.

Note that we can also use anonymous structure declaration, i. e., omit the name of the structure as shown below. However, we cannot use this inner structure in subsequent parts of the program.

```
struct person {
    char salut[5];
    char lname[20], fname[20], mname[20];
    char gender;
    struct {
        int dd, mm, yy;
    } dob;
};
```

### Example 13.11 Nested structures

- a) Nested structure to represent a rectangle

Consider now the representation of a rectangular region on the screen. Although a rectangle has four vertices, we can use a pair of opposite vertices (e. g. top-left and right-bottom) to represent it. Thus, we can declare a structure to represent a rectangle on screen as shown below.

```
struct rect {  
    double x1, y1, x2, y2;  
};
```

A better approach to declare this structure using nested structures is shown below.

```
struct point {  
    double x, y;  
};  
  
struct rect {  
    struct point lt, rb; /* top-left & right-bottom corners */  
};
```

Note that instead of using *x* and *y* coordinates as members, we now declare the `rect` structure more naturally in terms of the two corner points (i. e., variables of type `struct point`). Now if we define a variable `r1` of type `struct rect`, the *x* and *y* coordinates of its top-left corner can be accessed as `r1.lt.x` and `r1.lt.y`, respectively.

### b) Improved structure to represent a person

Let us again consider the structure `person` declared above. Observe that a person's name is represented using four members (`salut`, `lname`, `fname` and `mname`). We can further simplify this structure by combining these four name fields in another structure as shown below.

```
struct date {  
    int dd, mm, yy;  
};  
  
struct name {  
    char salut[5];  
    char last[20], first[20], middle[20];  
};  
  
struct person {  
    struct name name;  
    char gender;  
    struct date dob;  
};
```

```
struct person p;
```

The `name` structure, which is used to represent a person's name, declares four members: `salut`, `last`, `first` and `middle`. The person's name is also referred to in the `person` structure as `name`. We could have used some other name for this structure, say `person_name`. However, the use of the identifier `name` for the structure as well as the member is quite obvious and such usage is permitted in C. Note that the individual components of a person's name (`p.name`) can be accessed as `p.name.last`, `p.name.first`, etc.

### 13.4.2 Initializing Nested Structures

A nested structure is initialized by using the usual structure initialization syntax for the structure itself as well as for the contained structure members. Thus, in the general format for initialization of a structure given below

```
struct tag var = { expr1, expr2, ... };
```

we replace the initializer expression, corresponding to a structure member, with a structure initializer list enclosed in braces.

For example consider the improved declaration of structure `person` in Example 13.11b that used the structures `name` and `date` to represent the person's name and date of birth. We can initialize a variable of this structure as shown below.

```
struct person p = {
    {"Miss", "Bichkar", "Shalmali", "Rajan"}, 'F', {27, 8, 2006}
};
```

Note that other rules for structure initialization are applicable for the initialization of nested structures as well. In addition, we can also omit values for one or more members at the end of each contained structure as well. The compiler will initialize these members with default values. Consider, for example, the initialization statement given below which omits the date of birth and middle name of a character from a popular Hindi movie.

```
struct person gabbar = { "", "Singh", "Gabbar"}, 'M';
```

The middle name is initialized with a null value and all the components of date of birth are initialized to zero.

### 13.4.3 Processing Nested Structures

We have already seen that the nested structure members as well as individual members of nested structure members are accessed using the dot (.) operator. The nested structure members can be used as

function parameters and return values. Also, the individual members of contained structures can be used in expressions, passed to functions, etc.

### Example 13.12 Processing nested structures

Consider the function `person_print` given below to print information about a person available in the `person` structure (see Example 13.11b).

```
/* print a person's info */
void person_print(const struct person p)
{
    printf("Name : %s %s %s\n", p.name.salut, p.name.last,
           p.name.first, p.name.middle);
    printf("Gender : %c\n", p.gender);
    printf("Date of birth: %d/%d/%d\n", p.dob.dd, p.dob.mm, p.dob.yy);
}
```

Observe how individual components of contained structures (`name` and `dob`) are accessed using the dot operator (`p.name.last`). However, the code is bit cluttered. We can simplify this function by adding two functions, namely, `name_print` to print a person's name and `date_print` to print a date as shown below.

```
/* print a person's info */
void person_print(const struct person p)
{
    printf("Name : ");
    name_print(p.name);
    printf("Gender : %c\n", p.gender);
    printf("Date of birth : ");
    date_print(p.dob);
}

/* print a person's name */
void name_print(const struct name n)
{
    printf("%s %s %s\n", n.salut, n.last, n.first, n.middle);
}

/* print a date */
void date_print(const struct date d)
{
    printf("%d/%d/%d\n", d.dd, d.mm, d.yy);
}
```

---

However, remember that passing (and returning) large structures to (from) a function is quite inefficient and we should use the pass by value mechanism using pointers to structures as discussed in the next section.

## 13.5 Advanced Concepts

This section presents several advanced concepts on structures that include memory organization of structures, pointer to a structure and its use in passing and returning values to and from a function, array of structures, representing complex data using arrays and structures and structures containing pointer members.

### 13.5.1 Memory Organization of Structures

The members of a structure are stored in consecutive memory locations in the order specified in its declaration. However, a compiler may introduce small memory gaps between structure members so as to optimize their access. Thus, it is not a good idea to add the sizes of individual members of a structure to determine the total memory required for a structure variable. Instead, we should use the `sizeof` operator to determine the size of a structure or a structure variable. For example, if we have a structure named `employee` and a variable `emp` of type `struct employee`, we can either use `sizeof(struct employee)` or `sizeof(emp)` to determine the memory required to store such a variable.

Since the memory location of structure members is likely to be machine and compiler dependent, we should avoid accessing structure members directly from memory using low-level pointer operations. Instead, we should use the structure member operators (`.` and `->`).

However, note that the elements of a member array of a structure will be stored in consecutive memory locations, without any memory gap between individual array elements. Also, if we create an array of structures, the individual elements in this array will also be stored in consecutive memory locations without any memory gap between these elements. Thus, if an array `emp_arr` of type `struct employee` has  $n$  elements, the total memory required for this array can be calculated as  $n * sizeof(struct employee)$  or even as  $n * sizeof(emp\_arr[0])$ .

### 13.5.2 Pointer to a Structure

As we already know, a pointer is a very powerful feature of C language. In this section, we study the use of a pointer to a structure to implement the *pass by reference* mechanism for passing (returning) structures to (from) a function. This approach is very efficient compared to the default pass by value mechanism which involves structure copy.

#### Declaring a Structure Pointer

Consider the declaration of structure `point` given below.

```
struct point {  
    int x, y;  
};
```

We can declare a variable and a pointer of type `struct point` as

```
struct point a, *pa;
```

We can make pointer `pa` to point to variable `a` by assigning to it the address of `a` as

```
pa = &a;
```

Now we can use pointer `pa` to access the object pointed to by it (variable `a`) using the *pointer dereference* operator (\*) as `*pa`. We can also use pointer `pa` to access individual members of structure variable `a` using the usual dot operator as shown below for member `x`

```
(*pa).x
```

Note that the parentheses in the above expression are essential as the dot operator has higher precedence than the \* operator. Thus, the expression is tedious to write. Hence, C language provides a convenient shorthand for this expression using the -> **operator** as

```
pa->x
```

The precedence of the -> operator is the same as that of the dot operator and is higher than that of most other operators. Thus, we do not require any parentheses when we use the above construct in an expression. For example, increment, decrement and address-of operations can be performed on member `x` of structure `a` as

```
++pa->x      --pa->x      pa->x++      pa->x--      &pa->x
```

### Pointer to a Structure as a Function Parameter

As we know, the default pass by value mechanism used to pass a structure to a function is inefficient, particularly for large structures. It also makes the structure parameter input-only parameter. Thus, we have to use the return value mechanism to return the modified structure to the calling function. This operation is also inefficient and allows only one structure to be returned. We can overcome these problems by using a pointer to a structure as a function parameter.

Consider the `point_print` function given below to print a `point` structure having two members `x` and `y` of type `int`.

```
void point_print(struct point *p)
```

```
{  
    printf("(%d, %d)", p->x, p->y);  
}
```

This function can be called from some function, say `main`, as shown below.

```
int main()  
{  
    struct point a = {1, 2};  
    point_print(&a);  
}
```

When the `point_print` function is called, the address of structure variable `a` is assigned to pointer parameter `p`. Thus, `p` points to variable `a` in the `main` function. We can now use `p` to access the individual members of structure variable `a` (which is in the calling function) as `p->x` and `p->y`. The `point_print` function just prints these values.

Note that the use of a structure pointer parameter is very efficient as it involves copying a single pointer compared to copying the complete structure required in the default pass by value mechanism.

Now consider the `point_read` function given below used to read a `point` structure from the keyboard.

```
void point_read(struct point *p)  
{  
    scanf("%d %d", &p->x, &p->y);  
}
```

This function takes a pointer to the `point` structure as a parameter and uses it in the `scanf` function call to read the values for `x` and `y` members of the argument structure in the calling function. The values are stored in the argument structure and will be available in the calling function.

You might think that the address-of (`&`) operator is not required in the `scanf` function call as `p` is a pointer. However, note that `p->x` refers to a variable of type `int` (member `x` of structure `point`). Hence, the address of operator is required.

Observe that this function is efficient, compared to one that returns a structure as a function value, as it does not involve any structure copy operation.

Now consider again the `point_print` function given above. It does not (and inadvertently should not) modify the argument structure passed to it. However, the pointer parameter `p` allows us to modify the argument structure in the calling function. We can use a `const` pointer parameter to avoid such inadvertent modifications to the argument structures as shown below.

```
void point_print(const struct point *p)
```

```
{  
    printf("(%d, %d)", p->x, p->y);  
}
```

Now if we inadvertently modify the argument using pointer **p** (including the modifications in functions called from this function), the compiler will report an error. This prevents difficult-to-trace bugs in our program. It is good programming practice to use this feature.

#### Example 13.13 Pointer to a structure as a function parameter

##### a) Function to print a size-aware vector

The declaration of a structure to represent a size-aware vector was given in Example 13.6c and a function to print such a vector was given in Example 13.10 using the default pass-by-value mechanism. We can improve the efficiency of this function by using a pointer to a structure as shown below. Note that **pv** is declared as a pointer to **const struct vector**, to avoid inadvertent changes to the argument structure.

```
void vector_print(const struct vector *pv)  
{  
    int i;  
  
    for (i = 0; i < pv->len; i++)  
        printf("%d ", pv->data[i]);  
    printf("\n");  
}
```

##### b) Function to read a size-aware matrix from the keyboard

Consider the function **matrix\_read** given below used to read a size-aware matrix (see Example 13.6c for declaration of **struct matrix**).

```
void matrix_read(struct matrix *pm)  
{  
    int i, j;  
  
    printf("Enter matrix size: ");  
    scanf("%d %d", &pm->rows, &pm->cols);  
  
    printf("Enter matrix elements:\n");  
    for(i = 0; i < pm->rows; i++) {  
        for(j = 0; j < pm->cols; j++)  
            scanf("%d", &pm->data[i][j]);  
    }  
}
```

```
    }  
}
```

This function accepts a pointer to a structure and reads the data for a matrix in the argument structure. It can be called as shown below.

```
struct matrix a;  
matrix_read(&a);
```

#### c) Function for adding two size-aware matrices

Consider now the `matrix_add` function to add two size-aware matrices (see Example 13.6c for the declaration of `struct matrix`).

```
/* addition of size-aware matrices */  
int matrix_add(const struct matrix *a, const struct matrix *b,  
               struct matrix *c)  
{  
    int i, j;  
    if (a->rows != b->rows || a->cols != b->cols)  
        return 0; /* addition not possible */  
  
    /* perform addition operation on matrix elements */  
    for(i = 0; i < a->rows; i++) {  
        for(j = 0; j < a->cols; j++)  
            c->data[i][j] = a->data[i][j] + b->data[i][j];  
    }  
  
    c->rows = a->rows; /* set matrix size for addition matrix */  
    c->cols = a->cols;  
    return 1;  
}
```

This function has three pointer parameters, `a`, `b` and `c` of type `struct matrix *`. If the sizes of matrices `a` and `b` are unsuitable for addition, it returns 0 indicating failure. Otherwise, it adds matrices `a` and `b` in matrix `c`, sets the size of matrix `c` and returns 1, indicating success.

### 13.5.3 Array of Structures

The C language permits the declaration of an array of structures, i. e., an array in which each element is a structure of the same type. We can declare one-, two- or multidimensional arrays of structures. The syntax for the declaration of a one-dimensional array of structures is shown below.

```
struct tag arr [ expr ] ;
```

where *arr* is an array of structures and *expr* is a constant expression that specifies its size. It is assumed that the structure named *tag* has already been defined and is accessible.

We can generalize the above syntax to declare a multidimensional array of structures as

```
struct tag arr [ expr1 ] [ expr2 ] ... [ expr n ] ;
```

Note that we can declare multiple arrays of structures (of any dimension) in a single declaration. Also, we can declare an array of a structure as a member of another structure.

### Accessing the Elements of an Array of Structures and Their Members

Each element in an array of structures is a structure. An element in one-dimensional array *arr* of structures can be accessed as *arr[i]*. Similarly, an element of a two-dimensional array *arr* of structures can be accessed as *arr[i][j]* and so on. We can perform several operations on an array element that we can perform on a structure variable. Thus, we can assign it to another structure of same type, pass it to a function as an argument, etc.

We can access a member of an element of array *arr* of a structure as

*arr[i].member*

We can perform various operations on this elements depending on its type. As the precedence of the array subscript operator and dot operator are higher than that of most other operators, we do not require any parentheses for member access. In particular, the increment, decrement and address-of operations can be performed on these members as

<b>++ arr[i].member</b>	<b>arr[i].member++</b>	<b>&amp;arr[i].member</b>
<b>-- arr[i].member</b>	<b>arr[i].member--</b>	

### Example 13.14 Array of structures

Consider a polygon having *n* vertices. We can represent each vertex in a polygon using a structure *point* declared as follows:

```
struct point {  
    float x, y;  
};
```

Assuming that the polygon will have a maximum of 20 vertices, we can now declare an array of this structure to store the vertices of a polygon as

```
struct point polygon[20];
```

Assuming that the number of vertices of a polygon is stored in variable `n`, we can read the data for all the vertices from the keyboard using the following code:

```
for (i = 0; i < n; i++) {  
    printf("Enter vertex %d: ", i);  
    scanf("%f %f", &polygon[i].x, &polygon[i].y);  
}
```

### Initialization of Array of Structures

Consider the general format given below for the initialization of an array.

```
type arr [ size ] = { expr1, expr2, ... };
```

To initialize an array of structures, the array `type` in the above format is first replaced with `struct tag` and each expression in the initialization list is replaced by structure initializers. Thus, the general format to initialize an array of structures is as follows:

```
struct tag arr [ size ] = {{ expr1a, expr1b, ... }, { expr2a, expr2b, ... }, ... };
```

The usual rules of array and structure initialization are applicable here. Thus, we can omit initializers for one or more array elements at the end of an array. We can also omit initializers for one or more members in the initialization of individual structures. Note that the uninitialized members will be initialized with default values.

For example, we can initialize a `hexagon` in an array of `point` structure as shown below.

```
struct point hexagon[] = {{0,1}, {1,2}, {2,2}, {3,1}, {2,0}, {1,0}};
```

### Program 13.4 Determine word frequency in a given string

Write a program to determine the frequency of words in a given string containing English text with usual punctuation.

**Solution:** A program to determine the number of words and average word length is given in Program 12.4. It uses the `strtok` function to separate the words in a given string. The program given below uses the same technique to separate the words in a given string and determine and print the frequency of these words.

The program uses a structure named `word` to store a word and its count. The word string is stored in an array of 20 characters, which is adequate for most words in the English language. The `main` function uses an array of this structure, named `words`, to store the distinct words in the given string and their counts. It is assumed that the given string may have at most 100 distinct words.

```
/* Determine word frequency in a given string */
```

```

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

struct word {
    char str[20];      /* word string: assume max 19 characters */
    int count;          /* word count */
};

int word_freq(const char *str, struct word words[]);

int main()
{
    char str[] = "Alexander said, \"I came, I saw, I conquered!\"";
    struct word words[100];           /* assume max. 100 distinct words */
    int nword;                      /* no of words */
    int i;

    printf("Given string:\n%s\n", str);
    nword = word_freq(str, words);
    puts("\nWord frequency:");
    for(i = 0; i < nword; i++)
        printf(" %s: %d\n", words[i].str, words[i].count);

    return 0;
}
/* calculate frequency of words in a given string */
int word_freq(const char *str, struct word words[])
{
    char punct_str[] = ".,:;!?'\""; /* punctuator list */
    char *tmp_str;                 /* pointer to a copy of given string */
    char *wptr;                    /* pointer to a word */
    int nword;                     /* number of distinct words */
    int i;

    nword = 0;
    tmp_str = strdup(str);         /* copy of given string */
    wptr = strtok(tmp_str, punct_str); /* get ptr to first word */

    while (wptr != NULL) {
        /* search current word in 'words' array */
        for(i = 0; i < nword; i++) {
            if (strcmp(wptr, words[i].str) == 0)

```

```

        break;           /* current word found, stop search */
    }

/* if current word is not in words array, add it at loc nword */
if (i < nword) /* current word already in 'words' array */
    words[i].count++; /* increment its count */
else {           /* current word not in 'words' array */
    strcpy(words[nword].str, wptr); /* add word at pos. nword */
    words[nword].count = 1;          /* set freq count to 1 */
    ++nword;                      /* increment words count */
}

wptr = strtok(NULL, punct_str); /* get ptr to next word */
}
free(tmp_str); /* release memory allocated to tmp_str */
return nword;
}

```

The `main` function first initializes character array `str` with a string literal. It then displays this string and calls the `word_freq` function to determine the distinct words in string `str` and their counts in array `words`. The contents of this array are then printed.

The `word_freq` function accepts two parameters: `str` (the string to be processed) and `words` (an array of `struct word`). It returns an integer value representing the count of distinct words in the given string. Since the `strtok` function modifies the string being processed, the `word_freq` function creates a duplicate of string `str` using the `strdup` function and operates on this string. A character pointer `tmp_str` is used to point to this string.

A character array `punct_str` is used as a string of punctuation characters and is initialized with the string literal ". , ; : ! ? ' \ \" ". This string is used by the `strtok` function to separate the words in string `tmp_str`.

The `word_freq` function uses a local variable `nword` as a counter of distinct words in string `str`. Initially, this counter is initialized to zero and the `strtok` function is used to separate the first word in string `tmp_str`. A character pointer `wptr` is used to point to this word. Then a `while` loop is setup to process the entire string.

In each iteration of the loop, first the current word pointed to by `wptr` is searched for in the `words` array. If it is found, its count is incremented; otherwise, it is added to the `words` array as a new distinct word, its count is set to 1 and the `nword` counter is incremented. Then, the `strtok` function is called to separate the next word in string `tmp_str`.

The program output is given below.

```
Given string:  
Alexander said, "I came, I saw, I conquered!"  
  
Word frequency:  
 Alexander: 1  
 said: 1  
 I: 3  
 came: 1  
 saw: 1  
 conquered: 1
```

## Dynamic Memory Allocation

In Chapter 11, we studied how memory can be dynamically allocated for an array. This applies to an array of structures as well. For example, consider the declaration of a structure `point` and a pointer `polygon` as shown below.

```
struct point {  
    float x, y;  
};  
  
struct point *polygon;
```

We can dynamically allocate memory for  $n$  vertices in a polygon using the `malloc` function as shown below.

```
polygon = (struct point *) malloc(n * sizeof(struct point));  
if (polygon == NULL) {  
    printf("Error: Out of memory ...\\n");  
    exit(1);  
}
```

Now `polygon` can be used as an array having  $n$  elements of type `struct point`. Note that it is also possible to dynamically allocate memory for a single structure variable.

### 13.5.4 Representing Complex Data Using Arrays and Structures

Consider the declaration of a structure that contains an array `arr` having 10 elements:

```
struct tag {  
    ...  
    type arr [10];  
    ...
```

```
};
```

We can declare an array of this structure using the usual notation:

```
struct tag struct_arr [50];
```

Now we can access a structure at position  $i$  in this array as  $struct\_arr[i]$ . Each member of this structure including the array member can be accessed as  $struct\_arr[i].member\_name$ .

Also, we can access the  $j$  th element of member array **arr** as

```
struct_arr [i] .arr [j]
```

To initialize an array of structure containing arrays, we use the syntax for array initialization given below where each initializer expression corresponds to a structure containing an array.

```
type arr_name [arr_size] = { expr1, expr2, ... };
```

Consider that we wish to store the following information for a book: **title**, names of at most three **authors** and its **cost**. The declaration of structure **book** is given below.

```
struct book {
    char title[60];
    char authors[3][20];
    int cost;
};
```

We can initialize an array of this structure as shown below.

```
struct book my_books[] = {
    {"Programming with C", {"Brian Kernighan", "Dennis Ritchie"}, 200},
    {"Programming in ANSI C", {"Ram Kumar", "Rakesh Agrawal"}, 150},
    {"Let us C", {"Yashwant Kanetkar"}, 250}
};
```

The C language allows data structures of arbitrary complexity to be declared. For example, we can use a variable (or an array) of the structure declared above in another structure which can in turn be used in another structure and so on.

### 13.5.5 Structure Containing Pointer Members

The C language permits the use of a pointer variable as a structure member. These pointer members enable the use of dynamically allocated arrays, thus eliminating the drawbacks of static arrays as discussed in Chapter 11. Moreover, these pointer members also enable us to create advanced data structures such as linked lists, trees, graphs, etc.

Remember that when a structure variable is created, memory is allocated only for the pointer member; the memory required to store array elements must be dynamically allocated during program execution.

Consider the declaration of structure **person** given below that uses a character pointer instead of a character array to store the name of a person.

```
struct person {  
    char *name;      /* person's name */  
    char gender;    /* M: male, F: female */  
    int age;  
};  
  
struct person p;
```

To store the data of a person in structure variable **p**, we have to first allocate the required memory to pointer member **name** using a dynamic memory allocation function such as **malloc**. Fig. 13.1 shows a structure variable containing data. When variable **p** is no longer required, we must free the memory allocated to member **name** using the **free** function.

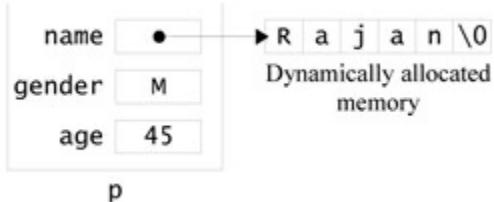


Fig. 13.1

#### Example 13.15 Structures containing pointer members

##### a) Structure to store information about a book

```
struct book {  
    char *title;        /* book title */  
    char **authors;    /* authors' names */  
    char *publisher;   /* publisher name */  
    char isbn_no[20];  /* ISBN number */  
    float price;  
    int pub_year;     /* publication year */  
};
```

This structure uses two character pointers, **title** and **publisher**, to store the book title and

publisher name, respectively. The member **authors** is declared as a pointer to pointer to **char**, i.e., **char \*\***. It can be used to store the names of any number of authors.

### b) Dynamic vector and matrix

In Example 13.6c, we declared a structure to represent a size-aware vector and matrix with the sizes of their **data** arrays as 20 and  $10 \times 10$ , respectively. While using such structures containing static arrays, there will be some wastage of memory. Moreover, the array sizes used are quite restrictive and not very suitable in practical situations. This problem can be overcome to some extent by increasing the size of the **data** array. However, it will cause a substantial increase in memory wastage. Both these problems can be completely overcome by declaring a dynamic vector and matrix as shown below.

```
struct dvec { /* dynamic vector */
    int len;
    int *data;
};

struct dmat { /* dynamic matrix */
    int rows, cols;
    int **data;
};
```

### Program 13.5 Program to read and print information about a person

The program given below reads the information about a person and prints it on the screen. It uses a structure containing a pointer member **name** to represent the information about a person. Note that the program uses the **dstr\_read** function presented in Chapter 11 to read a dynamically allocated string.

```
/* Program to read and print info of a person.
Uses a structure with char pointer to store person's name */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct person {
    char *name;      /* person's name */
    char gender;     /* M: male, F: female */
    int age;
};

char *dstr_read();
void person_read(struct person *p);
void person_print(struct person *p);
```

```
int main()
{
    struct person p;

    person_read(&p);
    person_print(&p);
    free(p.name);
    return 0;
}

/* read a dynamically allocated string */
char *dstr_read()
{
    char buf[100];
    char *str;

    gets(buf);      /* read a string in buffer */

    /* allocate required memory */
    str = (char *) malloc(strlen(buf) + 1);
    if (str == NULL) {
        printf("Error: Out of memory ...\\n");
        exit(1);
    }

    return strcpy(str, buf); /* copy string and retrun ptr */
}

/* read a person's info */
void person_read(struct person *p)
{
    printf("Enter person's information:\\n");
    printf("Name: ");
    p->name = dstr_read();
    printf("Gender (M/F): ");
    scanf(" %c", &p->gender);
    printf("Age : ");
    scanf("%d", &p->age);
}

/* print a person's info */
void person_print(struct person *p)
```

```
{  
    printf("Name : %s\n", p->name);  
    printf("Gender: %s\n", p->gender == 'M' ? "Male" : "Female");  
    printf("Age : %d\n", p->age);  
}
```

## Exercises

1. Answer the following questions in brief.
  - a. What is the basic difference between an array and a structure?
  - b. Is it possible to declare two structures with the same name?
  - c. Can we declare two structure with different names but identical members?
  - d. Which operators are used to access the members of a structure?
  - e. If **x** is a structure variable and **y** is its member, what is the difference between **++x.y** and **++(x.y)**?
  - f. What is the limitation of an anonymous structure?
  - g. What do you mean by a nested structure?
  - h. If a data item from a structure variable is accessed as **a.b[i].c**, explain what **a**, **b** and **c** are.
  - i. How is a structure variable different from an array with respect to its use as a function parameter?
2. State whether the following statements are true or false and rewrite the false statements correctly.
  - a. The members of a structure may be of either same or different types.
  - b. We can perform operations on structure variables using various operations such as arithmetic, relational, equality, assignment, etc.
  - c. When a structure is declared, the compiler allocates memory for all its members.
  - d. As the C language uses call by value mechanism to pass structures to a function, entire argument structure is copied to the parameter during a function call.
  - e. A nested structure is a concept in which a structure is declared inside another structure.
3. Write code segments to declare the structures and structure variables given below and initialize the variables with specified values, if any.
  - a. structure **time** to represent time values (hh:mm:ss), variable **midnight** with initial value 00:00:00.
  - b. structure **pixel** to represent a screen pixel with three members: **x**, **y** and **color** (all of type **int**), variables **center** with value (320, 240, 3) and **p** (no initial values)

- c. structure **employee** to represent an employee of an organization with the following structure members: **id**, **name**, **department**, **designation**, **dob** (date of birth) and **doj** (date of joining). Use structure **date** to declare **dob** and **doj** and suitable data types for other members.
- f. Identify errors, if any, in the program segments given below and rewrite them correctly.
- ```
struct pixel {
    int x, y;
    int color;
}
p = {100, 100, 5};
```
  - ```
struct date { int d, m, y};

date d1 = {25, 4, 2012}, d2;
```
  - ```
struct car {
    char make, model;
    int engine_capacity;
    int seating_capacity;
};
struct car = {"Maruti", "SX4",
    1600, 5};
```
  - ```
struct animal {
    char name[20];
    int legs;
} = ("Dog", 4);
```
5. Using suitable structure declarations, write functions for the following problems.
- Compare two given dates.
  - Multiply two complex numbers and return the result.

## Exercises (Advanced Concepts)

- Answer the following questions in brief.
  - Consider a structure variable **x** having **m** as one of its members. If **px** is a pointer to variable **x**, write the expressions to increment the value of **m** using the prefix and postfix increment operators.
  - What is the default mechanism used to pass a structure to a function?
  - Explain how a structure can be passed efficiently to a function?
- State whether the following statements are true or false and rewrite the false statements correctly.

- a. The elements of a structure are stored in the memory in the order of their declaration.
  - b. The memory required for a structure is equal to the sum of the memory required for its members.
  - c. A structure can be passed to a function efficiently using pass by reference mechanism.
  - d. We can declare a pointer to a structure but it is not possible to declare a pointer member.
  - e. We cannot initialize a structure variable containing pointer members using the initialization syntax.
3. Write C programs for the following problems.
- a. Initialize the data of several employees and print it in a tabular format. Use function `emp_print` to print the data of a single employee.
  - b. Create and print a list of persons and their mobile number. Use nested structures and pointer members.
  - c. Write a program to perform following operations on complex numbers: addition, subtraction, multiplication, division and magnitude. Write function for each operation and use call by reference mechanism to pass structures to these functions.
  - d. Consider the declaration of structure `book` given in Example 13.15a. Write a program that reads and prints the data of several books. Use functions `read_book` to read the data of a book and `print_book` to print it.
  - e. Consider the declaration of dynamic matrix (`dmat`) given in Example 13.15b. Write a menu-driven program to perform various operations on matrices that include transpose, addition, multiplication and inverse.

# 14 Files

Several real-life applications such as employee pay roll, library automation, inventory control, banking applications, etc. involve the processing of voluminous data. This includes storing, retrieving and updating data. Such data is usually split into several files and stored on auxiliary devices such as magnetic disks and tapes. In this chapter, we study how to process the data stored in files. We begin with the basics of file systems: files and streams. Then we take a quick look at the facilities provided in the standard I/O library for file processing. These include the functions to create files, open existing files, perform I/O operations on files, etc. Next we study the functions used to access files (`fopen` and `fclose`), perform character I/O operations (`fgetc`, `getc`, `fputc`, `putc`, `fgets`, `fputs` and `ungetc`) and formatted I/O operations (`fscanf` and `fprintf`).

The *Advanced Concepts* section presents the functions for direct I/O (`fwrite` and `fread`) and for file positioning (`fseek`, `ftell`, `rewind`, `fgetpos` and `fsetpos`).

## 14.1 File Basics

### 14.1.1 What is a File?

A **file** is a logical unit of data used to store related information. The information on secondary storage such as a hard disk, magnetic tape, CD ROM, etc. is arranged in the form of files. The operating system and various applications located on the computer, including program development environments, are all stored as files. The Microsoft Windows operating system files, for example, are usually stored in the `C:\WINDOWS` or `C:\WINNT` directory, whereas the files of the Code::Blocks development environment are usually stored in the `C:\Program Files\codeBlocks` directory.

We create files in various applications such as Microsoft Word, Excel, Powerpoint, Paint, Notepad, Wordpad, etc. For example, a letter or a resume created in Microsoft Word or Libre Office is a file that is usually stored on a hard disk. This file can be subsequently opened and modified if desired. Each time we modify the file, we must save the changes to the disk file. The C programs that we create are also stored as files on a hard disk.

Each file has a name associated with it. The conventions for naming a file are operating system dependent. For example, in MS DOS, a **file name** is restricted to the form `pppppppp.EEE`, where `PPPPPPPP` is a 1-8-character primary name and `EEE` is an optional 1-3-character extension. Since Turbo C/C++ is a DOS-based software, it uses this naming convention. Other operating systems such as

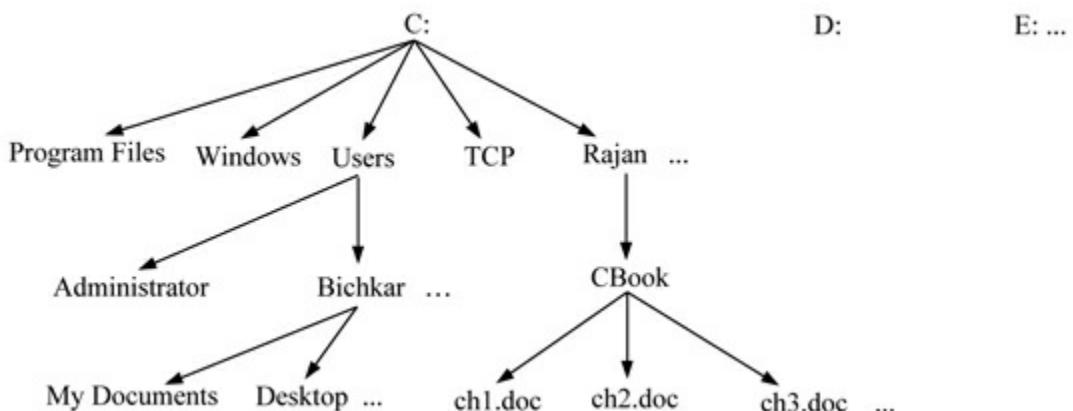
MS Windows and Linux permit larger file names that may contain spaces. Note that the file extension is used to identify the type of the file, e. g., .doc for MS Word document files, .C for C program files, .obj for object files, .exe for executable files and so on.

The information on an auxiliary device such as a hard disk is stored in **tree-structured directories**, as shown in Fig. 14.1, for MS Windows OS. The complete file specification takes the following format:

*drive:\filepath\filename*

where *drive* is a drive letter, *filename* is the name of the file and *filepath* is a path to the file. The *filename* may contain a file extension along with the primary name. The *filepath* is a sequence of directory names starting from the root directory and proceeds up to the directory in which the file *filename* is stored, separated by the backslash \. For example, the filepath of the file ch1.doc is \Rajan\CBook and its complete specification is C:\Rajan\CBook\ch1.doc.

While processing files, we are primarily concerned with **data files** – the files containing data for an application. For example, in an application that processes students' marks, we are concerned with examination-related information for each student. This data file will typically contain, for each student, the examination seat number, name, marks obtained in all subjects, total marks, percentage marks, result (Pass/Fail), etc. On the other hand, in a graphics editor application, the data file may comprise parameters of graphics objects (such as lines, circles, polygons, etc.) drawn by the user in a particular drawing.



**Fig. 14.1** Tree structured file system in Microsoft Windows OS

In most applications, we work on specific types of files. For example, imaging software works on image files, whereas a C beautifier program takes a C program file as input and writes a properly formatted program to an output file (or display). However, in some applications, we may have to deal with several types of files. For example, in file compression software, such as *WinZip*, we will be concerned with all types of files including text, executable and other system files. Another similar software is a CD-burning software such as *Nero Express*, in which we have to write different types of files to a CD/DVD.

## Types of Files

A file is a sequence of characters (e.g., ASCII values). Depending on the contents of a file, there are two types: *text* and *binary*. A **text file** stores textual information. On the other hand, a **binary file** stores the internal (binary) representation of data.

The C language provides facilities to process both types of files. Although a file created in text mode can be opened in binary mode and vice-versa, this should usually be avoided as interpretation of the file contents will be different in these modes.

### **Text files**

As the name indicates, a **text file** contains textual information. It usually contains printable characters such as letters, digits, special symbols, etc. and some special characters such as tab, newline, etc. Each text file has a special end-of-file marker (EOF with ASCII value 26) as the last character. This character is written to a file when a file opened for write operation is closed. The numbers are stored in a text file as a sequence of digits (and other characters). For example, an integer number 100 will be stored in a text file as three characters (1, 0 and 0).

A text file is made up of lines. Each line is a sequence of zero or more characters followed by a newline character. In the DOS and Windows operating systems, the newline is represented as a carriage return (CR) followed by linefeed (LF), whereas in UNIX and Linux, the newline is represented as a linefeed (LF) character.

A text file can usually be opened in any text editor such as Notepad, Wordpad, Turbo C editor, etc. We can also display the contents of such a file at the command prompt by using the **TYPE** command (in MS DOS and MS Windows OS) or the **cat** command (in UNIX/Linux OS).

### **Binary files**

A **binary file** is also a sequence of ASCII values. The newline character has no special significance in such a file and there is no special end-of-file marker. The file length from the directory entry of such a file is used to determine the end of such a file.

Note that unlike text files, we cannot display the contents of a binary file using text editors or by using the **TYPE** or **cat** command at the command prompt. We have to open such a file in an application that can process it (typically the application which created it).

A binary file stores the internal (binary) representation of data to a disk file. Thus, while using 2-byte integer numbers, the number 100 will be stored as 2 bytes with values 0 and **0x64**, respectively. The advantages of this approach are as follows:

1. Numeric data requires less space in a binary file than in a text file. For example, in Turbo C/C++, a variable of type **int** requires two bytes for its internal representation and it can store a value in the range -32768 to 32767. Such a variable will require only 2 bytes in a binary file irrespective of its value. However, in a text file, it will require 1–6 bytes depending on its value and we may also have to write an additional space character after each number.

2. If each record in a file has the same composition (number of fields and types), then each record has the same length. This allows random access to a file in which we can access any desired record without processing the file contents in a sequential manner.
3. The `fread` and `fwrite` functions used for binary file I/O can read/write one or more records at a time. Thus, they are convenient as well as efficient.

### Sequential and Random Access

Some applications involve sequential processing of the data in a file, e. g., student examination results and employee payroll. In the first case, the data of all the students appearing for an examination is to be processed to declare the examination results; in the second case, the data of all the employees in an organization is to be processed every month to prepare the employee payroll.

Some applications, on the other hand, require processing of data at random. For example, in a library automation application, we are required to issue a specific book to a particular library member, whereas in a banking application, we need to update the amount deposited or withdrawn in the account of a particular person.

Depending on the way the contents of a file are accessed, we can categorize them as **sequential** files and **random access** files.

### File Structure

A file usually contains several records. A record is a collection of related data items. In a text file, each line of text is considered as a record; in a binary file, the data of each entity is considered as a record. Thus, in an application to process the students' marks, the data of one student is considered as a record. A record consists of one or more fields. Each field stores a specific value such as a student's name, marks in a subject, total marks, etc.

#### 14.1.2 Streams

The devices used with a computer, such as a keyboard, monitor, printer, hard disk, magnetic tape, etc., have widely varying properties regarding data input and output. To simplify data I/O operations, the C standard library supports a simple mode of input and output based on the concept of a stream.

A **stream** is a sequence of characters. There are two types of streams: *text* and *binary*. The sequence of characters in a text stream is composed into lines, i. e., a sequence of zero or more characters followed by a newline character. The newline character has no significance in a binary stream.

The various input and output devices have widely varying speed of data transfer. For example, a dot matrix printer with a typical speed of 300 cps (characters per second) is a very slow device, whereas a CD ROM drive at 52x speed has  $150 \times 52$ , i. e., 7800 KB/s speed. The CPU, on the other hand, can process data at a much higher speed. Thus, while performing I/O operations, the CPU must wait until the device involved in the data transfer operation is ready for I/O operation. To avoid long delays and to free the CPU for other work, the concept of **buffering** is used, in which instead of performing I/O directly with a device, data is written to a memory buffer. Since memory is an electronic circuit,

buffered I/O operations are very fast. A stream may be **buffered**, **unbuffered** or **line buffered**.

To understand how buffering works, first consider that a program transfers data to an output device. The CPU first writes the data to a buffer at a very high speed. Once the buffer is filled (or a newline is encountered in a text stream), the CPU stops the output operation and is free to do other jobs. The contents of the buffer are now sent to the device at the required speed by the associated hardware. Once the buffer is empty (i. e., all the data is sent to the output device), the CPU again fills the buffer with new output data.

The reverse happens when data is read from an input device. The data from the input device is first received in a buffer. During this, the CPU is free to perform other activities. Once the buffer is full (or a newline is encountered for a text file), the CPU quickly reads the data from the buffer and continues with program execution.

### Built-in Streams

The primary devices used for user interaction are the keyboard and display. Almost every program performs some data input and output operations with these devices. Hence, the C language provides three built-in streams for performing I/O with these devices: **stdin**, **stdout** and **stderr**. Every C program has these three streams associated with it.

The standard input stream, **stdin**, is associated with the keyboard for performing input operations. The **stdout** and **stderr** streams are associated with the display. The **stdout** stream is used for the program output, whereas **stderr** is used for reporting errors.

#### 14.1.3 Standard Library Support for File Processing

The standard library provides a structure named **FILE**, several constants and numerous functions for file processing. These declarations are given in the `<stdio.h>` header file.

The **FILE structure** contains information for controlling the stream associated with the file. It includes the file position indicator, end-of-file indicator and pointer to the associated buffer. The commonly used constants are **NULL**, **EOF**, **SEEK\_SET**, **SEEK\_CUR**, **SEEK\_END**, **FOPEN\_MAX**, etc. Of particular interest is the constant **EOF**, which is the value returned by file functions when an end-of-file is encountered.

The functions for file processing can be divided into five broad categories: *file access*, *input/output*, *file positioning*, *error handling* and *file operations*. The input/output category can be further subdivided into *character I/O*, *formatted I/O* and *direct I/O*. The commonly used functions for file processing are summarized in Table 14.1.

## 14.2 File Access Functions

A file must be opened before we perform any I/O operation on it. The **fopen** function is used to open

a file. This function can also create a file if it is not present. Similarly, we must close a file after the operations on that file are completed. The **fclose** function is used to close a file. Thus, the steps for performing file I/O are:

1. Open a file (in the desired mode, as explained shortly).
2. Perform data input/output operations on it.
3. Close the file.

In addition to the **fopen** and **fclose** functions, the file access category provides two functions: **freopen** to reopen an already opened file in a different mode and **fflush** to flush the buffer associated (if any) with an open file.

#### 14.2.1 The **fopen** Function

The **fopen** function is used to open a file for I/O operations. Its prototype is given below.

```
FILE* fopen( const char *fname, const char *mode );
```

When a file is opened, a stream is associated with it and a buffer may be allocated. Also, a structure of type **FILE** is initialized and a pointer to this structure is returned by **fopen**. This pointer is used for subsequent operations on this file. However, if a file could not be opened, this function returns a **NULL** value.

The parameter *fname* represents the name of the file being opened and the parameter *mode* specifies the mode in which the file is to be opened. For example, to open a file named **NUMBERS.DAT** in the *read* mode, we can call **fopen** as follows:

**Table 14.1** Commonly used file processing functions in the C language

Category	Function	Purpose
File access	fopen	Opens a file
	fclose	Closes a file
	freopen	Changes the file associated with a stream
	fflush	Flushes the buffer associated with a file
Formatted I/O	fscanf	Performs formatted input operation on an input stream
	fprintf	Sends formatted output to an output stream
Character I/O	fgetc	Reads a character from an input stream
	fputc	Writes a character to an output stream
	ungetc	Pushes back a character to an input stream
	fgets	Reads a string from an input stream
	fputs	Writes a string to an output stream
Direct I/O	fread	Reads a specified number of data items of specified size from an input stream
	fwrite	Writes a specified number of data items of specified size to an output stream
File positioning	fseek	Repositions the file pointer of a stream
	ftell	Returns the current file pointer for a stream
	rewind	Repositions the file pointer to file beginning
Error handling	feof	Tests if end-of-file has been reached for a file
	ferror	Tests if an error has occurred for a file
	clearerr	Resets the error and end-of-file indicators for a file
	perror	Prints a system error message
Operations on files	remove	Deletes a file
	rename	Renames a file
	tmpfile	Creates a temporary binary file
	tmpnam	Generates a unique file name (usually for a temporary file)

```
FILE *fp;
fp = fopen("NUMBERS.DAT", "r");
```

The file name may contain the drive letter and file path. Thus, if NUMBERS.DAT is in the C:\STUD\DAT directory, we can write the above call using the complete file specification as

```
fp = fopen("C:\\STUD\\\\DAT\\NUMBERS.DAT", "r");
```

where a backslash character (\) is represented using a double backslash (\\\) in a string constant. Note that if the drive letter and file path are not specified, the current directory on the current drive is assumed.

A file may be opened as a text or a binary file in one of three basic modes: *read*, *write* and *append*. The behaviour of **fopen** depends on the existence of specified file and the specified mode, as follows:

- 1. If an existing file is opened in read mode, the file pointer is positioned at beginning of the file; the read operation will begin from here. However, if the specified file does not exist, no file is opened and the **fopen** function returns **NULL**. Thus, we should specify the name of an existing file for a read operation.
- 2. If an existing file is opened for a write operation, the file is opened, its contents are erased (i. e., lost forever) and the file pointer is positioned at the beginning of the file; the write operation will begin from here. However, if the specified file does not exist, the **fopen** function creates that file and positions the file pointer at its beginning. Thus, when opening a file in the write mode, we should specify the name of a new (i. e., non-existent) file unless the data in an existing file is unimportant.
- 3. In the append (or add at end) mode, which is an output mode, the data written to a file is added to its end without erasing its previous contents, if any. Thus, if an existing file is specified, it is opened for a write operation with the file pointer positioned at its end. However, if the specified file does not exist, a new file is created and the file pointer is positioned at its beginning.

The *mode* string is used to specify the type of the file and the mode of operation. A file can be opened in the **read**, **write** and **append mode** by specifying the mode string as "**r**", "**w**" and "**a**", respectively. We can also specify the '**+**' character in the mode string, as in "**r+**", "**w+**" and "**a+**", to indicate the file **update mode** in which the file is opened for both read and write operations.

The type of file is specified by appending the character **t** (for text file) or **b** (for binary file) to the mode string. Thus, the "**rt**" mode specifies a text file opened in the read mode, whereas "**w+b**" or "**wb+**" specifies a binary file opened for an update operation. However, if we do not specify the file type (text or binary) in the mode string, it is decided by the value of the **\_fmode** global variable: **O\_TEXT** for text file (which is the default value on startup) and **O\_BINARY** for a binary file. Thus, by default, the mode strings "**r**", "**w**" and "**a**" will open text files.

Note that when a file is opened for an update operation, the output operation can be followed by an input operation only after an end-of-file is encountered during the input operation. On the other hand, an input operation may be followed after an output operation only after an intervening call to the **fflush**, **fseek**, **rewind** or **fsetpos** function.

#### **14.2.2 The **fclose** Function**

When I/O operations on a file are complete, we must close the file using the **fclose** function. The prototype of this function is as follows:

```
int fclose( FILE *fp );
```

The argument *fp* is a pointer to the file to be closed. This pointer is initialized when the file is opened. The function returns a zero value when a file is closed successfully; otherwise, it returns **EOF**.

When **fclose** is called with a file opened for an output or update operation, the contents of the file buffer are flushed, i. e., they are written to the file, the buffer is de-allocated and the file is closed. On the

other hand, when `fclose` is called with a file opened for buffered input, the buffer is de-allocated (its contents are discarded) and the file is closed.

### Example 14.1 Opening and closing a file

The program segment given below illustrates how a file specified by the user is opened as a text file for input and closed after the input operations are over.

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

int main()
{
    FILE *fp;
    char fname[50];

    /* read name of the file to be opened */
    printf("Enter input file name: ");
    gets(fname);

    /* open the specified file */
    fp = fopen(fname, "rt");
    if (fp == NULL) {
        printf("Error: Unable to open file %s\n", fname);
        exit(1);
    }

    /* perform input operations on given file */
    ...

    fclose(fp);      /* close the file */
    return 0;
}
```

The variable `fp` is a pointer to a file structure and the character array `fname` is used to store the file name. The program first reads the name of the file to be opened using the `gets` function.

The `fopen` function is then used to open the specified file in *read* mode and the return value (a file pointer) is assigned to variable `fp`. If the file opening operation was unsuccessful (`fp` is `NULL`), the program displays an error message and exits with error code 1. Otherwise, program execution continues and we can perform the desired input operations on the specified file using the file pointer `fp`. Finally, when the input operations are over, the file is closed using `fclose`.

Note that we can rewrite the code for opening a file in a concise way by calling `fopen` from within

the condition in the **if** statement as shown below.

```
if ((fp = fopen("STUD.DAT", "rt")) == NULL) {  
    printf("Error: Unable to open file STUD.DAT\n");  
    exit(1);  
}
```

Note that since the assignment operator has lower precedence than the equality operator, the **fopen** function call is enclosed within a pair of parentheses. Thus, the value returned by the **fopen** function is first assigned to the variable **fp** and then compared with **NULL**. Without the parentheses, the value returned by **fopen** will first be compared with **NULL** and then the outcome of this comparison will be assigned to variable **fp**, which is incorrect.

### Example 14.2 Function to open a file

In Example 14.1, observe that the code to read a file name from the keyboard and open the file spans several lines. Since almost every program involving file processing will require one or more files to be opened, it is a good idea to write a utility functions to perform this operation. Such a function will also enable us to keep the code in subsequent example programs concise. We will generalize this function to accept the file name from either the calling function or the keyboard.

The **fileopen** function given below has three parameters: **fname**, **mode** and **msg**. The parameter **fname** usually specifies the name of the file to be opened. However, if message string **msg** is not a **NULL** string, the function displays **msg** and reads a file name from the keyboard. The specified file is then opened in the given **mode** using **fopen**. If the file is opened successfully, the function returns the file pointer; otherwise, it displays an error message and exits the program.

```
/* open a file in specified mode */  
FILE *fileopen(char *fname, const char *mode, const char *msg)  
{  
    FILE *fp;  
  
    /* if msg is not a NULL string, read file name from keyboard */  
    if (strlen(msg) > 0) {  
        printf("%s", msg);  
        scanf("%s", fname);  
    }  
  
    /* open the file */  
    if ((fp = fopen(fname, mode)) == NULL) {  
        printf("Error: Unable to open file %s\n", fname);  
        exit(1);  
    }  
}
```

```
    return fp;  
}
```

To open a text file called `marks.dat` in read mode, we can call this function as shown below.

```
fp = fileopen("marks.dat", "rt", "");
```

On the other hand, to read the file name from the keyboard and open that file as a text file in read mode, we can call this function as follows:

```
char fname[50];  
  
fp = fileopen(fname, "rt", "Enter input file name: ");
```

Note that we should copy this function in each program that uses it. This can be tedious, as we may have to add the prototype of this function at the beginning of the program. Alternatively, we can save this function in a text file, say `fileopen.c`, along with the required `#include` statements given below.

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

Now we can include this file in each program as shown below.

```
#include "fileopen.c"
```

Observe that the filename `fileopen.c` has been enclosed in double quotes, the convention to be followed for including user-defined files in a C program. We will use this approach in subsequent programs that require this function.

## 14.3 Character I/O

This category provides functions and macros for reading and writing characters and character strings. These include the `getc` and `putc` macros and the `fgetc`, `fputc`, `fgets`, `fputs` and `ungetc` functions. The `getc` and `putc` are implemented as macros for efficient operation.

The `fgetc` function and the `getc` macro read a single character from an input stream, whereas the `fputc` function and the `putc` macro write a single character to an output stream. The `fgets` and `fputs` functions read and print a character string from/to a stream. Finally, the `ungetc` function is used to push back a character to a specified input stream.

### 14.3.1 The `fgetc` Function and the `getc` Macro

The **fgetc** function and **getc** macro read a single character from a specified input stream. Their prototypes are given below.

```
int fgetc( FILE *fp );
int getc( FILE *fp );
```

The **fgetc** function reads an **unsigned char**, the next character from a specified input stream and returns it as a value of type **int**. However, if end-of-file is reached for the specified input stream, it returns **EOF**.

As already mentioned, **getc** does exactly what **fgetc** does. However, it is usually implemented as a macro for efficient operation. Hence, we may prefer the **getc** over **fgetc** function. However, the arguments of **getc** should not have side effects because its parameters may be evaluated more than once, causing incorrect program behaviour.

#### Program 14.1 Display a text file

Write a program to display the contents of a specified text file.

**Solution:** The program segment given below reads the contents of a text file and displays it on the screen. It is assumed that the desired file is opened in read mode and the file pointer returned by the **fopen** function is assigned to variable **fp**, as explained in Example 14.1.

```
ch = getc(fp);
while (ch != EOF) {
    printf("%c", ch);
    ch = getc(fp);
}
```

In this program segment, initially the **getc** macro is used to read the first character from the input stream **fp**. Then a **while** loop is set up to read and display the remaining characters, if any, from the stream, one character at a time. The **while** loop is executed as long as end-of-file is not reached. Observe that if the specified file is empty, the first character read from the file is **EOF** and the statements within the **while** loop are not executed. We can rewrite this code in a concise way as shown below.

```
while ((ch = getc(fp)) != EOF)
    printf("%c", ch);
```

This coding style is similar to that used to open a file. Note that the expression **ch = getc(fp)** is enclosed in parentheses because the assignment operator has lower precedence than the inequality operator. Thus, the character read by the **getc** function is first assigned to variable **ch**, which becomes the value of the assignment expression. This value is then compared with **EOF**. The above **while** loop can be interpreted as *while the character which is read from the stream fp and assigned to variable ch is not equal to EOF, print it on the screen*.

A complete program that uses this concise code to display the contents of a user-specified text file is given below. It first reads, in character array `fname`, the name of the file to be displayed and opens this file as a text file for read operation. Then the above `while` loop is used to display the contents of the file. Finally, the file is closed.

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

int main()
{
    FILE *fp;
    char fname[50];
    int ch;

    /* read file name */
    printf("Enter name of file to be displayed: ");
    gets(fname);

    /* open specified file as a text file in read mode */
    if ((fp = fopen(fname, "rt")) == NULL) {
        printf("Error: Unable to open file %s\n", fname);
        exit(1);
    }

    /* display contents of file fname */
    while ((ch = getc(fp)) != EOF)
        printf("%c", ch);

    fclose(fp);
    return 0;
}
```

We can greatly improve the readability of this program using the `fileopen` function from Example 14.2 as shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include "fileopen.c"

int main()
{
    FILE *fp;
    char fname[50];
```

```

int ch;

/* read file name and open it as a text file in read mode */
fp = fopen(fname, "rt", "Enter name of file to be displayed: ")

/* display contents of file fname */
while ((ch = getc(fp))!= EOF)
    printf("%c", ch);

fclose(fp);
return 0;
}

```

### Program 14.2 Text file statistics

Write a program to count the number of characters, words and lines in a text file.

**Solution:** Let us use a `while` loop in conjunction with the `getc` macro to read a text file character by character as explained in Program 14.1. The counting of characters and lines is very straightforward, the code for which is given below.

```

/* count characters and lines in text file */
nchar = nline = 0;
while ((ch = getc(fp)) != EOF) {
    nchar++;
    if (ch == '\n')
        nline++;
}

```

To count the words, we use the logic presented in Section 12.3.3. A complete program to count the number of characters, words and lines in a given text file is given below.

```

#include <stdio.h>
#include <ctype.h>
#include "fileopen.c"

int main()
{
    FILE *fp;
    char fname[50];
    int nchar, nword, nline;
    int ch, in_word;

    /* read file name and open it as a text file for reading */

```

```

fp = fileopen(fname, "rt", "Enter input file name: ");

/* count characters, words and lines in text file */
nchar = nword = nline = 0;
in_word = 0;
while ((ch = getc(fp)) != EOF) {
    nchar++;
    if (ch == '\n')
        nline++;
    if (in_word == 0) {
        if (!isspace(ch)) {
            in_word = 1;
            nword++;
        }
    }
    else if (isspace(ch))
        in_word = 0;
}
printf("Characters: %d Words: %d Lines: %d\n", nchar, nword, nline
fclose(fp);
return 0;
}

```

The program output that shows the statistics of a file containing the above program (with file function included) is given below.

```

Enter input file name: filestat.c
Characters: 1270 Words: 185 Lines: 56

```

### 14.3.2 The **fputc** Function and the **putc** Macro

The **fputc** function and **putc** macro are used to write a character to an output stream. Their prototypes are given below.

```

int fputc( int c, FILE *fp );
int putc( int c, FILE *fp );

```

The **fputc** function converts the character specified as its first argument to **unsigned char** and writes it to a specified output stream **fp**. It returns the same character as a value of type **int**. However, if an error occurs in the write operation, the function return **EOF**.

The **putc** macro does exactly what **fputc** does. However, it is usually implemented as a macro for

efficient operation. Note that the parameters of `putc` should not produce *side effects* as they may be evaluated more than once, giving undesirable output.

### Program 14.3 Copy a file

Write a program to copy a text file to another file.

**Solution:** To copy the contents of a file to another, we read the source file one character at a time and write it to the target file until the end-of-file mark is reached in the source file.

Assuming `fp_source` and `fp_target` as file pointers for source and target file, respectively, the program segment given below copies all the characters in the source file to the target file, including the EOF mark.

```
/* copy contents of source file to target file */
while ((ch = getc(fp_source))!= EOF)
    putc(ch, fp_target);
```

The complete program to copy a file is given below.

```
#include <stdio.h>
#include "fileopen.c"

int main()
{
    FILE *fp_source, *fp_target;
    char source[50], target[50];
    int ch;

    /* open source and target files */
    fp_source = fileopen(source, "rt", "Source file name: ");
    fp_target = fileopen(target, "wt", "Target file name: ");

    /* copy contents of source file to target file */
    while ((ch = getc(fp_source))!= EOF)
        putc(ch, fp_target);

    /* close the files */
    fclose(fp_source);
    fclose(fp_target);
    return 0;
}
```

### Program 14.4 Concatenate two files

Write a program to create a new file by concatenating the contents of two text files.

**Solution:** To concatenate two text files and create a new file, we open the source files in read mode and the target file in the write mode. Let the file pointers `fp_source1` and `fp_source2` be associated with the source files and file pointer `fp_target` be associated with the target file. The code to concatenate two files and create the target file is given below.

```
/* copy contents of first file to target file */
while ((ch = getc(fp_source1))!= EOF)
    putc(ch, fp_target);

/* copy contents of second file to target file */
while ((ch = getc(fp_source2))!= EOF)
    putc(ch, fp_target);
```

The complete program is given below. Note that the use of the `fileopen` function from Example 14.2 has made this program very concise and readable.

```
#include <stdio.h>
#include "fileopen.c"

int main()
{
    FILE *fp_source1, *fp_source2, *fp_target;
    char source1[50], source2[50], target[50];
    int ch;

    /* open the files */
    fp_source1 = fileopen(source1, "rt", " First Source file name: ");
    fp_source2 = fileopen(source2, "rt", " Second Source file name: ")
    fp_target = fileopen(target, "wt", "Target file name: ");

    /* copy contents of first file to target file */
    while ((ch = getc(fp_source1))!= EOF)
        putc(ch, fp_target);

    /* copy contents of second file to target file */
    while ((ch = getc(fp_source2))!= EOF)
        putc(ch, fp_target);

    /* close the files */
    fclose(fp_source1);
```

```
fclose(fp_source2);
fclose(fp_target);
return 0;
}
```

### 14.3.3 The fgets and fputs Functions

The **fgets** function reads a sequence of character, i. e., a character string from an input stream. Its prototype is given below.

```
char *fgets(char *s, int n, FILE *fp);
```

The characters from the input stream are read into a character array *s* until a newline character is read, *n* – 1 characters are read or the end-of-file is reached. If a newline character is read, it is retained in the character array. The function then adds a terminating null character to string *s* and returns it. However, the function returns a NULL value if an end-of-file is encountered before any character is read or an error occurs during input.

The **fputs** function writes a character string to an output stream. Its format is given below.

```
int fputs(const char *s, FILE *fp);
```

All the characters in string *s* except the null terminator are written to stream *fp*. The function returns the numbers of characters written to the stream or EOF if an error occurs during output.

#### Program 14.5 Display, filter and copy a file using the fgets and fputs functions

A program to display a specified text file using the **fgets** and **fputs** functions is given below.

```
#include <stdio.h>
#include "fileopen.c"

#define BUF_SZ 80

int main()
{
    char in_file[50];
    char buf[BUF_SZ];
    FILE *fp_in;

    /* open source file and display its contents */
    fp_in = fileopen(in_file, "rt", "Enter input file name: ");
```

```

while (fgets(buf, BUF_SZ, fp_in) != NULL)
    fputs(buf, stdout);

fclose(fp_in);
return 0;
}

```

If we wish to filter the given file, i. e., display only the lines that contains a specific text (given in `char` array `key`), we can do so by searching string `buf` using the `strstr` function as shown below.

```

while (fgets(buf, BUF_SZ, fp_in) != NULL) {
    if (strstr(buf, key) != NULL)
        fputs(buf, stdout);
}

```

However, remember that string `buf` should be sufficiently long to accommodate longer lines in the input file; otherwise, smaller portions of lines containing specified text will be displayed.

Note that we can copy the given input file to another file by replacing the `stdout` stream in the `while` loop (from the `main` function) by an output stream (say `fp_out`, that is associated with a text file opened in write mode) as shown below.

```

while (fgets(buf, BUF_SZ, fp_in) != NULL)
    fputs(buf, fp_out);

```

#### 14.3.4 The `ungetc` Function

If a character read from an input stream is not appropriate for the current operation, we can push it back to the stream using the **`ungetc` function**. A subsequent character read operation (such as `fgetc` or `getc`) on that stream will return this character. The prototype of the `ungetc` function is given below.

```
int ungetc( int c, FILE *fp );
```

This function converts the character specified by argument `c` to `unsigned char` before pushing it to stream `fp`. If the operation is successful, the function returns the character pushed to the stream; otherwise, it returns an `EOF`.

The `ungetc` function is useful in input scanning operations. Consider that we wish to read an integer number from a stream one character at a time (using `fgetc` or `getc`). Let us also assume that a number is followed by either a whitespace or a non-numeric character. For example, each line given below contains two integer numbers: 123 and 1234.

123 2345  
123, 2345

Note that we have no information about the number of digits in the number being read. Only when we read a non-numeric character, will we come to know that the digits in the number are over. However, we have already read the non-numeric character that follows the number being read. This character may be a part of subsequent data item. Hence, we push it back to the stream and it will be read by a subsequent read operation on that stream.

The following points may be noted about the `ungetc` function:

1. C language guarantees the pushback of only one character. If we perform more than one `ungetc` operation on a stream without an intervening read or file positioning operation on that stream (such as `fseek`, `fsetpos` or `rewind`), the `ungetc` function may fail.
2. The character pushed to a stream is actually stored in the buffer associated with the stream. This character is discarded by file positioning operations.

## 14.4 Formatted I/O

The formatted I/O category provides functions for input and output of formatted data. These include the `fscanf`, `fprintf` and `vprintf` functions.

The **`fscanf` function** behaves the same as the `scanf` function, except that it reads the data from a specified input stream `fp`, instead of the keyboard. Its format is given below.

```
int fscanf(FILE *fp, const char *format, arg1, arg2, ... );
```

Similarly, the `fprintf` function behaves the same as the `printf` function, except that it prints the data to a specified stream `fp`. Its format is given below.

```
int fprintf(FILE *fp, const char *format, expr1, expr2, ... );
```

### Program 14.6 Text file containing prime numbers in a given range

The program given below generates prime numbers in a given range and appends them to `primes.dat` file. Then it displays the `primes.dat` file.

```
/* append prime numbers in a given range to primes.dat file and
   then display the file */
#include <stdio.h>
#include "fileopen.c"

int is_prime(int n);

int main()
```

```

{
    char fname[] = "primes.dat";
    FILE *fp;
    int m, n, i;

    /* add prime numbers in a given range to primes.dat file */
    printf("Enter range : ");
    scanf("%d %d", &m, &n);
    fp = fopen(fname, "at+", "");
    for(i = m; i <= n; i++) {
        if (is_prime(i))
            fprintf(fp, "%d ", i);
    }
    fclose(fp);

    /* display contents of primes.dat file */
    fp = fopen(fname, "rt");
    printf("Prime numbers in primes.dat file:\n");
    while(fscanf(fp, "%d", &i) != EOF)
        printf("%d ", i);

    fclose(fp);
    return 0;
}

/* test if n is a prime number */
int is_prime(int n)
{
    int d;
    for (d = 2; d < n; d++) {
        if (n % d == 0)
            break;
    }

    return (d == n) ? 1 : 0;
}

```

The program accepts a range from the keyboard, opens `primes.dat` as a text file in append mode, writes prime numbers in the specified range to it using a `for` loop and then closes the file. Observe the use of the `fprintf` function to write the prime numbers.

The `primes.dat` file is then opened again in read mode and a `while` loop is used to read and display its contents. Observe the use of the `fscanf` function to read the numbers from the file.

The output obtained when the program is executed for the first time is given below.

```
Enter range : 1 20
Prime numbers in primes.dat file:
2 3 5 7 11 13 17 19
```

The program is executed again and the output obtained is given below.

```
Enter range : 21 40
Prime numbers in primes.dat file:
2 3 5 7 11 13 17 19 23 29 31 37
```

Although the program works as expected, note that `primes.dat` file is opened and closed twice in each program execution. This overhead can be eliminated by opening the file in update mode ("wt+"); this allows both read and write operations on a file. The modified program is given below. Observe the use of `rewind` function to reposition the file pointer to the beginning of the file after the prime numbers are written to it (only the `main` function is given to save space).

```
int main()
{
    char fname[] = "primes2.dat";
    FILE *fp;
    int m, n, i;

    /* add prime numbers in a given range to primes.dat file */
    printf("Enter range : ");
    scanf("%d %d", &m, &n);
    fp = fopen(fname, "at+", "");
    for(i = m; i <= n; i++) {
        if (is_prime(i))
            fprintf(fp, "%d ", i);
    }

    rewind(fp);
    printf("Prime numbers in primes.dat file:\n");
    while(fscanf(fp, "%d", &i) != EOF)
        printf("%d ", i);

    fclose(fp);
    return 0;
}
```

---

Write a program to accept from keyboard the information of several students appearing for the HSC examination (examination seat number, name and marks in six subjects) and store it in a data file named **marks.dat**.

**Solution:** Let us define a structure **student** to store the HSC marks data of a student as follows:

```
struct student {  
    int seat_no;  
    char name[60];  
    int marks[6];  
};
```

We also define two functions: **stud\_read\_data** to read the examination data of a student from the keyboard and **stud\_print\_data** to print it to a text file. Their prototypes are given below.

```
int stud_read_data(struct student *stud);  
void stud_print_data(FILE *fp, const struct student *stud);
```

Both functions have a parameter **stud** which is a pointer to **struct student**. This helps us avoid copying the entire **Student** structure, which is quite large in size, when these functions are called.

The **stud\_read\_data** function uses parameter **stud** to return to the calling function the data of a student entered from the keyboard. It also returns an integer as a function value to indicate the availability of data. As examination seat numbers are positive integer values, a zero or negative value can be entered as the seat number to indicate the end of data entry, in which case, the function returns 0. Otherwise, it returns 1 after reading the remaining data of a student.

To read the data of several students from the keyboard and write it to an output file associated with file pointer **fp**, we use a **while** loop as shown below.

```
while (stud_read_data(&stud))  
    stud_print_data(fp, &stud);
```

The complete program is given below in which the **main** function first opens **marks.dat** file in append mode using the **fileopen** function defined in Example 14.2. The append mode allows data to be added to an existing file. Then the student data is accepted from the keyboard using the **stud\_read\_data** function as long as it returns *true*, and it is written to the **marks.dat** file using the **stud\_print\_data** function. Finally the file is closed.

```
/* create HSC examination data for several students */  
#include <stdio.h>  
#include "fileopen.c"  
  
struct student {  
    int seat_no;
```

```
char name[60];
int marks[6];
};

int stud_read_data(struct student *stud);
void stud_print_data(FILE *fp, const struct student *stud);

int main()
{
    struct student stud;
    FILE *fp;

    fp = fileopen("marks.dat", "a", "");
    /* open MARKS.DAT in append mode */
    /* read student data and store it in file */
    while (stud_read_data(&stud))
        stud_print_data(fp, &stud);

    fclose(fp);
    return 0;
}

/* read HSC examination data of a student from keyboard */
int stud_read_data(struct student *stud)
{
    int i;
    printf("\nEnter HSC examination data for a student:\n");
    printf("Seat number: ");
    scanf("%d", &stud->seat_no);
    if (stud->seat_no <= 0)
        return 0; /* indicates end of data entry */

    printf("Student name: ");
    fflush(stdin);
    gets(stud->name);
    printf("Marks in six subjects: ");
    for (i = 0; i < 6; i++)
        scanf("%d", &stud->marks[i]);
    return 1;
}

/* print HSC examination data of student to a text file */
void stud_print_data(FILE *fp, const struct student *stud)
{
```

```
int i;
fprintf(fp, "%d %s ", stud->seat_no, stud->name);
for(i = 0; i < 6; i++)
    fprintf(fp, "%d ", stud->marks[i]);
fprintf(fp, "\n");
}
```

The program is executed and the data of one student is first entered as shown below.

```
Enter HSC examination data for a student:
```

```
Seat number: 1
```

```
Student name: Dineshkumar Mittal
```

```
Marks in six subjects: 67 45 85 76 65 55
```

```
Enter HSC examination data for a student:
```

```
Seat number: 0
```

The program is executed again and the data of four more students is added. The contents of the **marks.dat** file are as shown below.

```
1 Dineshkumar Mittal 67 45 85 76 65 55
2 R. Mahesh 67 65 54 45 43 34
3 Prathamesh K. Sonawane 80 87 89 90 98 99
4 Ms. Swati S. Patankar 37 45 56 67 87 64
5 Mr. Mohammad J. A. Qureshi 46 54 43 39 49 47
```

Observe the variations in students' names that have 2–5 components including the salutation. This level of complexity is quite common in real-life applications. In the next program, we discuss how such a file can be easily read using the **fscanf** function.

#### Program 14.8 Process H.S.C. examination marks

Write a program to read the contents of a HSC marks data file **marks.dat** created in Program 14.7 and process it to determine the examination result (the file includes total marks, percentage marks, result (pass/fail) and class obtained for each student) and store it in **result.dat** file.

**Solution:** A structure **student** used to store the data and result of a student is given below. It has four fields, namely, **tot\_marks**, **perce**, **result** and **clas** in addition to those used in Program 14.7.

```
struct student {
    int seat_no;
    char name[60];
```

```
int marks[6];
int tot_marks;
float perce;
char result;
int clas;
};
```

To perform file I/O, two functions are defined, whose prototypes are given below.

```
int stud_read_data(FILE *fp, struct student *stud);
void stud_print_res(FILE *fp, const struct student *stud);
```

These functions are similar to those used in Program 14.7. The `stud_read_data` function returns 0 if end-of-file is reached in the input file; otherwise, it reads the data of a student and provides it to the calling function through a pointer to `struct student`. The `stud_print_res` function writes the processed data of a student to an output file.

The program defines four more functions to process student data and determine the examination result. Their prototypes are given below.

```
void stud_process_data(struct student *stud);
char hsc_result(int marks[]);
int hsc_totmarks(int marks[]);
int hsc_class(char result, float perce);
```

The `stud_process_data` function calls `hsc_result`, `hsc_totmarks` and `hsc_class` functions to determine the examination result of a student. Although these functions could have accepted a parameter of type `struct student *`, they are designed to be more useful in other situations as well. For example, the `hsc_result` and `hsc_totmarks` functions accept an integer array as a parameter.

The `stud_read_data`, `stud_process_data` and `stud_print_res` functions are used in the `main` function to process all records in the `marks.dat` file and write the result to the `result.dat` file as well as the screen as shown below.

```
while (stud_read_data(fp_marks, &stud)) {
    stud_process_data(&stud);
    stud_print_res(stdout, &stud);
    stud_print_res(fp_res, &stud);
}
```

The complete program is given below.

```
/* process HSC examination data from marks.dat and prepare result fil
#include <stdio.h>
#include <string.h>
```

```
#include "fileopen.c"

struct student {
    int seat_no;
    char name[60];
    int marks[6];
    int tot_marks;
    float perce;
    char result;
    int clas;
};

int stud_read_data(FILE *fp, struct student *stud);
void stud_print_res(FILE *fp, const struct student *stud);
void stud_process_data(struct student *stud);
char hsc_result(int marks[]);
int hsc_totmarks(int marks[]);
int hsc_class(char result, float perce);

int main()
{
    struct student stud;
    FILE *fp_marks, *fp_res;

    /* open data files */
    fp_marks = fileopen("marks.dat", "rt", "");
    fp_res = fileopen("result.dat", "wt", "");

    /* read data from marks.dat, process it & store result in result.d
    while (stud_read_data(fp_marks, &stud)) {
        stud_process_data(&stud);
        stud_print_res(stdout, &stud);
        stud_print_res(fp_res, &stud);
    }

    /* close files */
    fclose(fp_marks);
    fclose(fp_res);
    return 0;
}

/* read HSC examination data of a student from a text data file */
int stud_read_data(FILE *fp, struct student *stud)
```

```

{
    int i;
    if (fscanf(fp, "%d", &stud->seat_no) == EOF)
        return 0;
    fscanf(fp, "%[^\0123456789]s", stud->name);
    for (i = 0; i < 6; i++)
        fscanf(fp, "%d", &stud->marks[i]);
    return 1;
}

/* print HSC examination result of student to a text data file */
void stud_print_res(FILE *fp, const struct student *stud)
{
    static char *class_str[] = {"---", "Third", "Second", "First", "Di-
int i;

fprintf(fp, "%2d %-30s ", stud->seat_no, stud->name);
for(i = 0; i < 6; i++)
    fprintf(fp, "%2d ", stud->marks[i]);
fprintf(fp, "%3d %5.2f ", stud->tot_marks, stud->perce);
fprintf(fp, "%4s ", stud->result == 'P' ? "Pass" : "Fail");
fprintf(fp, "%-12s\n", class_str[stud->clas]);
}

/* process data of a student to determine examination result */
void stud_process_data(struct student *stud)
{
    int i;
    stud->result = hsc_result(stud->marks);
    if (stud->result == 'P') {
        stud->tot_marks = hsc_totmarks(stud->marks);
        stud->perce = stud->tot_marks / 6.0;
    }
    else stud->tot_marks = stud->perce = 0;

    stud->clas = hsc_class(stud->result, stud->perce);
}

/* determine result (Pass/Fail) of a student */
char hsc_result(int marks[])
{
    int i;
    for(i = 0; i < 6; i++) {

```

```

        if (marks[i] < 35)
            return 'F'; /* Fail */
    }
    return 'P';      /* Pass */
}

/* determine total marks of a student */
int hsc_totmarks(int marks[])
{
    int i, tot = 0;
    for(i = 0; i < 6; i++)
        tot += marks[i];
    return tot;
}

/* determine class obtained by a student */
int hsc_class(char result, float perce)
{
    int i;
    if (result == 'P') {
        if (perce >= 75.0)
            return 4;           /* Dist */
        else if (perce >= 60.0)
            return 3;           /* First */
        else if (perce >= 50.0)
            return 2;           /* Second */
        else return 1;          /* Third */
    }
    else return 0;             /* NO class */
}

```

Observe that the name of a student is read using the `fscanf` statement given below.

```
fscanf(fp, "%[^0123456789]s", stud->name);
```

The format string `%[^0123456789]s` reads a string having any character other than digits (which actually mark the next input field in the record). Thus, student names in diverse formats are easily handled by the `fscanf` function.

The program is executed and the output written to the `result.dat` file is given below.

1 Dineshkumar Mittal	67 45 85 76 65 55 393 65.50	Pass	Firs
2 R. Mahesh	67 65 54 45 43 34 0 0.00	Fail	---
3 Prathamesh K. Sonawane	80 87 89 90 98 99 543 90.50	Pass	Dist

4 Ms. Swati S. Patankar	37 45 56 67 87 64 356 59.33 Pass Seco
5 Mr. Mohammad J. A. Qureshi	46 54 43 39 49 47 278 46.33 Pass Thir

Also observe that if the result of a student is *Fail*, his total marks and percentage marks are printed as 0. Instead, we could have written "—" or blank spaces. However, this will pose problems in reading this result file subsequently.

#### Program 14.9 Search a record in a sequential file

Write a program to search the `marks.dat` file created in Program 14.8 and display the result of the desired student.

**Solution:** In Program 14.8, we have studied how to process the HSC marks data file and prepare a result file. In this program, we search this result file to display the result of a desired student. Since the records in this file are of varying length, we cannot directly determine the position of desired record. Thus, the file must be read sequentially starting from the beginning.

We use `stud_read_res` function to read the result of a student from result file into structure `student` which is the same as that used in Program 14.8. Note that the structure members `result` and `clas` are of type `char` and `int`, respectively, whereas the corresponding data in the result file is of type `char` strings.

Another function named `stud_search_res` is used to search the result file for the record of a particular student using examination seat number as the search key. This function uses a linear search in which each record is read (using `stud_read_res` function) and examined one by one using a `while` loop. The function returns 1 if the desired record is found, and 0 otherwise. The record that is found is available to the calling function through the function parameter of type `struct student *`.

The `main` function first opens the `result.dat` file as a text file in read mode. It then reads the seat number of the student to be looked for from the keyboard and calls `stud_search_res` function to search the result for this student. If the desired record is found, it is displayed using `stud_print_res` function; otherwise, an appropriate error message is displayed.

```
/* Search HSC examination result file */
#include <stdio.h>
#include <string.h>
#include "fileopen.c"

struct student {
    int seat_no;
    char name[60];
    int marks[6];
    int tot_marks;
    float perce;
```

```
char result;
int clas;
};

int stud_search_res(FILE *fp, int seat_no, struct student *stud);
int stud_read_res(FILE *fp, struct student *stud);
void stud_print_res(FILE *fp, const struct student *stud);

int main()
{
    struct student stud;
    FILE *fp_res;
    int seat_no;

    fp_res = fileopen("result.dat", "rt", ""); /* open result file */

    printf("Enter seat number of student to search: ");
    scanf("%d", &seat_no);

    /* search result file and display search result */
    if (stud_search_res(fp_res, seat_no, &stud))
        stud_print_res(stdout, &stud);
    else printf("No student found with seat number %d", seat_no);

    fclose(fp_res);
    return 0;
}

/* search for a specified student in result file */
int stud_search_res(FILE *fp, int seat_no, struct student *stud)
{
    if (stud_read_res(fp, stud) == 0)
        return 0;
    while (seat_no != stud->seat_no) {
        if (stud_read_res(fp, stud) == 0)
            return 0;
    }
    return 1;
}

/* read HSC examination data of a student from result file */
int stud_read_res(FILE *fp, struct student *stud)
{
```

```

int i;
char result[5], clas[15];
static char *class_str[] = {"---", "Third", "Second", "First", "Di
if (fscanf(fp, "%d", &stud->seat_no) == EOF)
    return 0;

fscanf(fp, "%[^0123456789]s", stud->name);
for (i = 0; i < 6; i++)
    fscanf(fp, "%d", &stud->marks[i]);

fscanf(fp, "%d", &stud->tot_marks);
fscanf(fp, "%f", &stud->perce);
fscanf(fp, "%s", &result);
stud->result = result[0];
fscanf(fp, "%s", &clas);

for (i = 0; strcmp(clas, class_str[i]) != 0; i++)
    ;
stud->clas = i;
return 1;
}

/* include stud_print_res function from Program 14.8 */

```

The output of the program obtained by executing it twice is given below.

```

Enter seat number of student to search: 3
3 Prathamesh K. Sonawane      80 87 89 90 98 99 543 90.50 Pass Dist

```

```

Enter seat number of student to search: 10
No student found with seat number 10

```

Note that the functions from Program 14.7 will work correctly with the modified structure declaration from Programs 14.8 and 14.9, even in the presence of extra structure members.

Also note that the functions in this program can be used in Program 14.8 as both programs use the same declaration of structure **student**. This has, however, led to some complications in function **stud\_read\_res** while reading the data for **result** and **clas** members. We can simplify this function by defining another structure, say **student\_res**, with members **result** and **clas** as character arrays.

Also note that the functions in this program can be used in Program 14.8 as both programs use the same declaration of structure **student**. This has, however, led to some complications in function

`stud_read_res` while reading the data for `result` and `clas` members. We can simplify this function by defining another structure, say `student_res`, with members `result` and `clas` as character arrays.

## 14.5 Advanced Concepts

### 14.5.1 FILE Structure

The **FILE structure** is declared in the `<stdio.h>` header file as follows:

```
typedef struct {
    short           level;          /* fill/empty level of buffer */
    unsigned        flags;          /* File status flags */
    char            fd;             /* File descriptor */
    unsigned char   hold;           /* Ungetc char if no buffer */
    short           bsize;          /* Buffer size */
    unsigned char   *buffer;        /* Data transfer buffer */
    unsigned char   *curp;          /* Current active pointer */
    unsigned        istemp;         /* Temporary file indicator */
    short           token;          /* Used for validity checking */
} FILE;
```

/\* This is the FILE object \*/

This structure contains information for controlling the stream that includes the file position indicator, end-of-file indicator and pointer to the associated buffer.

### 14.5.2 Pagination Control in Display of Text Files

In Program 14.1, we studied how to display a text file on the screen. Although it correctly displays text files, it has severe limitations. First, when we display large text files, usually containing more than 25 lines, the screen scrolls and we are unable to read the contents at the beginning of the file. This problem can be overcome by providing pagination control, in which we pause the output whenever a full screen of text is displayed.

The text file may also have one or more lines with more characters than the output line length, which is 80 by default in a Windows command shell (as well as Turbo C/C++). The program should be able to handle such situations and display the output correctly. Further, the program should also be able to handle user-specified tab size. In this section, we discuss how such capabilities can be included in the file display program.

Let us write a function named `fileshow` to display a specified text file. Using the simple file display code given in Program 14.1 and the `fileopen` function from Example 14.1b, this function can be written as shown below.

```

void fileshow(char *fname)
{
    FILE *fp;
    int ch, nline;

    fp = fileopen(fname, "rt", "");

    /* display contents of file fname */
    while ((ch = getc(fp))!= EOF)
        printf("%c", ch);
    fclose(fp);
}

```

Now let us add pagination control to this function. A text screen usually has 25 rows, each containing 80 characters. When we print a *printing* character to the screen, the cursor automatically moves to the next character position. Similarly, if we print a character in the last column of a line, the cursor usually moves to the first column in the next line. Thus, we can print only 79 characters on a screen line without causing the cursor to move to the next line.

Let us first assume a simple situation in which each line in the given text file has at most 79 characters. We can now determine the number of lines displayed on the screen by counting the newline characters in the input stream. Thus, to add pagination control to the function, the `while` loop should be modified as shown below (and line counter `nline` should be declared).

```

/* display contents of file one screen at a time */
nline = 0;
while ((ch = getc(fp)) != EOF)  {
    printf("%c", ch);

    if (ch == '\n')  {
        nline++;
        if (nline == 24)  {
            printf("Press any key to continue...");
            ch = getch();
            printf("\n"); /* use clrscr() if available */
            nline = 0;
        }
    }
}

```

First, `nline` is initialized to 0. Then a `while` loop reads the characters in the specified file, one at a time. Within this loop, the character is first displayed. Next, if it is a newline character, the line counter `nline` is incremented and if it equals 24, a message *Press any key to continue ...* is displayed. The program then waits for a key press after which, a newline is printed and the `nline` counter is reset to 0

to continue the display of the next full screen of information.

Now let us consider a more realistic situation in which the text file may contain longer lines with 80 or more characters. Note that now we cannot rely solely on newline characters in the text file to count the number of lines displayed on the screen. The modified code segment that uses a column counter `ncol` in conjunction with the row counter `nrow` is given below.

```
/* display contents of file one screen at a time */
nline = ncol = 0;
while ((ch = getc(fp)) != EOF) {
    printf("%c", ch);

    ++ncol;
    if (ch == '\n' || ncol == 80) {
        ncol = 0;
        if (++nline == 24) {
            printf("Press any key to continue...");
            ch = getch();
            printf("\n"); /* use clrscr() if available */
            nline = 0;
        }
    }
}
```

Initially, the counters `nline` and `ncol` are initialized to 0 and a `while` loop is set up as before. After a character read from the file is displayed on the screen, the `ncol` counter is incremented and an `if` statement is used to test whether we have reached the end of the current line (if `ch` is a newline character or `ncol` equals 80). If yes, the column counter is reset to 0 and the line counter is incremented. If the line counter equals 24, the program waits for a key press, after which the screen is cleared and `nline` is reset to 0 to start displaying the next full screen of information.

However, this code is still not complete. We have not yet considered one subtle yet important point, the presence of tab characters in the input file. A tab character causes the cursor to move to the next tab position which occurs at every eight positions. Hence, the statement to increment the column count in the above program segment should be replaced by the following code:

```
if (ch == '\t')
    ncol = (ncol / 8 + 1) * 8; /* move to next tab stop */
else ncol++;
```

The complete function `fileshow2` that displays a file containing longer lines as well as tabs with pagination control is given below.

```
/* fileshow: function to display a file with pagination control.
Allows display of files having longer line (80 or more chars) and tab
```

```

/*
void fileshow2(char *fname)
{
    FILE *fp;
    int ch, nline, ncol;

    fp = fileopen(fname, "r", "");

    /* display contents of file one screen at a time */
    nline = ncol = 0;
    while ((ch = getc(fp)) != EOF) {
        printf("%c", ch);

        if (ch == '\t')
            ncol = (ncol / 8 + 1) * 8; /* move to next tab stop */
        else ncol++;
        if (ch == '\n' || ncol == 80) {
            ncol = 0;
            if (++nline == 24) {
                printf("Press any key to continue...");
                ch = getch();
                printf("\n"); /* use clrscr() if available */
                nline = 0;
            }
        }
    }
    fclose(fp);
}

```

As a final improvement to the file display function, let us accept tab size as an input parameter and print the file using this tab size. Note that if the input character is a tab, we cannot directly print it on the screen because the default tab stops occur after every 8 character positions. Hence, we need to print a tab using the appropriate number of spaces. However, the number of spaces to be printed to move the cursor to the next tab stop depends on the current cursor position. Hence, the code to process the tab character is slightly complex.

We first determine, in variable `new_col`, the cursor position after the tab character is printed. If this position is outside the screen (`new_col > 80`), we assign value 80 to `new_col`. Then we print space character from the current cursor position up to (but not including) the `new_col` position. Finally, the value of `ncol` is updated. The modified function is given below.

```

/* fileshow3: function to display a text file.
   Supports pagination, longer lines and tabs and user specified tab

```

```

void fileshow3(char *fname, int tabsz)
{
    FILE *fp;
    int ch, nline, ncol;

    fp = fileopen(fname, "r", "");

    /* display contents of file one screen at a time */
    nline = ncol = 0;
    while ((ch = getc(fp)) != EOF) {
        if (ch == '\t') {
            int new_ncol, i;
            /* print tab character as spaces and update the ncol counter
             * first determine column corresponding to next tab stop */
            new_ncol = (ncol / tabsz + 1) * tabsz;
            if (new_ncol > 80)
                new_ncol = 80;
            /* now print spaces up to new_ncol column */
            for (i = ncol; i < new_ncol; i++)
                printf(" ");
            ncol = new_ncol; /* update ncol */
        }
        else {
            printf("%c", ch);
            ++ncol;
        }

        if (ch == '\n' || ncol == 80) {
            ncol = 0;
            if (++nline == 24) {
                printf("Press any key to continue...");
                ch = getch();
                printf("\n"); /* use clrscr() if available */
                nline = 0;
            }
        }
    }

    fclose(fp);
}

```

#### 14.5.3 Direct Input/Output

The direct I/O functions allow us to read or write internal representation of data items from/to a disk file. Thus, the files created using the functions in this category are binary files. The C language provides two functions in this category: **fread** and **fwrite**.

### The **fwrite** Function

The **fwrite** function is used to write a specified number of data items of given size to an output stream. Its prototype is given below.

```
size_t fwrite( const void*s, size_t sz, size_t n, FILE *fp );
```

The function writes  $n$  elements, each of size  $sz$ , from a memory location pointed to by pointer  $s$  to the output stream  $fp$ . However, if an error occurs on  $fp$ , the actual number of items written to the output stream may be less than  $n$ . The function returns the numbers of data items actually written to the output stream.

Note that parameter  $s$  is declared as a **void** (typeless) pointer, allowing items of any type to be written to the output stream. This is known as *polymorphic* behavior.

### The **fread** Function

The **fread** function reads a specified number of data items of given size from an input stream. Its prototype is given below.

```
size_t fread(void *s, size_t sz, size_t n, FILE *fp );
```

This function reads  $n$  data items, each of size  $sz$  from the input stream  $fp$  and stores them in a memory location pointed to by pointer  $s$ . However, if an end-of-file is encountered or an error occurs during the read operation, the function may read less than  $n$  data items. The actual number of items read from the input file is returned by the **fread** function.

Note that as with the **fwrite** function, parameter  $s$  is a **void** pointer allowing data to be read into an array of any type.

#### Program 14.10 Create a binary file books.dat to store data about several books

Write a program to read the following information about books from the keyboard: accession number, title, author (only one author), publication year and price. Store this information in a binary data file **books.dat** using the **fwrite** function.

**Solution:** The program given below defines a structure named **book** to store data about a book. It also uses the function **book\_read** to read this data from the keyboard in a parameter of type **struct book \***. The function returns 1 as long as valid data is entered, and 0 when the user enters 0 or a negative value for the book accession number.

The `main` function first opens the `books.dat` file in mode "ab" (a binary file in append mode). It then uses a `while` loop to read the book data one at a time and write it to the file using the `fwrite` function, as long as `book_read_data` returns true. Finally, the file is closed.

```
/* Create a book database books.dat */
#include <stdio.h>
#include "fileopen.c"

struct book {
    int acc_no;
    char title[40];
    int n_author;
    char author[3][20];
    int pub_year;
    float price;
};

int book_read(struct book *b);

int main()
{
    struct book b;
    FILE *fp;

    fp = fileopen("books.dat", "ab", "");

    /* read book data and store it in file */
    while (book_read(&b))
        fwrite(&b, sizeof(struct book), 1, fp);

    fclose(fp);
    return 0;
}
/* read data of a book from keyboard */
int book_read(struct book *b)
{
    int i;

    printf("\nEnter book data:\n");
    printf("Accession number: ");
    scanf("%d", &b->acc_no);
    if (b->acc_no <= 0) /* end of data entry */
        return 0;
```

```

printf("Book name: ");
fflush(stdin);
gets(b->title);

printf("Number of authors: ");
scanf("%d", &b->n_author);
fflush(stdin);
for (i = 0; i < b->n_author; i++) {
    printf("Author: ");
    gets(b->author[i]);
}

printf("Publication year: ");
scanf("%d", &b->pub_year);
printf("Price: ");
scanf("%f", &b->price);
return 1;
}

```

The program is executed and data about several books is entered as shown below:

<u>Acc. No.</u>	<u>Book Title</u>	<u>Author(s)</u>	<u>Pub Year</u>	<u>Price</u>
1001	Programming with ANSI C	Brian Kernighan, Dennis Ritchie	1992	175
1002	Let us C	Yashavant Kanetkar	2010	300

Program execution is shown below for data entry of only one book to save space.

```

Enter book data:
Accession number: 1001
Book name: Programming with ANSI C
Number of authors: 2
Author: Brian Kernighan
Author: Dennis Ritchie
Publication year: 1992
Price: 175

```

```

Enter book data:
Accession number: -1

```

#### **Program 14.11** Display the contents of a binary file books.dat

Write a program to display the contents of the books.dat file created in Program 14.10.

**Solution:** The program to display the contents of the books.dat file created in Program 14.10 is given below. Since this is a binary file, the program opens it in "rb" mode (a binary file for read operation). The program then uses a **while** loop to read the file contents, one book at a time and display it using function **book\_print** as shown below.

```
while (fread(&b, sizeof(struct book), 1, fp))
    book_print(&b);
```

The **fread** function reads a book record (of size **sizeof(struct book)**) and stores it in variable **b** of type **struct book**. As this function returns the number of records read from the file, the **while** loop continues as long as the **fread** function returns 1. The data of each book is displayed on the screen using the **book\_print** function. The control leaves the loop when **fread** function returns 0, i. e., when there are no records left in the file. Finally the file is closed.

The **fread** function reads a book record (of size **sizeof(struct book)**) and stores it in variable **b** of type **struct book**. As this function returns the number of records read from the file, the **while** loop continues as long as the **fread** function returns 1. The data of each book is displayed on the screen using the **book\_print** function. The control leaves the loop when **fread** function returns 0, i. e., when there are no records left in the file. Finally the file is closed.

```
/* Displays the contents of books.dat file */
#include <stdio.h>
#include "fileopen.c"

struct book {
    int acc_no;
    char title[40];
    int n_author;
    char author[3][20];
    int pub_year;
    float price;
};

int book_print(const struct book *b);

int main()
{
    struct book b;
    FILE *fp;

    fp = fileopen("books.dat", "rb", "");
    /* read book data, one book at a time and display it on the screen
```

```

        while (fread(&b, sizeof(struct book), 1, fp))
            book_print(&b);

        fclose(fp);
        return 0;
    }

/* display book data */
int book_print(const struct book *b)
{
    int len, tot_len, i;

    printf("%4d %-30s", b->acc_no, b->title);

    printf("%s", b->author[0]);
    tot_len = strlen(b->author[i]);
    for (i = 1; i < b->n_author; i++) {
        printf(", %s", b->author[i]);
        tot_len += strlen(b->author[i]) + 2;
    }
    printf("%*s", 35-tot_len, "");

    printf("%4d %4.0f\n", b->pub_year, b->price);
}

```

The program output is shown below.

1001	Programming with ANSI C	Brian Kernighan, Dennis Ritchie	19
1002	Let us C	Yashavant Kanetkar	

Observe how the spacing is controlled in the `book_print` function. To accommodate each record in a single output line (80 characters assumed), the book title is printed using only 30 characters and the list of authors' names using only 35 characters. The number of characters required to print the list of authors' names is calculated in variable `tot_len`. Observe how the second-last `printf` statement uses "%\*s" format to print the remaining spaces in a field width of 35 characters.

Alternatively, we could have formed a local string containing the authors' names (say, in `char` array named `authors`) and printed it using the desired field width of 35 characters as shown below. See the comment showing the calculation of the minimum size of the `authors` array (62 characters).

```

int book_print(const struct book *b)
{
    int i;

```

```

char authors[62]; /* 19*3(authors) + 2*2 (" , " strings) + 1 null *

printf("%4d %-30s", b->acc_no, b->title);

strcpy(authors, b->author[0]);
for (i = 1; i < b->n_author; i++) {
    strcat(authors, ", ");
    strcat(authors, b->author[i]);
}
printf("%-35s", authors);

printf("%4d %4.0f\n", b->pub_year, b->price);
}

```

#### 14.5.4 File Positioning

All file I/O functions (character I/O, formatted I/O and direct I/O) perform operations on a file in a sequential manner in which the file contents are read from a file or written to it starting from beginning to end. C provides several functions to perform random file I/O in which we can position the file pointer at any desired location in the file before the read or write operation is performed on it. These functions include **fseek**, **ftell**, **rewind** and **fsetpos**.

##### The **fseek** Function

The **fseek** function is used to set the file position indicator associated with a specified stream to the desired location. Its format is given below.

```
int fseek( FILE *fp, long offset, int whence );
```

The parameter *whence* takes one of three values: **SEEK\_SET** **SEEK\_CUR** and **SEEK\_END**, specifying file beginning, current file position and end of file, respectively. The parameter *offset* specifies the offset of the desired file position in bytes from the position specified by *whence*. A positive value for *offset* specifies the position after that specified by *whence*, whereas a negative *offset* specifies a position before it. If successful, the function returns 0; otherwise, it returns a non-zero value.

##### Example 14.3 File positioning

The statement given below sets the file position indicator **fp** at an offset of 0 bytes from **SEEK\_END**, i.e., at the end of stream **fp**.

```
fseek(fp, 0, SEEK_END);
```

The statement given below sets the file position indicator **n** bytes before the end-of-file.

```
fseek(fp, -n, SEEK_END);
```

Now Consider the example given below.

```
if (fseek(fp, 100, SEEK_CUR) == 0)           /* fseek successful */
{
    /* perform operation */
    ...
}
else {
    printf("Error is positioning file pointer...\n");
    exit(1);
}
```

In this example, the `fseek` function tries to position the file position indicator 100 bytes after `SEEK_CUR`, i. e., the current file position. The value returned by this function is tested in an `if` statement and file I/O is performed if the seek operation is successful; otherwise, an error message is printed and the program ends.

### The **ftell** and **rewind** Functions

The **ftell** function returns the current value of the file position indicator associated with a specified stream. However, it returns `-1L` on failure. Its format is given below.

```
long ftell( FILE *fp );
```

The **rewind** function resets the value of file position indicator associated with a specified stream to the file beginning. Its format is given below.

```
void rewind(FILE *fp );
```

Noted that the call `rewind(fp)` is equivalent to `fseek(fp, 0, SEEK_SET)`;

### The **fgetpos** and **fsetpos** Functions

The **fgetpos** function returns the current value of file position indicator associated with a specified stream, whereas the **fsetpos** function sets the file position indicator associated with a specified stream. The prototypes of these functions are given below.

```
int fgetpos(FILE *fp, fpos_t *pos );
int fsetpos(FILE *fp, fpos_t *pos );
```

The **fgetpos** function stores the file position indicator in object pointed by argument `pos`, whereas the **fsetpos** functions sets the file position indicator value to that stored in object pointed by

argument **pos**. Note that these functions return 0 value when successful and non-zero otherwise.

## Exercises

1. Answer the following questions in brief:
  - a. How is a text file created in Linux OS different from that created in MS Windows OS?
  - b. What is a stream? State the types of streams.
  - c. Why is it necessary to use buffering in streams?
  - d. State the names of built-in streams.
  - e. State the functions for the following operations:  
read a character from input stream  
push back a character to an input stream
  - f. State the purpose of the following functions: **fclose**, **fprintf**, **fgets**
2. Write a program to read a text file and replace two or more consecutive space characters in it with a single space character.
3. Write a C program to create a data file for employee salary data.

## Exercises (Advanced Concepts)

1. Answer the following questions in brief:
  - a. State the functions for direct file I/O
  - b. State the purpose of the following functions: **fseek**, **rewind**, **fsetpos**
  - c. State the SEEK\_... constants used in **fseek** function and explain their meaning.
  - d. Explain the **fread** function.
  - e. State the advantages of direct file I/O.
  - f. Explain why a binary file containing numeric data will usually require less space compared to a text file containing same data.
2. Write a complete program to manage the data of your personal library books. The program should use binary files to create such data, update it and delete unwanted data.
3. Write a program similar to C *beautifier* program in which the input C program should be properly indented and formatted before it is displayed on the screen.

# 15 Searching and Sorting

This chapter presents searching and sorting techniques. They are particularly useful in system software and database applications. Linear search is considered first followed by binary search which is very efficient. The sorting techniques covered include bubble sort, selection sort and insertion sort.

The *Advanced Concepts* section presents recursive binary search and improvements in basic sorting techniques. In addition, the use of standard library functions for binary search (`bsearch`) and quick sort (`qsort`) is presented.

## 15.1 Searching

**Searching** is an operation in which we look for the presence of a specific value in a given array (or some other data structure). If the value is present, usually we are interested in its location. The array being searched may have multiple entries of the value being searched. In this case, we may be interested in the first, *i*th, last or all occurrences of this value.

There are two commonly used search techniques: *linear search* and *binary search*. This section presents the functions for these techniques assuming that the array being searched is of type `int`. Note that for other array types, we should suitably modify these functions.

### 15.1.1 Linear (or Sequential) Search

In **linear search**, all the elements of an array are examined one by one in a sequential manner, usually starting from the beginning. If the element being searched is found, its position is reported. One advantage in linear search is that it does not require any specific ordering of array elements, i. e., it can be performed on unsorted arrays as well.

The function `ivec_lsearch` given below performs a linear search for value `val` in an integer array `x` having `n` elements and returns the position of the first element found. However, if the element is not found, the function returns `-1`, which is not a valid array position.

```
int ivec_lsearch(int x[], int n, int val)
{
    int i;
    for (i = 0; i < n; i++) {
```

```
    if (x[i] == val)
        return i;
}
return -1;
}
```

If the element being searched for is present at position 0, this function requires only one comparison operation. On the other hand, if it is present in the last array position or is not present in the array, the search operation requires  $n$  comparisons. Thus, on an average, we require  $n/2$  comparisons to search for an element using linear search. This is time-consuming, particularly for large arrays.

To understand how this function can be used, consider the `main` function given below.

```
int main()
{
    int a[] = {2, 13, 7, 11, 19, 29, 23, 17, 5, 3};
    int num, pos;

    printf("Enter element to be searched: ");
    scanf("%d", &num);

    pos = ivec_lsearch(a, 10, num);
    if (pos >= 0)
        printf("%d found at position %d\n", num, pos);
    else printf("%d not found\n", num);

    return 0;
}
```

The program is executed twice and the output is given below.

```
Enter element to be searched: 24
24 not found
```

```
Enter element to be searched: 29
29 found at position 5
```

Note that if we wish to search an array of some different type, say `float`, we can modify the first line of the above function as shown below.

```
int fvec_lsearch(float x[], int n, float val)
```

If we need to search more than one type of array, we should write multiple searching functions, one for

each type. This also applies to other searching functions discussed in this section.

### 15.1.2 Binary Search

**Binary search** is a much faster search technique compared to linear search. However, it requires the elements of the array being searched to be sorted. In this section, we assume that the array being searched is sorted in ascending order.

Fig. 15.1 illustrates the process of binary search. In this technique, we first divide the given array into two almost equal halves by determining the position of the middle element of the array as

```
mid = (left + right) / 2;
```

where `left` and `right` are indexes of the first and last array element, respectively. The search element `val` is then compared with the array element at position `mid`. This comparison has three possible outcomes:

1. `val < x[mid]`: In this case, the right half array is discarded and the `right` pointer is positioned at the element before the `mid` position using the equation `right = mid - 1`. The search then continues in the left half of the array.
2. `val > x[mid]`: In this case, the left half array is discarded by positioning the `left` pointer at the element next to the `mid` position using the equation `left = mid + 1`. The search then continues in the right half of the array.
3. `val == x[mid]`, i. e., the element being searched is found at position `mid`. In this case, the position `mid` is returned.

These steps (determine the `mid` position of the portion of the array being searched, compare search element `val` with `a[mid]` and take appropriate action) can be coded as shown below:

0	1	2	3	4	5	6	7	8	9
2	5	11	16	21	27	33	<b>42</b>	45	51
left			mid				right		

0	1	2	3	4	5	6	7	8	9
2	5	11	16	21	27	33	<b>42</b>	45	51
left			mid				right		

**Fig. 15.1** Binary search (Search 42): (a) Value being searched is greater than element at mid position, (b) Left half array is discarded and the search continues with right half. The element being searched is found at new mid position

```

mid = (left + right) / 2;

if (val < x[mid])
    right = mid - 1;      /* discard right half */
else if (val > x[mid])
    left = mid + 1;      /* discard left half */
else return mid;          /* value found at position mid */

```

These steps are performed repeatedly as long as there are one or more elements in the portion of the array being searched, i. e., `left <= right`. If the pointers `left` and `right` cross each other, the element being searched for is not present in the array.

The `ivec_bsearch` function given below uses a `while` loop to implement binary search. Note that if the search value is found, its position will be returned from within the `while` loop. However, if it is not found, i. e., when `left > right`, control will leave the `while` loop and the function will return -1.

```

/* binary search - iterative version */
int ivec_bsearch(int x[], int n, int val)
{
    int left = 0;
    int right = n - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if (val < x[mid])
            right = mid - 1;      /* discard right half */
        else if (val > x[mid])
            left = mid + 1;      /* discard left half */
        else return mid;          /* value found at position mid */
    }

    return -1;                  /* value not found */
}

```

To call this function, use the `main` function from the previous section and modify array initialization and function call statements as shown below. Note that the array elements must be entered in ascending order.

```

int a[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}; /* must be sorted */

pos = ivec_bsearch(a, 10, num);

```

This function will not work correctly if the array being searched is sorted in descending order. In this case, we will have to reverse the conditions in the `if` statement as shown below.

```
if (val > x[mid])                                /* discard right half */
    right = mid - 1;
else if (val < x[mid])                          /* discard left half */
    left = mid + 1;
else return mid;                                /* value found at position mid */
```

### 15.1.3 Recursive Binary Search

The `ivec_rbsearch` function, a recursive implementation of binary search, is given below. It accepts four parameters: the array to be searched (`x`), the range of elements to be searched (`left` and `right`) and the value to be searched (`val`). If `left <= right`, the portion to be searched is divided into two halves by determining the `mid` position and value `val` is compared with `x[mid]`. If `val < x[mid]` or if `val > x[mid]`, `ivec_rbsearch` is recursively called (from within a `return` statement) for the left or right portion. Otherwise, the value of `mid` is returned as the search value is found at `mid`. Note that if the condition `left <= right` is false, the specified value is not found and the function returns -1.

```
int ivec_rbsearch(int x[], int left, int right, int val)
{
    if (left <= right) {
        int mid = (left + right) / 2;

        if (val < x[mid])
            return rbsearch(x, 0, mid-1, val);

        else if (val > x[mid])
            return rbsearch(x, mid+1, right, val);

        else return mid;          /* value found at position mid */
    }

    return -1;                  /* value not found */
}
```

To call this function, modify the initialization and function call statements in the `main` function from the previous section as shown below.

```
int a[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}; /* must be sorted */
pos = ivec_rbsearch(a, 0, 9, num);
```

## 15.2 Sorting

**Sorting** means arranging the given data elements in some specified order. In this section, we consider sorting operations on a list of integer numbers stored in an array. These numbers are to be sorted in either ascending or descending order of their magnitudes.

The sorting operation can also be performed on arrays containing other types of data. For example, we can sort an array containing alphanumeric data (strings) in lexicographical order. This sort operation can be performed by considering uppercase and lowercase letters as either equivalent or distinct characters.

There are a large number of sorting techniques that include bubble sort, selection sort, insertion sort, shell sort, quick sort, merge sort, heap sort, radix sort, etc. This section covers the first three techniques.

Assume that the array is to be sorted in ascending order and it is initialized as follows:

```
int a[6] = {55, 85, 25, 40, 35, 20};
```

Note that if we wish to sort an array of different type, we should suitably modify the functions given in this section. Also, if a program requires different types of arrays to be sorted, we should write a sorting function for each type.

### 15.2.1 Bubble Sort

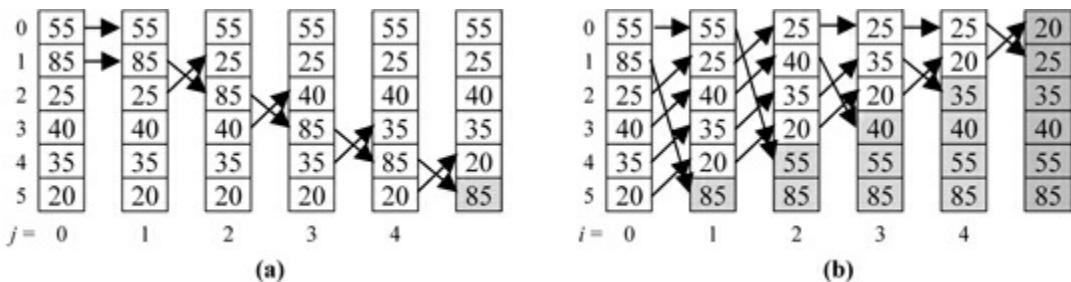
**Bubble sort** is the simplest and most well-known algorithm for sorting. In this method, the adjacent array elements are compared (starting from the element at position 0) and are swapped (i. e., exchanged) if they are not in the correct order. This process is called a **pass**. Usually, we require  $n - 1$  passes to sort an array having  $n$  elements.

Fig. 15.2a illustrates one pass (through the data) of bubble sort. The value of  $j$  written below the array indicates that the elements  $a_j$  and  $a_{j+1}$  are being compared. If the elements are out of order, they are exchanged as indicated by cross arrows. Note that for an array having  $n$  elements, this comparison (and exchange if required) operation should be performed  $n - 1$  times for values of  $j$  from 0 to  $n - 2$ . The algorithm to perform this first pass is given below.

```
for (j = 0; j < n-1; j++) {
    if (a[j] > a[j+1])
        swap a[j] and a[j+1]
}
```

Observe that after the first pass is over, the largest element in the list (85) is correctly positioned at the end of the array (as shown in grey). Each pass will thus position one element at the correct position except the last pass in which two elements will be correctly positioned. Hence, we require  $n - 1$  passes to sort a list containing  $n$  elements. Fig. 15.2b shows five passes required to sort a given list of six elements in which the sorted elements are shown in grey. Observe that the elements with lower weights move slowly upwards, similar to bubbles in water. Hence the name bubble sort.

In each pass, we can perform  $n - 1$  comparisons. However, observe that at the beginning of the  $i$ th pass, the sorted portion of the array contains  $i$  elements. Thus, the unsorted list contains only  $n - i$  elements. Hence, for the  $i$ th pass, we require only  $n - i - 1$  or  $n - 1 - i$  comparisons. The function `bubble_sort` to sort an array containing  $n$  elements is given below.



**Fig. 15.2** Bubble sort ( $n=6$ ): (a) First pass, (b) All ( $n - 1$ ) passes

```
void ivec_bubble_sort(int a[], int n)
{
    int i, j;

    for (i = 0; i < n-1; i++)          /* n-1 passes */
        for (j = 0; j < n-1-i; j++)  /* n-1-i comparisons */
            if (a[j] > a[j+1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
}
```

The `main` function that uses the above function to sort a given array is given below.

```
int main()
{
    int arr[10] = {55, 85, 25, 40, 35, 20};

    ivec_bubble_sort(arr, 6);

    printf("Sorted Array: ");
    ivec_print(arr, 6);

    return 0;
}
```

Note that as the arrays are passed by reference, sorting array `a` in `ivec_bubble_sort` actually sorts the argument array `arr` specified in a call to this function. The output of this function is given below:

```
Sorted Array: 20 25 35 40 55 85
```

We can define the `iswap` function to exchange the values of two integers as shown below.

```
void iswap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

We can now use this function to simplify the `ivec_bubble_sort` function as shown below.

```
void ivec_bubble_sort(int a[], int n)
{
    int i, j;

    for (i = 0; i < n-1; i++)          /* n-1 passes */
        for (j = 0; j < n-1-i; j++) { /* n-1-i comparisons */
            if (a[j] > a[j+1])
                iswap(&a[j], &a[j+1]);
        }
}
```

Note that if we wish to sort an array of different type, say `char`, we should add a function to swap two characters (say `cswap`) and modify the first and last lines of the function as shown below.

```
void cvec_bubble_sort(char a[], int n)

cswap(&a[j], &a[j+1]);
```

Finally note that to sort the array in descending order, we should modify the condition in the `if` statement as `a[j] < a[j+i]`.

### 15.2.2 Selection Sort

In **selection sort**, the sorted list is constructed from one end of an array by repeatedly selecting the minimum or maximum number from a given unsorted list and positioning it at one end of the sorted list. In this section, we consider sorting a given list in ascending order. If the minimum number is selected from the unsorted list, the sorted list is constructed forward from the beginning of the array.

However, if the maximum number is selected, the sorted list is constructed backwards from the end of the array.

Like bubble sort, selection sort also comprises of passes. In each pass, one element is added at the beginning (or end) of the sorted list, except the last pass in which two elements are added. Hence, we require a total of  $n - 1$  passes.

In this section, we present a simple (but inefficient) version of selection sort. The sorted list is constructed from the beginning of the array by adding to it the minimum number from the remaining array, as shown in Fig. 15.3. The  $n - 1$  passes are implemented using a `for` loop with values of loop variable  $i$  from 0 to  $n - 2$ . The inner `for` loop, with values of the loop variable  $j$  from  $i+1$  to  $n-1$ , sets up comparisons of element  $a[i]$  with the remaining array elements ( $a[i+1]$  to  $a[n-i]$ ) and exchanges them if they are not in order. Fig. 15.3a shows the first pass ( $i = 0$ ), whereas Fig. 15.3b shows all the passes. The elements in the sorted list are shown in grey. The function given below implements this approach.

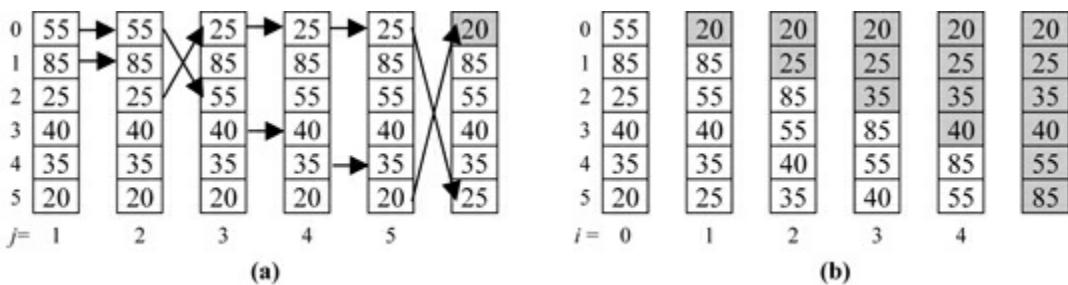


Fig. 15.3 Simple but inefficient selection sort: (a) First pass ( $i = 0$ ), (b) All passes

```
void ivec_selection_sort(int a[], int n)
{
    int i, j;

    for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++) {
            if (a[i] > a[j])
                iswap(&a[i], &a[j]);
        }
}
```

### 15.2.3 Insertion Sort

**Insertion sort** divides the given elements into two sub-lists: sorted and unsorted. It selects elements from the unsorted list one at a time and inserts them in the sorted list at the appropriate position. Fig. 15.4 illustrates such a sort in which the given array is sorted in ascending order.

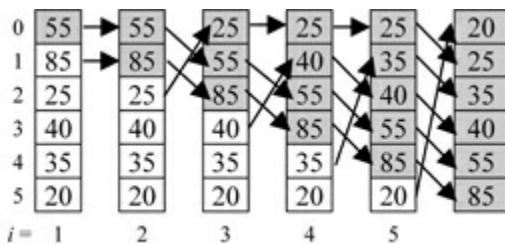
The sorted list is usually maintained at the beginning of the given array. Initially, it comprises

element  $a[0]$ , as a single element is always sorted. Like bubble and selection sort, insertion sort also requires several passes. In each pass, one element (usually the first) from the unsorted list is inserted at the appropriate position in the sorted list. Thus, we require  $n - 1$  passes. These passes are set up using a **for** loop with the values of loop variable  $i$  from 1 to  $n - 1$ . Observe that in the  $i$ th pass, the sorted list comprises elements from position 0 to  $i - 1$  and the element at position  $i$  is selected for insertion.

To insert an element at the correct position in the sorted list, let us use a simple (but inefficient) approach that involves repetitive exchange operations implemented using an inner **for** loop as shown below.

```
for (j = i; j > 0 && a[j-1] > a[j]; j--)
    iswap(&a[j-1], &a[j]);
```

The loop variable  $j$  assumes decreasing values starting from  $i$  (i. e., from the element to be inserted). In each iteration of this loop, the element  $a[j]$  is exchanged with the previous element,  $a[j - 1]$ . Thus, as this loop progresses, the selected element moves up in the sorted list. The inner loop is executed as long as  $j > 0$  (i. e., there exists an element at position  $j - 1$  with which element  $a[j]$  is to be exchanged) and  $a[j - 1] > a[j]$  (i. e., the previous element in the sorted list ( $a[j - 1]$ ) is greater than the element being inserted,  $a[j]$ ). The function using this approach is given below.



**Fig. 15.4 Insertion sort (The shaded portion is the sorted list)**

```
void ivec_insertion_sort(int a[], int n)
{
    int i, j;

    for (i = 1; i < n; i++)
        for (j = i; j > 0 && a[j-1] > a[j]; j--)
            iswap(&a[j-1], &a[j]);
}
```

### 15.3 Advanced Concepts

This section presents several advanced concepts that include alternative/efficient implementations for

sorting techniques presented in the previous section, sorting arrays of strings and standard library functions for searching and sorting (`bsearch` and `qsort`).

### 15.3.1 Alternative/Efficient Implementations for Sorting Functions

#### Bubble Sort: Alternative Implementations

While studying bubble (and other) sorting techniques, it is possible that you might come across slightly different implementations which are likely to confuse you. Hence, let us understand how bubble sort can be implemented in different ways.

Observe that in `ivec_bubbie_sort` function given in Section 15.2.1, the value of the outer loop variable `i` refers to the number of elements sorted thus far. If we use variable `i` to refer to the number of unsorted elements in the list, we can rewrite the code as shown below.

```
void ivec_bubble_sort(int a[], int n)
{
    int i, j;

    for (i = n; i > 1; i--)          /* n-1 passes */
        for (j = 0; j < i-1; j++) { /* i-1 comparisons */
            if (a[j] > a[j+1])
                iswap(&a[j], &a[j + 1]);
        }
}
```

You might find this code difficult to understand. However, observe that we have changed only the control expressions in the `for` loops. The outer loop is straightforward. As we have  $n$  unsorted numbers in the initial list, the initialization is done as `i = n` and the loop is set up to perform  $n - 1$  passes as required. The inner loop is set up to compare adjacent pairs starting from the beginning of the array as in the previous implementation. As variable `i` refers to the number of elements in the unsorted list, we now require  $i - 1$  comparisons in the inner `for` loop. Thus, the initial and final values of loop variable `j` are  $0$  and  $i - 1$ .

We can also come up with other implementations of bubble sort by constructing the sorted list at the beginning of the array rather than at the end. Now we have to compare adjacent pairs starting from the end of the array. One such implementation is given below.

```
void ivec_bubble_sort(int a[], int n)
{
    int i, j;

    for (i = 0; i < n-1; i++)      /* n-1 passes */
        for (j = n-1; j > i; j--) { /* n-1-i comparisons */

```

```

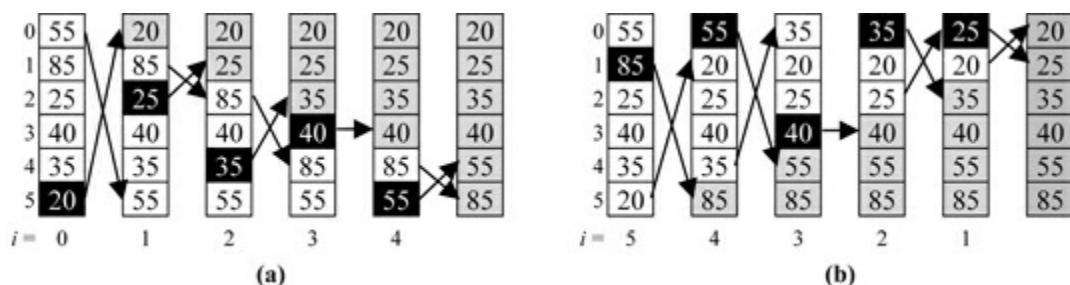
        if (a[j-1] > a[j])
            iswap(&a[j-1], &a[j]);
    }
}

```

Now, variable **i** refers to the number of sorted elements in the list and the outer loop has been set up to perform  $n - 1$  passes. In the  $i$ th pass, the unsorted list contains  $n - i$  elements requiring  $n - i - 1$  comparisons. To perform these comparisons starting from the end of the array, the inner **for** loop is set up with initial and final expressions as  $j = n - 1$  and  $j > i$ , respectively. Also observe that the **if** statement has been modified to compare elements  $a[j-i]$  and  $a[j]$ .

### Selection Sort: Efficient Implementation

The **ivec\_selection\_sort** function given in Section 15.2.2 is quite inefficient as it performs too many exchange operations. We can improve its efficiency by avoiding these exchanges. In the  $i$ th pass of improved sort, we first determine **min\_pos**, the position of the minimum element from the unsorted list and then exchange it with element  $a[i]$ . Such a sort is illustrated in Fig. 15.5a where the minimum values are selected in each pass (shown as black boxes) and they are arranged from the beginning of the array. The code for this modified approach is given below. Note that loop variable **i** is the position where the minimum element is positioned.



**Fig. 15.5** Efficient selection sort: (a) select minimum values, (b) select maximum values

```

void ivec_selection_sort(int a[], int n)
{
    int i, j;

    for (i = 0; i < n - 1; i++) {
        /* determine position of element having minimum value */
        int min_pos = i;

        for (j = i + 1; j < n; j++) {
            if (a[j] < a[min_pos])
                min_pos = j;
        }

        if (min_pos != i)
            iswap(&a[min_pos], &a[i]);
    }
}

```

```

    }

    iswap(&a[i], &a[min_pos]); /* exchange a[i] & a[min_pos] */
}
}

```

Fig. 15.5b shows an alternative approach for selection sort in which we select the maximum value from the unsorted list and position it at the beginning of the sorted list that is constructed from the end of the array. The index variable *i* now takes values as  $n-1, n-2, \dots, 2, 1$  which are positions where selected elements are positioned. The code segment for this approach is given below.

```

void ivec_selection_sort(int a[], int n)
{
    int i, j;

    for (i = n-1; i > 0; i--) {
        /* determine position of element having maximum value */
        int max_pos = i;

        for (j = 0; j < i; j++) {
            if (a[j] > a[max_pos])
                max_pos = j;
        }

        iswap(&a[i], &a[max_pos]); /* exchange a[i] & a[max_pos] */
    }
}

```

### Insertion Sort: Efficient Implementation

The insertion sort function given in Section 15.2.3 is also inefficient as it performs repeated exchange operations in the inner loop to insert an element in the sorted list. The efficiency of this algorithm can be improved by avoiding these exchange operations as shown below.

```

void ivec_insertion_sort(int a[], int n)
{
    int i, j;

    for (i = 1; i < n; i++) {
        int temp = a[i];

        for (j = i; j > 0 && a[j-1] > temp; j--)
            a[j] = a[j-1];
    }
}

```

```
    a[j] = temp;
}
}
```

In this approach, we first copy the element being inserted, `a[i]`, in a temporary variable `temp` and then set up the inner `for` loop. Observe that the termination condition of the inner loop is modified to compare the previous element with variable `temp` as `a[j-1] > temp`). Instead of exchanging the elements in each iteration, we simply copy the elements from the previous array position as `a[j] = a[j-1]` until the correct position is found for the element being inserted. After this, variable `temp` is copied to position `a[j]`.

### 15.3.2 Sorting Strings

As we know, there are two ways to store a list of strings: a two-dimensional array of type `char` and array of character pointers (`char *arr[]`). The use of a two-dimensional array of type `char` leads to very inefficient implementation of sorting algorithms as the entire string will have to be copied during the exchange operation. On the other hand, an array of `char` pointers allows an efficient implementation as only the string pointers are exchanged during the exchange operation rather than copying the entire strings.

In this section, we study how to sort the strings stored in an array of `char` pointers. For example,

```
char *names[] = {
    "Rahul", "Rakesh", "Natasha", "Apurva", "Fardeen", "Martin"
};
```

The function to exchange the values of two character pointers was presented in Example 11.3c and is given below.

```
void pstr_swap(char **str1, char **str2)
{
    char *temp = *str1;
    *str1 = *str2;
    *str2 = temp;
}
```

The function `svec_bubble_sort` to sort the strings in ascending order is given below.

```
void svec_bubble_sort(char *str[], int n)
{
    int i, j;
```

```

for (i = 0; i < n - 1; i++)
    for (j = 0; j < n-1-i; j++) {
        if (strcmp(str[j], str[j+1]) > 0)
            pstr_swap(&str[j], &str[j+1]);
    }
}

```

Having understood this, the implementation of functions for selection sort and insertion sort is quite easy and is left as an exercise.

### 15.3.3 Standard Library Functions for Searching and Sorting

Since sorting and searching are very common operations required by C programmers, the C standard library provides functions for the best searching and sorting techniques. These include functions for binary search (`bsearch`) and quick sort (`qsort`). They are summarized in Table 15.1.

Recall that the searching and sorting functions presented thus far are very restrictive as they work with only integer arrays. If we require any of these operations to be performed on arrays of different data type, we need to rewrite the corresponding function for the desired type. However, the `bsearch` and `qsort` functions are very general as they can be used with arrays of almost any data type including the standard and user-defined data types. This is called **polymorphic behaviour**. Moreover, these functions use a user-defined function for comparing array elements, allowing an array to be searched or sorted in a desired manner.

**Table 15.1** Utility functions declared in the `stdlib.h` header file for searching and sorting

Function	Typical call	Description
<code>bsearch</code>	<code>bsearch(&amp;k, arr, n, w, fcmp)</code>	Performs binary search: Searches the given key $k$ in an array $arr$ containing $n$ elements each of size $w$ bytes. The elements are compared using the user-defined function $fcmp$ . If key $k$ is found in $arr$ , it returns a pointer to it and <code>NULL</code> otherwise.
<code>qsort</code>	<code>qsort(arr, n, w, fcmp)</code>	Sorts a given array: Sorts a given array $arr$ containing $n$ elements each of size $w$ bytes. The elements are compared using user-defined function $fcmp$ .

### Writing Comparison Functions

Before we study the `bsearch` and `qsort` functions, let us understand how to write a comparison function `fcmp`. The declaration of this function from prototypes of `bsearch` and `qsort` functions is as shown below.

```
int (*fcmp) (const void *, const void *)
```

Thus, `fcmp` is a pointer to a function that takes two parameters, both of type `const void*`. These

parameters point to the elements being compared during searching and sorting operations. Note that in `qsort`, they point to elements to be compared from the array being sorted; in `bsearch`, the first parameter points to the element being searched (also called the *key*) and the second parameter points to another element with which the key is being compared. Since the function parameters are `void` pointers, elements of different types can be compared when this function is called from within the `bsearch` or `qsort` functions, allowing polymorphic behaviour.

Let us assume that the parameters of the comparison function are named `pa` and `pb`. The body of a comparison function should compare the elements pointed to by the pointer parameters (`*pa` and `*pb`) and return an integer value similar to that returned by `strcmp` function. Thus, for an ascending order array, the function should return a negative value if `*pa < *pb`, 0 if `*pa == *pb` and a positive value if `*pa > *pb`. For a descending order array, these values should be reversed. Thus, the function should return a positive value if `*pa < *pb` and a negative value if `*pa > *pb`.

Note that generally a separate comparison function is required for each combination of array data type and sort order (ascending, descending, etc.). For example, let us use functions `icmp` and `ircmp` to compare integer data, where `icmp` is used for `int` arrays in ascending order and `ircmp` for `int` arrays in descending order.

We can use two approaches to define a comparison function. In the first approach, which is commonly used, we use parameters of type “constant pointer to specific data type”, e. g., `const int *` for the `icmp` function. Thus, the `icmp` function can be defined using a straightforward approach as shown below.

```
/* comparison function for bsearch & qsort (ascending int array) */
int icmp(const int *pa, const int *pb)
{
    if (*pa < *pb)
        return -1;
    else if (*pa == *pb)
        return 0;
    else return 1;
}
```

However, we can write this function in a very concise manner by returning the difference of the values being compared, i. e., `*pa - *pb`, as shown below.

```
/* comparison function for bsearch & qsort (ascending int array) */
int icmp(const int *pa, const int *pb)
{
    return *pa - *pb;
}
```

It can be easily verified that this function returns correct values as required. We can now write the **ircmp** function for a descending order array by just returning the value **\*pb - \*pa**.

```
/* comparison function for bsearch & qsort (descending int array)*/  
int ircmp(const int *pa, const int *pb)  
{  
    return *pb - *pa;  
}
```

Another comparison function, **fcmp**, for an ascending array of **float** type is given below. Note that the difference **\*pa - \*pb**, which is a **float** value, is converted to **int** using a typecast.

```
/* comparison function for bsearch & qsort (ascending float array)*/  
int fcmp(const float *pa, const float *pb)  
{  
    return (int) (*pa - *pb);  
}
```

We can write similar comparison functions for arrays of other built-in data types. However, note that the signatures of the comparison functions given above do not match with that required by **bsearch** and **qsort** functions. Thus, while using these comparison functions in a call to the **bsearch** and **qsort** functions, we have to use a typecast as illustrated below for the **icmp** function to avoid a compiler warning.

```
(int (*)(const void *, const void *)) icmp
```

It is possible to altogether eliminate such a complex typecast by using another approach for writing comparison functions in which the function parameters are defined as **const void \***, as required by the **bsearch** and **qsort** functions. For example, the **icmp** function can be written as shown below.

```
/* comparison function for bsearch & qsort (ascending int array) */  
int icmp(const void *pa, const void *pb)  
{  
    return *(int *) pa - *(int *) pb;  
}
```

Observe how the function parameters (**const void** pointers) are converted to the desired type (**int** pointer) by using typecast before they are dereferenced. This is essential for two reasons: first, we cannot dereference **void** pointers and second, we wish to interpret the values pointed to as integer values.

Finally recall that we can use the **bsearch** and **qsort** functions for user-defined data types such as

structures. The comparison functions for such arrays are presented later in this section.

## Quick Sort

The C standard library provides the **qsort function** which is an implementation of quick sort. The prototype of this function provided in the **stdlib.h** header file is as follows:

```
void qsort (void *base, size_t nelem, size_t width,  
           int (*fcmp)(const void *, const void *));
```

This function has four parameters, namely, **base**, **nelem**, **width** and **fcmp**. The last parameter is a pointer to a comparison function, whereas the array to be sorted is specified using the first three parameters: **base**, **nelem** and **width**. The **base** parameter is a **void** pointer to the beginning of the array, **nelem** specifies the number of elements in the array and **width** specifies the size of the array element in bytes. The width is usually specified using the **sizeof** operator. Note that as the **base** parameter is a **void** pointer, we can pass an array of any type to this function (polymorphic behavior). Note that although the **qsort** function will generally be used to sort entire arrays, we can sort a part of an array as well. For example, to sort elements from index 5 to 10 in array **a**, we can call the **qsort** function with **&a[5]** and 6 as arguments for **base** and **nelem**, respectively.

The last parameter, **fcmp**, is a pointer to a comparison function that takes two parameters of type **const void \*** and returns an integer value as already discussed in the previous section. When this function is called from the **qsort** function, its arguments point to the array elements being compared during the sort operation.

### Example 15.1 Using the **qsort** function

The **main** function given below illustrates the use of the **qsort** function to sort an array of type **int** in ascending and descending order.

```
int main()  
{  
    int a[] = {5, 3, 4, 1, 8, 7, 10, 2, 6, 9};  
  
    int n = sizeof(a) / sizeof(a[0]); /* no of elems in a[] */  
  
    printf("Sort integer array:\n");  
    ivec_print("Given array", a, n);  
  
    qsort(a, n, sizeof(a[0]), (int (*)(const void *, const void *)) ic  
    ivec_print("Sorted in ascending order", a, n);  
  
    qsort(a, n, sizeof(a[0]), (int (*)(const void *, const void *)) ir
```

```

    ivec_print("Sorted in descending order", a, n);

    return 0;
}

```

Initially, the number of elements in array `a` is determined by dividing array size by the size of element `a[0]`. This is a good idea as it allows us to add or remove elements from the array without worrying about exactly how many elements are initialized in it. Then the array is printed using the `ivec_print` function.

The program uses two comparison functions, `icmp` and `ircmp`, whose definitions are provided earlier in this section (implemented using `const int *` parameters). The `icmp` and `ircmp` functions are used in the `qsort` function call to sort an integer array in ascending and descending order, respectively. The sorted arrays are printed using the `ivec_print` function. The definitions of these functions are omitted to save space.

Note that we could have specified the size of an array element as `sizeof(int)`. However, the use of expression `sizeof(a[0])` is better as it prevents errors in case we later change the array type, say to `long int`. The output of the above function is given below.

```

Sort integer array:
Given array: 5 3 4 1 8 7 10 2 6 9
Sorted in ascending order: 1 2 3 4 5 6 7 8 9 10
Sorted in descending order: 10 9 8 7 6 5 4 3 2 1

```

## Binary Search

The C standard library provides the `bsearch` function to perform binary search on a sorted array. The prototype of this function provided in the `stdlib.h` header file is as follows:

```

void *bsearch (const void *key, const void *base, size_t nelem, size_t width,
               int(*fcmp)(const void *, const void *));

```

This function has five parameters, namely, `key`, `base`, `nelem`, `width` and `fcmp`. The first parameter is a pointer to the element to be searched, the next three parameters specify the array to be searched as explained in the `qsort` function and the last parameter is a pointer to a comparison function. Note that the array being searched must be sorted (either ascending or descending) and as with the `qsort` function, the `bsearch` function can be used to search an entire array or any part of it.

The last parameter, `fcmp`, is a pointer to a comparison function that takes two parameters of type `const void *` and returns an integer value as already discussed. When this function is called from the `bsearch` function, the first argument passed is a pointer to the element being searched (the `key`) and the second argument is a pointer to an array element.

If the element being searched (key) is found in the array base, this function returns a pointer to it. We can easily determine the index of this element by subtracting the value of base from this pointer. However, if the search is unsuccessful, the function returns a NULL pointer.

### Example 15.2 Binary search using the bsearch function

The `main` function given below illustrates the use of the `bsearch` function.

```
int main()
{
    int primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
    int n, key, *ptr;

    n = sizeof(primes) / sizeof(primes[0]); /* no of elems in primes[]
    ivec_print("Prime numbers", primes, n);

    printf("Enter element to be searched: ");
    scanf("%d", &key);

    ptr = bsearch(&key, primes, n, sizeof(primes[0]),
                  (int (*)(const void *, const void *)) icmp);
    if (ptr)
        printf("%d found at position %d\n", key, ptr - primes);
    else printf("%d not found\n", key);

    return 0;
}
```

The above function searches an integer array named `primes` which is initialized with the first 10 prime numbers in ascending order of their value. Initially, the number of elements in `primes` array is determined by dividing the size of `primes` by the size of `primes[0]`. The `primes` array is then printed using `ivec_print` function and an integer number to be searched is read from the keyboard into variable `key`.

Then `bsearch` function is then called to search the value of `key` in `primes` array. The `icmp` function developed earlier in this section is used as a comparison function. The pointer value returned by `bsearch` is assigned to pointer variable `ptr` which is then tested using an `if-else` statement and an appropriate message is printed. If `ptr` is not `NULL`, i. e., if the search is successful, the element position is also displayed using the expression `ptr - primes`.

## Searching and Sorting Arrays with More Complex Data

We know that the `bsearch` and `qsort` functions are polymorphic and can be used with arrays of built-in as well as user-defined data types. However, we have to write a separate comparison function for each data type and sorting order.

In this section, we will study how to use the `qsort` function to sort arrays containing more complex data. As the use of the `bsearch` function is very similar to that of the `qsort` function, its discussion is avoided to save space.

### ***Sorting strings (two-dimensional array of type `char`)***

As we already know, we can store a list of strings in a two-dimensional array of type `char` as in

```
char city[][][20] = {"Pune", "Mumbai", "Delhi", "Chennai", "Kolkata"};
```

In this case, the `strcmp` and `stricmp` functions from the C standard library can be used as string comparison functions, where the `stricmp` function ignores character case. However, note that these functions will work only for arrays in ascending order. The comparison functions for arrays in descending order can be written easily by returning negated values returned by the `strcmp` or `stricmp` functions. For example, the `strrcmp` function for an array of strings in reverse (i. e., descending) order is given below.

```
/* qsort comparison function (for descending 2D array of char) */
int strrcmp(const char *pa, const char *pb)
{
    return -strcmp(pa, pb);
}
```

The `qsort` function call given below sorts the `city` array in ascending order using the `strcmp` standard library function as a comparison function.

```
qsort(city, 5, 20, (int (*)(const void *, const void *)) strcmp);
```

To sort the array in descending order, we should replace the `strcmp` function in the above call with the `strrcmp` function given above.

### ***Sorting strings (array of pointers to `char`)***

Note that the use of a two-dimensional array of type `char` to store strings to be sorted is a very inefficient way as the sort algorithm exchanges the strings by copying their entire contents. We can avoid these string copy operations and thereby greatly improve the sort efficiency by using an array of character pointers where the pointers will be exchanged rather than the actual strings, as explained in Example 11.3c.

Consider an array of character pointers given below used to store the names of five cities.

```
char *city[] = {"Pune", "Mumbai", "Delhi", "Chennai", "Kolkata"};
```

Each element in this array is actually a pointer to `char` type, i. e., `char *`. We know that during sort operation, the `qsort` function passes pointers to two elements from this array to the comparison function. Thus, the parameters passed to the comparison functions are of type `char **`. However, the `strcmp` and `stricmp` standard library functions expect parameters of type `char *`. Thus, these functions cannot be directly used as comparison functions and we need to define our own comparison functions. The comparison function `pstrcmp` for an array in ascending order is given below.

```
/* qsort comparison function (ascending array of char pointers) */
int pstrcmp(const char **pa, const char **pb)
{
    return strcmp(*pa, *pb);
}
```

Also, as already explained, the comparison function `pstrcmp` for an array in descending order is obtained by negating the value returned by the `strcmp` function in the above function definition.

The `qsort` function call to sort the city array in ascending order is given below.

```
qsort(city, 5, sizeof(city[0]),
      (int (*)(const void *, const void *)) str_cmp);
```

### *Sorting array of structures*

Consider that we wish to sort an array of dates using the `qsort` function. For this, we must first write a comparison function that accepts two parameters of type `struct date *`. In Program 13.3, we studied the `date_cmp` function for the comparison of two dates that used parameters of type `struct date` instead of `struct date *`. The modified function with the same name (`date_cmp`) but pointer parameters is given below.

```
/* qsort comparison function (for ascending array of dates) */
int date_cmp(const struct date *pa, const struct date *pb)
{
    if (pa->dd == pb->dd && pa->mm == pb->mm && pa->yy == pb->yy)
        return 0;

    else if (pa->yy > pb->yy ||
             pa->yy == pb->yy && pa->mm > pb->mm ||
             pa->yy == pb->yy && pa->mm == pb->mm &&
                           pa->dd > pb->dd)
        return 1;
```

```
    else return -1;
}
```

The `main` function given below first creates an array of dates, calculates the number of elements in it, sorts it using `qsort` function in ascending order and then prints the sorted array using `date_vec_print` function (whose implementation is left as an exercise).

```
int main()
{
    struct date d[] = {{27,8,2006}, {24,6,1972}, {7,3,1965},
                        {24,10,1998}, {10,9,2007}, {30,4,2012}};
    int n = sizeof(d) / sizeof(d[0]);

    qsort(d, n, sizeof(d[0]),
          (int (*) (const void *, const void *)) date_cmp);

    date_vec_print(d, n);

    return 0;
}
```

Note that the `date_cmp` function given above sorts the dates in ascending order. To sort the dates in descending order, we can obtain the `date_rcmp` function by replacing the return values 1 and  $-1$  with  $-1$  and 1, respectively. However, we can also implement the `date_rcmp` function by negating the values returned by `date_cmp` function as shown below.

```
/* qsort comparison function (for descending array of dates) */
int date_rcmp(const struct date *pa, const struct date *pb)
{
    return -date_cmp(pa, pb);
}
```

## Exercises

1. Answer the following questions in brief:
  - a. State the names the searching techniques.
  - b. State the difference between linear search and binary search with respect to the order of elements in the array being searched.
  - c. Justify the name *insertion sort*.
  - d. Explain the basic principle used in bubble sort.

- e. State names of any five sorting techniques
  - f. How many passes are required to sort an array having 100 elements using insertion sort?
  - g. Can a linear search be performed on a sorted array and a binary search on unsorted array?
2. Write C code for the following:
- a. Function to perform linear search on an array of type `long double`.
  - b. Non-recursive function to perform binary search on an array of `long int` sorted in descending order.
  - c. Recursive function to perform binary search on an array of `double` sorted in ascending order.
  - d. Function to sort elements of a character string using bubble sort. Do not use swap function.
  - e. Sort an array of short integers using insertion sort. Also, define and use a swap function to exchange array elements.
3. Assume that an array of `int` is initialized with following elements: 10, 8, 17, 12, 15, 4, 6, 3, 21, 13 and 11. With the help of neat illustrations explain how
- a. linear search will be performed on this array for following elements: 17 and 25
  - b. binary search will be performed for following elements: 15, 8 and 20.
4. Assume that an array of `int` is initialized with following elements: 13, 5, 8, 2, 16, 11 and 4. With the help of neat illustrations explain how these elements will be sorted
- a. in descending order using bubble sort .
  - b. in ascending order using insertion sort .
  - c. in descending order using selection sort.
5. Write complete programs for the following problems. Define and use functions for various operations involved.
- a. Accept several integer numbers from keyboard, sort them using bubble sort and display the sorted array.
  - b. Initialize an integer array using random number generator, sort this array using selection sort and display sorted array. Then search several numbers from this array using a non-recursive binary search.
  - c. Read a string from a keyboard and extract distinct letters from this strings (ignoring case) to a character array. Sort this array in descending order using insertion sort. Then read another string from keyboard and then determine how many letters from this second string are present in sorted character array using recursive binary search.

## Exercises (Advanced Concepts)

- l. Answer the following questions in brief:
  - a. Name the standard library functions for searching and sorting operations.
  - b. What is a polymorphic function? Explain how polymorphism has been implemented with `qsort` standard library function.
  - c. Write a comparison function to search an array of `long double` elements in descending order using `bsearch` function.
  - d. Name the parameters and their types for `qsort` standard library function.
  - e. Explain the role of comparison function in `bsearch` function.
2. Write C functions for the following:
  - a. To demonstrate how binary search will be performed on a sorted array. Show the positions of elements at *left*, *mid* and *right* positions in each iteration and write appropriate messages explaining the operations performed.
  - b. Consider the bubble sort presented in Section 15.2.1. The performance of this sorting algorithm can be improved by recording the position of last exchange operation in each pass (as all the elements below this position will be in sorted order). Write a function for such a sort.
  - c. Consider the insertion sort algorithm discussed in Section 15.2.3. The performance of this algorithm can be improved greatly by using the binary search to determine the position for the next element to be inserted in sorted portion of the list in each pass. Write such an algorithm.
  - d. To sort an array of strings represented in an array of pointers using efficient insertion sort given in Section 15.3.1.
3. Write comparison functions (to be used in `qsort` and `bsearch` functions) for the following:
  - a. vector of `float` in ascending order
  - b. vector of `char` in descending order
  - c. vector of `double` numbers in order of their absolute values in descending order
  - d. several strings stored using 2-D array of `char` in ascending order
  - e. strings stored using an array of pointers in descending order
  - f. vector of complex numbers (`struct complex`) in order of their magnitudes in ascending order
  - g. array of `struct Student` (with members `fname` and `lname`), in ascending order of `lname` and `fname`.
  - h. array of pointers to `struct Employee` (with members `name`, `dept` and `salary`), in department wise descending order of `salary`.
4. Write complete programs for the following. Define and use functions for various operations involved.

- a. Initialize an integer array with 2-digit values using random number generator, sort this array using `qsort` function and display sorted array. Then use `bsearch` function to search this array several times, as decided by the user.
- b. Initialize a two dimensional array of type `char` with names of several cities. Sort this array using `qsort` function and then search it using `bsearch` function. Ignore case during sort and search operations.
- c. Consider a structure `Player` having two fields: `name` and `runs` scored in a cricket match. Write a program to accept data of 11 players in a team in an array of structures and sort this data using `qsort` function in decreasing order of runs scored. Then search this array using `bsearch` function for given player's name and display runs scored by him/her.
- d. Rewrite above program using an array of pointers to `struct Player` instead of an array of `struct Player`.

# 16 Miscellaneous Concepts

This chapter presents some miscellaneous concepts in the C programming language. These include bitwise operators, enumerated constants, user-defined types (`typedef`) and some details about the standard C library.

## 16.1 Bitwise Operators

The C language provides six **bitwise operators** to manipulate the bit patterns of integral values (integers and characters). They include **not**, i. e., **complement** (`~`), **and** (`&`), **or** (`|`), **exclusive or**, i. e., **xor** (`^`), **left shift** (`<<`) and **right shift** (`>>`). These operators work directly on the bit patterns of the operands, i. e., on sequences of bits (0 and 1) representing the operands. In addition, C language provides five compound assignment operators (`&=`, `|=`, `^=`, `<<=` and `=`).

The **complement** operator (`~`) is a unary prefix operator and is used, as in `1`, whereas all other operators are binary infix operators and are used as in `a op b`.

First consider these bitwise operations on individual bits. The **bitwise and** operator evaluates as 1 if both operands are 1, and zero otherwise. The **bitwise or** operator evaluates as 1 if either or both operands is 1, and zero otherwise. The **bitwise xor** operator evaluates as 1 if either of the operands (but not both) is 1, and zero otherwise. Finally, the **bitwise not** operator complements the value of the operand, i. e., it returns 1 if the operand is zero and vice versa. These operations are summarized in Table 16.1.

**Table 16.1** Evaluation of bitwise operators on 1 bit values

Inputs		and	or	xor
a	b	<code>a &amp; b</code>	<code>a   b</code>	<code>a ^ b</code>
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Input	not
<code>a</code>	<code>~a</code>
0	1
1	0

While working with integral numbers, the bitwise operations are performed on all bits. As we know, `char` values are represented using 1 byte (i. e., 8 bits), `int` values using either 2 or 4 bytes, `short int` using 2 bytes and `long int` using 4 bytes. These values may be signed or unsigned. However,

the working of binary operators is illustrated here using 4-bit numbers **a** and **b** having decimal values 11 (binary **1011**) and 7 (binary **0111**), respectively.

The bitwise **and**, **or** and **xor** operations are performed on corresponding bits of two integer operands by applying bit operations, as shown in Table 16.1. Thus, **a & b** (i. e., **1011 & 0111**) evaluates as **0011**, i. e., decimal 3 as shown in Fig. 16.1. Also, **a | b** evaluates as **1111** (decimal 15) and **a ^ b** evaluates as **1100** (decimal 12).

Variable	Decimal value	<b>a &amp; b</b>	<b>a   b</b>	<b>a ^ b</b>
<b>a</b>	11	1011	1011	1011
<b>b</b>	7	0111	0111	0111
	output (binary)	0011	1111	1100
	output (decimal)	3	15	12

**Fig. 16.1** Effect of bitwise and, or and xor operators on 4-bit unsigned int numbers **a=11** and **b=7**

The complement operation is performed on all the bits of a number. Thus, the expression  $\sim a$  evaluates as **0100** (decimal 4). This complement operation is also called as 1's complement.

The **left shift** (`<<`) and **right shift** (`>>`) are binary infix operators used, as in `val << n` and `val >> n`. They shift the bits in the first operand (`val`) by a number of bit positions specified in the second operand (`n`).

Note that when a value is shifted left, the empty bit positions on the right are filled with 0 bits. Also, the bits that move out of the number are lost. Thus, expression **a << 1** left-shifts bits in variable **a** (binary **1011**) by 1 bit position to obtain **0110** (decimal 6) and expression **a << 2** left-shifts bits in variable **a** by 2 bit positions to obtain **1100** (decimal 12) as shown in Fig. 16.1.

Operation	not	Left shift		Right shift			
		Unsigned	Signed	Unsigned	Signed		
<b>Bits in variable</b>	<b>~a</b>	<b>a &lt;&lt; 1</b>	<b>a &lt;&lt; 2</b>	<b>a &gt;&gt; 1</b>	<b>a &gt;&gt; 2</b>	<b>a &gt;&gt; 2</b>	<b>b &gt;&gt; 2</b>
<b>Output</b>	1011	1011 ↓↓↓↓	1011 ↓↓↓↓	1011 ↓↓↓↓	0101 ↓↓↓↓	0010 ↓↓↓↓	0111 ↓↓↓↓

**Fig. 16.2** Effect of bitwise not, left-shift and right-shift operators on 4-bit int numbers (variable **a** is an unsigned int and **c** is a signed int)

The working of the right shift operator depends on whether the number being shifted is **signed** or **unsigned**. When an unsigned number is shifted to the right, the empty bit positions on left (i. e., MS bits) are filled with 0 bits. This shift is called **logical right shift**. Also, note that the bits that move out of a number are lost. For example, assuming that variable **a** (binary **1011**) is an **unsigned int**

variable, the expression `a >> 1` evaluates as `0101` (decimal 5) and `a >> 2` as `0010` (decimal 2).

When a `signed int` is shifted to the right, the bits are shifted as usual, except that the MS bit is copied to itself. Thus, the empty MS bit positions all become equal to the sign bit. This shift is called the **arithmetic right shift**. For example, assuming `a` (binary `1011`) and `b` (binary `0111`) to be `signed int` variables, the expression `a >> 2` evaluates as `1110` and expression `b >> 2` evaluates as `0001`.

Observe that a right shift by 1 bit position is equivalent to integer division by 2. Also, a left shift by 1 bit position is equivalent to multiplication by 2, provided the most significant bit of the number being shifted is 0.

The bitwise compound assignment operators are similar to other compound assignment operators. For example, the assignment expression `a &= expr` is equivalent to the expression `a = a & (expr)`.

### Precedence and Associativity of Bitwise Operators

The bitwise complement operator is a unary operator and has the precedence and associativity as other unary operators. Thus, its precedence is higher than the arithmetic operators and it has right-to-left associativity.

All other bitwise operators have left-to-right associativity. The precedence of the bitwise shift operators is just below that of the arithmetic operators and higher than that of the relational operators. The bitwise `and`, `xor` and `or` operators have precedence (in that order) below that of the equality operators (`==` and `!=`) and above that of the logical operators (`&&` and `||`) as seen in Appendix B.

Finally, the bitwise compound assignment operators (`&=`, `^=`, `|=`, `<<=` and `>>=`) have the precedence and associativity as other assignment operators. Thus, they have precedence below that of the conditional operator (`? :`) and above that of the comma operator (`,`) and right-to-left associativity.

## 16.2 Enumerated Types

Consider that we need to work with the colours in a rainbow, e. g., to paint a rainbow on the screen. We thus have to work with seven colours, namely, violet, indigo, blue, green, yellow, orange and red. These colours can be represented using integer values starting with 0. This enables us to use various program constructs such as conditions or loops to process these colours. However, programs written using such code often become difficult to understand as can be seen from the statement given below.

```
color = 4;
```

Although we can add a comment on such a line, it would be much more readable if we directly use the colour names in our programs, as in

```
color = Yellow;
```

One solution is to use symbolic constants using the `#define` statements:

```
#define VIOLET 0
#define INDIGO 1
#define BLUE 2
#define GREEN 3
#define YELLOW 4
#define ORANGE 5
#define RED 6
```

This is inconvenient, particularly when there are a large number of constants to be defined. Moreover, program debugging is difficult as these symbolic constants are not available during the debugging phase.

A better solution to this problem is to declare an **enumerated type** containing **enumerated constants**. An enumerated type is a user-defined type with values that can be represented using names instead of integers. For example, we can use enumerated constants Jan, Feb, ..., Dec to represent months rather than the integer numbers 1 to 12.

### 16.2.1 Declaring Enumerated Types

The format to declare an enumerated type is

```
enum enum_type { EnumConst1, EnumConst2, ..., EnumConstN };
```

This declares a type named *EnumType* and enumerated constants *EnumConst1*, *EnumConst2*, ..., *EnumConstN*. These constants have consecutive integer values starting from 0.

Consider the following declaration for our rainbow example.

```
enum color {Violet, Indigo, Blue, Green, Yellow, Orange, Red};
```

This statement declares an enumerated type named **color** and seven enumerated constants, namely, **Violet**, **Indigo**, **Blue**, **Green**, **Yellow**, **Orange** and **Red** with consecutive values from 0 to 6. Note that these constants are written in capitalized case to distinguish them from variable names.

Like other declarations, enumerated types can be defined at the beginning of any function or block, with the only requirement that the declaration must precede their use. However, such local declarations will restrict their use to that function or block only. A better approach is to define enumerated types at the beginning of the program, immediately after the `#include` statements, enabling their use in subsequent programs (structure declarations and functions).

The enumerated data types are useful in numerous programming situations where we refer to the individual items by specific names rather than integer numbers. For example, the names of days and months, keywords of a programming language, program menu items, educational qualifications, class obtained in examination, names of animals and birds, models of various products such as cars, mobiles, processors, soap and so on. Consider the declarations of several enumerated types given below.

```
enum day {Sunday, Monday, Tuesday, Wednesday, Thursday,  
          Friday, Saturday};  
enum file_menu {New, Open, Close, Save, SaveAs, Print, Exit};  
enum qualification {SSC, HSC, Diploma, BE, BTech, ME, MTech, PhD};  
enum class {Dist, First, Second, Pass};  
enum mobile_operator {Aircel, Airtel, BSNL, Docomo, Idea, RCom};
```

Readers are encouraged to use such enumerated types in their programs to improve program readability.

### 16.2.2 Variables of Enumerated Types

Once an enumerated type is declared, we can declare (and also initialize) variables of that type using the formats given below.

```
enum enum type var1, var2, ...;  
enum enum type var1 = enum_val1, var2 = enum_val2, ...;
```

Note that the keyword **enum** must precede the enumerated type name. Thus, once we declare enumerated type **color** as explained earlier, we can declare (and also initialize) variables of type **color** as shown below.

```
enum color clr;  
enum color c1 = Violet, c2 = Red;
```

The declaration of an enumerated type and variables can also be combined. The variables are written between the closing brace and the semicolon. These variables can also be initialized. Consider the declaration given below.

```
enum dir {Up, Down, Left, Right} d1 = Left, d2 = Right, dir;
```

This statement declares an enumeration **dir** and three variables **d1**, **d2** and **dir**. The variables **d1** and **d2** are initialized with the values **Left** and **Right**, respectively. Note that the name of an enumeration (**dir**) can also be used as a variable name. Observe that combining variable declarations with that of an enumeration makes the code somewhat difficult to read and should generally be avoided.

Further note that we can omit enumeration name in the above declaration as

```
enum {Up, Down, Left, Right} d1 = Left, d2 = Right, dir;
```

However, this approach is suitable only if we do not have to subsequently use the enumeration name in our program, e. g., to declare more variables, to convert type using typecast, or to pass parameters to a function.

### 16.2.3 Specifying Values for Enumerated Constants

By default, the enumerated constants are assigned consecutive values starting from 0. However, we can specify the desired integer values for one or more enumerated constants. The subsequent constants are assigned the next integer values. For example, consider the declaration of enumerated type `month` given below.

```
enum month {Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,  
          Oct, Nov, Dec};
```

The constant `Jan` is assigned value 1 and subsequent constants are assigned values from 2.

Now consider another declaration of enumerated type `color`:

```
enum color {Violet, Indigo, Blue=10, Green, Yellow=20, Orange, Red};
```

First, the constants `Violet` and `Indigo` are assigned values 0 and 1, respectively. As the constant `Blue` is assigned value 10, constant `Green` is assigned value 11. Also, as `Yellow` is assigned value 20, constants `Orange` and `Red` are initialized with values 21 and 22, respectively.

Further note that the values assigned need not be in any specific order. Moreover, two or more constants may have same values as illustrated in the example given below.

```
enum subject {Physics=40, Chemistry, Biology, Algebra=40, Geometry};
```

### 16.2.4 Working with Enumerated Data

Consider the following variable declarations of enumerated types `day` and `month` whose declarations are given earlier in this section.

```
enum day d, d1, d2;  
enum month m, m1, m2;
```

Since enumerated constants and variables are of integer type, we can perform most of the operations on enumerated entities (variables and constants) that we can perform on integer entities. Thus, we can assign values of integer as well as enumerated variables (and constants) to other enumerated variables. For example,

```
d1 = Monday;  
d2 = d1;  
m1 = 5; /* May */
```

However, we should be careful as C compilers will not give any error or warning if we assign incorrect values as shown below.

```
d1 = Feb; /* d1 is assigned value 2, which is Tuesday */
m2 = 15; /* invalid month */
```

We can use various arithmetic operators (+, -, \*, /, %, ++, --) on enumerated entities. Some of these operations are particularly useful if the enumerated constants have consecutive integer values. For example, we can add or subtract an integer constant to an enumerated entity to obtain the subsequent or previous enumerated value, use - operator to determine the numbers of values between two enumerated entities and the ++ and -- operators to obtain the next and previous values, respectively. Consider the examples given below.

```
d = (d + 5) % 7; /* week day 5 days after d */
m++; /* next month */
```

Note that the C language does not perform any range check on the value being assigned to an enumerated variable.

We can use the `scanf` and `printf` functions to perform input/output operations on enumerated entities. For example,

```
scanf("%d", &m);
printf("%d %d %d\n", m, Monday, Feb);
```

Note that we should enter integer value for variable `m` and the `printf` statement will print integer values. Also note that some compilers may give a warning as the `%d` format expects an argument of type `int *` and not a pointer to enumerated type.

We can also use relational, equality and logical operators with enumeration entities. Consider, for example, the code segment given below to test the value of day `d`.

```
if (d == Sunday || d == Saturday)
    printf("Holiday");
else if (d >= Monday && d <= Friday)
    printf("Weekday");
else printf("Invalid day");
```

If the enumerated constants have consecutive values, we can process them using loops. For example, assuming that an array `month` is used to store month names as

```
char *month[] = {"---", "January", "February", "March", "April",
                 "May", "June", "July", "August", "September",
                 "October", "November", "December"};
```

we can use a `for` loop given below to print the month names.

```
for (m = Jan; m <= Dec; m++)
```

```
printf(" %s\n", month[m]);
```

Observe that enumerated variable `m` is used as an array index.

We can use enumerated variables as function parameters and return values. For example, assuming that function `is_leap` is available to test whether a given year is leap or not, the function `month_days` given below determines the number of days in a given month.

```
int month_days(enum month m, int y)
{
    static int mdays[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30 ,
                          31, 30, 31};
    return mdays[m] + (m == Feb && is_leap(y)? 1 : 0);
}
```

We can declare arrays of enumerated type. For example, consider array `months30` that stores the months that have 30 days.

```
enum month months30[] = {Apr, Jun, Sep, Nov};
```

We can declare pointer variables of enumerated type. Consider the code segment given below.

```
enum month m = Jun, *pm;
pm = &m;
printf("Month: %d Days: %d\n", *pm, month_days(*pm, 2012));
```

Finally, we can use enumerated variables as structure members as illustrated in the declaration given below.

```
struct tv_serial {
    char name[50];
    enum day telecast_day[7];
};
```

## 16.3 Renaming Types Using `typedef`

The **`typedef`** feature allows us to give an alternative (possibly short and more meaningful) name to an existing data type and improve program readability. For example, instead of using the `int` data type to declare variables to represent marks in three subjects, we can associate a more meaningful name (say `Marks`) for the `int` data type using `typedef` as:

```
typedef int Marks;
```

Now we can declare variables using the new type name `Marks` as shown below:

```
Marks phy, chem, math;
```

It should be understood that **Marks** is just another name for the **int** data type and it does not restrict the values of variables **phy**, **chem** and **math** to the desired range, say 0 to 100. Also, we can perform all the operations on these variables that we can perform on variables of type **int** including the use of the **%d** format in the **scanf** and **printf** statements.

The **typedef** can be used to give convenient short names to built-in data types, as shown below.

```
typedef unsigned short int UShort;
typedef long double LDouble;
typedef const unsigned long int ULongConst;
```

These **typedefs** declare **UShort**, **LDouble** and **ULongConst** as convenient shorthands for **unsigned short int**, **long double** and **const unsigned long int**, respectively. Note that while declaring variables using type name **ULongConst**, we must initialize them; otherwise, the compiler will report an error. Also note that we have used capitalized names for **typedef** names as a convention.

We can use other features of the C language in conjunction with the new type names. For example, we can define **const** variables, arrays and pointers using type name **UShort** as

```
const UShort a = 1234;
UShort b[10];
UShort *pa = &a;
```

The **typedef** is a powerful mechanism that allows us to declare new names for complex types that involve arrays, pointers, functions and their combinations. The syntax for such declarations can be quite confusing in the beginning. However, it is not very difficult if we use the following rule: *To declare a new type name for a particular type, first declare this name as if we were declaring a variable of that type and then precede that declaration with the **typedef** keyword.*

For example, to declare **UShort** as a new type name for **unsigned short int**, first declare **UShort** as if it is a variable of that type as

```
unsigned short int UShort;
```

and then precede this declaration with the **typedef** keyword as shown below.

```
typedef unsigned short int UShort;
```

Finally note that the **typedefs** are generally written at the beginning of a program, usually after the **#include** statements. This allows their use in any of the structures, enums and functions including the **main** function. We can also write them in any function, in which case, their use is restricted to that function only.

### 16.3.1 Using `typedef` with Structures

Recall that we have to use the `struct` keyword every time we declare a structure variable or parameter. This inconvenience can be avoided using `typedef`. For example, consider the `typedef` given below.

```
typedef struct complex {  
    float re, im;  
} Complex;
```

This code declares `Complex` as a new type name for `struct complex`. Note that as per our convention, the type name is in capitalized case (`Complex`), whereas the structure name (`complex`) is in lowercase letters. Also, observe how our rule simplifies writing such declarations: the new type name `Complex` is first written as if it is a variable of type `struct complex` and then the declaration is preceded by the `typedef` keyword.

If you prefer, the declarations of structure and `typedef` can be written separately as

```
struct complex {  
    float re, im;  
};  
  
typedef struct complex Complex;
```

Observe that our rule for writing a `typedef` is applicable in this case as well.

Once the new type name `Complex` is declared using either of the approaches given above, we can declare variables using this new type name (as well as by using `struct complex`) as shown below.

```
Complex a = {1, 2};  
struct complex b, c;
```

Now we can simplify the definition of the `complex_add` function (Example 13.5b) using this new type name and greatly improve its readability, as shown below.

```
/* addition of two complex numbers */  
Complex complex_add(Complex a, Complex b)  
{  
    Complex c;  
    c.re = a.re + b.re;  
    c.im = a.im + b.im;  
    return c;  
}
```

Note that as discussed earlier, we can continue to use other features of the C language in conjunction with a new type name. For example, we can declare arrays and pointers of this type as

```
Complex a[10];
Complex *pa;
```

Finally note that the `typedef` name can be the same as that of the structure name. For example,

```
typedef struct Complex {
    float re, im;
} Complex;
```

Also, we can even omit the structure name as shown below.

```
typedef struct {
    float re, im;
} Complex;
```

### 16.3.2 Using `typedef` with Enumerated Types

The previous discussion applies to the enumerated types as well. Thus, we can declare a new type name for enumerated types using `typedef` as

```
typedef enum color {Violet, Indigo, Blue, Green, Yellow,
Orange, Red} RbColor;
```

or if you prefer, we can separate these declarations as

```
enum color {Violet, Indigo, Blue, Green, Yellow, Orange, Red};
typedef enum color RbColor;
```

These declarations define `RbColor` as a new type name for `enum color`. We can use this new type name wherever we can use `enum color`. Thus, we can define various entities using this new type name, such as variables, arrays, pointers, function parameters, structure members, etc. In addition, we can also use `enum color` for this purpose, although it is unnecessary now. For example, we can declare variables as shown below.

```
RbColor clr = Blue;
enum Color c1 = Violet, c2 = Red;
```

Note that the `typedef` name can be the same as that of the enumerated type name and we can also omit the enumerated type name, as in case of `typedefs` for structures.

## 16.4 More on Library Functions and Facilities

### 16.4.1 The `printf` Function

The `printf` is a very powerful function. It was discussed in Sections 4.2.1 and 4.4.3. In this section, we cover additional details of this function.

The **format specifier** of the `printf` function allows more details to be included for further control of the values being printed, as shown below.

%	Flags (-, +, #, blank)	Minimum field width (n, 0n, *)	.	Precision .0, .n, *, none)	Input size modifier (h, l, L)	Conversion character (d, i, f, c, s, u, o, x, X, e, E, g, G, n, p, %)
---	------------------------------	--------------------------------------	---	----------------------------------	-------------------------------------	---

#### Conversion Characters

The format specifier provides a large number of **conversion characters**. Five of them (d, i, f, c and s) have been already discussed in Section 4.2.1. The remaining conversion characters are summarized in Table 16.2. Note that these mainly include the conversion characters for printing unsigned decimal integer numbers and floating-point numbers in scientific notation.

**Table 16.2** Additional conversion characters for the `printf` function (in addition to d, i, f, c and s)

Conversion character	Argument type	Argument converted to
u	int	Unsigned decimal integer
o	int	Unsigned octal integer
x X	int	Unsigned hexadecimal integer using digits a ... f or A ... F
e E	double	Scientific format using e or E for exponent
g G	double	Decimal/scientific notation depending on given value and precision
n	int *	Stores the count of characters printed, in a variable pointed to by the argument
p	Pointer	Pointer value (i. e., memory address) printed in hexadecimal notation
%	None	Prints a % symbol

As we already know, the d and i conversion characters are used to print signed integers as decimal numbers. Four other conversion characters, namely, u, o, x and X, are provided to print unsigned integers. They can also be used to print non-negative values of int type. The conversion characters u and o print such a number in the decimal and octal formats, respectively. The conversion characters x and X, on the other hand, print it as a hexadecimal number using characters a ... f and A ... F,

respectively.

In Section 4.4.3, we studied that the conversion character 'f' is used to print floating-point numbers in the decimal format. Four other conversion characters are provided, namely, e, E, g and G, which allow such numbers to be printed in a scientific format. The default precision for printing floating-point numbers using these conversion characters is 6 digits.

The e and E conversion characters always print floating numbers in scientific format, as in [-]mantissa e ± expo and [-]mantissa E±expo, respectively, where [-] indicates that the sign is printed only for negative numbers and ± indicates the sign is always printed before the exponent. The mantissa is printed as a floating-point number in decimal format as d. ddddddd, i.e., one digit before the decimal point and six digits (the default precision) after it and the value of the exponent is adjusted accordingly. The value of mantissa is rounded to the sixth decimal place. The exponent is printed using two digits in Turbo C/C++ and three digits in Dev-C++ and Code::Blocks.

The conversion character p is used to print a pointer value as a hexadecimal number. The conversion character n does not produce any output. Rather it stores the number of characters printed so far, by the current printf call, in an integer variable pointed to by the corresponding argument. This argument should be a pointer to an integer variable. Finally, note that conversion character % is used to print the % character itself in the output.

### Example 16.1 Using conversion characters in the printf function

a) Consider the printf statements given below.

```
printf("%e\t %e\t %e\n", 120.0, 123456700.0, 987654300.0);
printf("%e\t %e\t %e\n", 1.2345, 1.23456789, 1.23454321);
printf("%e\t %e\t %e\n", 1e2, 123e3, 12345678e10);
```

Study carefully the output obtained in Turbo C++:

1.200000e+02	1.234567e+08	9.876543e+08
1.234500e+00	1.234568e+00	1.234543e+00
1.000000e+02	1.230000e+05	1.234568e+17

The g and G conversion characters print floating-point numbers either in decimal or scientific format depending on the value being printed and the precision to be used, with exponent specified using letters e and E, respectively. By default, the values are printed using at most six digits (excluding the exponent part). The decimal notation is used only if a value being printed can be accommodated in six digits; otherwise scientific notation is used. The output obtained using the %g specification in place of %f in the above printf statements is given below.

120	1.23457e+08	9.87654e+08
1.2345	1.23457	1.23454

100            123000            1.23457e+17

Also note that using the %g specification, the values 123.4567, 123456.7 and 1234567.8 will be printed as 123.457, 123457 and 123457e+06, respectively.

**b)** Consider the program segment given below.

```
int a, b, c;
printf("%p %p %p\n", &a, &b, &c);
printf("%d %n %f %n\n", 100, &a, 1.2, &b);
printf("%s\t\%n\n", "Hello", &c);
printf("Characters printed: a=%d b=%d c=%d\n", a, b, c);
```

When executed in Turbo C++, it produced the output shown below.

```
FFF4 FFF2 FFF0
100 1.200000
Hello
Characters printed: a=4 b=14 c=8
```

The first `printf` statement prints the addresses of variables `a`, `b` and `c` using the `%p` format specifier. Note that the addresses printed in the output may differ depending on the IDE used, other variables declared before these variables and the programming model in Turbo C/C++.

The second and third `printf` statements demonstrate the working of the `%n` format specifier. Observe that in the second `printf` statement, four characters (including a space) are printed before the first `%n` is encountered and fourteen characters are printed before the second one. Similarly, eight characters are printed including three escape sequences in the next `printf` statement. Thus, the values of variables `a`, `b` and `c` are 4, 14 and 8, respectively.

### Minimum Field Width

By default, the `printf` function prints output using as many character positions as required, without any spaces before or after the output values. However, we can set the minimum field width for each value being printed by using one of three options in the format specifier: `n`, `0n` and `*`, where `n` is an integer number. The minimum field width may also be omitted specifying zero field width. Note that if a value being printed cannot be accommodated in the specified field, the value is not truncated. Rather, additional character positions are used.

As we already know, the first option sets a field width of `n` character positions. By default, the values are printed right justified in this field width, i. e., spaces are padded, if required, on the left. The second option (`0n`) is similar to the first except that while printing right-justified numbers, zeros are padded on the left side instead of spaces. The third option (\*) is more flexible and allows the field width to be specified using an argument to the `printf` function, which is written just before the value being printed.

## Input Size Modifier

The input size modifier field allows the values of types `short`, `long` and `long double` to be printed. The `short` modifier (`h`) can be used only for integer values, the `long` modifier (`l`) can be used with either integer or floating-point values and the `long double` modifier (`L`) only with floating-point values. This is summarized below.

Input size modifier	Conversion character	Argument interpreted as
<code>h</code>	<code>d i o u x X</code>	<code>short int</code>
<code>l</code>	<code>d i o u x X</code>	<code>long int</code>
	<code>e E f g G</code>	<code>double</code>
<code>L</code>	<code>e E f g G</code>	<code>long double</code>

## Flags

Four flags can be used immediately after the % symbol. By default, the values are printed right-justified, padded with either spaces or zeros on the left. The use of the '-' flag causes the values to be left justified, with spaces padded on the right. The use of the '+' flag causes the sign to be printed for all numbers, positive as well as negative, whereas a blank flag causes a space to be printed in place of a sign for positive numbers. The '#' flag specifies that the argument should be converted using an alternate form as follows:

Conversion character	Effect of # flag
<code>d i u c s</code>	No effect
<code>o</code>	Prepends <code>0</code> to a non-zero argument which is printed in octal format
<code>x X</code>	Prepends <code>0x</code> or <code>0X</code> to an argument which is printed in a hexadecimal format
<code>e E f</code>	Always prints a decimal point even if there are no digits after it
<code>g G</code>	Always prints a decimal point even if there are no digits after it. Also, does not remove the trailing zeros.

## Precision

In Section 4.4.3, we studied how the precision field can be used to control the number of digits in a floating-point number. In addition, we can use this field to specify either the minimum number of digits to be printed from a number or the maximum number of characters from a string. The effect of specifying precision is summarized in Table 16.3.

**Table 16.3** The effect of specifying precision in `printf` function

Conversion Character	Default precision	Effect of specifying precision as ".n"
d i u o x X	1	At least $n$ digits are printed. If the argument being printed has less than $n$ digits, the output is left-padded with zeros
f e E	6	$n$ digits are printed after the decimal point
g G	Significant digits	At the most, $n$ digits are printed after the decimal point
c	1	No effect
s		At the most, $n$ characters are printed

### Example 16.2 Specifying field width and precision in the printf function

Assume that the variables are initialized as shown below.

```
int a = 12, b = 5;
float f = 1.2345;
```

a) Consider the example given below.

```
printf("%*d\n", a, b);
printf("%*f %*f\n", a, 1.2, b, 1.234);
```

Observe that the field widths in format specifications in these `printf` statements are specified as \*. The actual field width for each occurrence of \* is specified by an argument variable. The first `printf` statement uses value of variable `a` (i. e., 12) as field width to print the value of variable `b`. The second statement prints values 1.2 and 1.234 using values of variables `a` and `b`, respectively, as the field widths. The output of these statements is given below.

```
5
1.200000      1.234000
```

b) Consider another example given below.

```
printf("%*. *f %*. *g\n", a, b, f, a, b, f);
printf("%*. *f\n", 20, 15, f);
```

In this example, the field width as well as precision in format specifications are specified as \*. The first `printf` statement prints the value of variable `f` (1.2345) using formats `*. *f` and `*. *g`. Observe that the field width and precision are specified by values of variables `a` and `b`, respectively. The second `printf` statement prints the value of variable `f` using 20 and 15 as the field width and precision, respectively. The output of these statements is given below.

```
1.23450      1.2345
```

1.234500050544739

We expect that the second `printf` statement will print the value of variable `f` as `1.234500000000000`. However, observe that this is printed as `1.234500050544739`. This is not a printing mistake, rather it is because the floating numbers cannot be represented exactly.

#### 16.4.2 `modf`, `frexp` and `ldexp` Functions

This section discusses three mathematical functions, namely, `modf`, `ldexp` and `frexp`. These functions are summarized in Table 16.4.

**Table 16.4** More functions in the mathematical library of the C language

Function name	Function prototype	Explanation
<code>modf</code>	<code>double modf(double x, double *ipart)</code>	Splits the integer and fractional part of <code>x</code> . Stores the integer part in the variable pointed to by <code>ipart</code> and returns the fractional part.
<code>frexp</code>	<code>double frexp(double x, double *expo)</code>	Splits <code>x</code> into mantissa and exponent. Stores the exponent in the variable pointed to by <code>expo</code> and returns the mantissa.
<code>ldexp</code>	<code>double ldexp(double x, double y)</code>	Returns $x \times 2^y$

The **modf function** is used to split a given floating-point number into integer and fractional parts. The prototype of this function is given below.

```
double modf( double x, double *ipart );
```

Thus, the function accepts two arguments, `x` of type `double` and `ipart` (meaning integer part) of type pointer to `double`. The function stores the integer part of `x` as a double value in the variable pointed to by `ipart` and returns the fractional part. Consider the code given below:

```
double ipart, fpart;  
fpart = modf(1.25, &ipart);
```

In this example, the integer part of 1.25, i. e., 1.0 is stored in variable `ipart` and the fractional part (i. e., 0.25) is returned by the function and is stored in variable `fpart`. Note the use of the address operator in the second argument to this function.

The **frexp function** splits a given floating-point number into mantissa and exponent. The prototype of this function is given below.

```
double frexp( double x, double *expo );
```

The function accepts a floating-point number in argument `x` of type `double`. It stores the exponent part of `x` in the variable pointed to by `expo` and returns the mantissa. Consider the code given below:

```
double mant, expo;  
mant = frexp(1.25, &expo);
```

In this example, the exponent part of 1.25 is stored in variable `expo` and the mantissa of 1.25 is returned by the function and is stored in the variable `mant`.

Finally note that the **ldexp** function which is called `ldexp(x, y)` returns the value of  $x \times 2^y$

#### 16.4.3 Ranges of Integral Data Types

We know that the ranges of various integral data types in C language are implementation dependent. The actual ranges in a particular implementation are defined in the **limits.h header file** as symbolic constants. To make our programs portable, we should use these constants rather than implementation specific values. For example, to refer to the maximum value of an integer number, we should use the constant `INT_MAX` rather than the values 32767 or 2146473647.

The ranges of integral data types as defined in the `limits.h` header file are summarized in Table 16.5. The names of these constants are very easy to remember as they follow a convention in which the type names `char`, `short int`, `int` and `long int` are first referred as `CHAR`, `SHRT`, `INT` and `LONG`, respectively, followed by `"_MIN"` or `"_MAX"`. Thus, the minimum values of the `int` and `short int` types are defined as `INT_MIN` and `SHRT_MIN`, respectively.

**Table 16.5** Constants in `Limits.h` header file that specify the ranges of integral quantities

Data type	Min. value	Max. value
<code>char</code>	<code>CHAR_MIN</code>	<code>CHAR_MAX</code>
<code>short</code>	<code>SHRT_MIN</code>	<code>SHRT_MAX</code>
<code>int</code>	<code>INT_MIN</code>	<code>INT_MAX</code>
<code>long</code>	<code>LONG_MIN</code>	<code>LONG_MAX</code>
<code>signed char</code>	<code>SCHAR_MIN</code>	<code>SCHAR_MAX</code>
<code>unsigned char</code>		<code>UCHAR_MAX</code>
<code>unsigned short</code>		<code>USHRT_MAX</code>
<code>unsigned int</code>		<code>UINT_MAX</code>
<code>unsigned long</code>		<code>ULONG_MAX</code>

The constants for the signed and unsigned data types use the letters 'S' and 'U', respectively, followed by the basic name of the constant, as explained above. For example, the maximum value of `unsigned int` and `unsigned long` data types are defined as `UINT_MAX` and `ULONG_MAX`, respectively. Note

that the `int`, `short int` and `long int` data types are signed by default. Thus, in the case of signed data types, only the constants for `signed char` are defined. Also, in the case of unsigned types, only the constants for maximum values are defined, as the minimum value of any unsigned type is zero.

#### 16.4.4 Traditional Math Constants

Table 16.6 lists various constants usually defined in the `math.h` header file. Note that these constants are not defined in ANSI C. They are defined in Turbo C/C++ as well as the MinGW compiler used in Dev-C++ and Code::Blocks.

**Table 16.6** Traditional mathematical constants defined in the `math.h` header file

Constant	Value	Constant	Value
<code>M_PI</code>	$\pi$	<code>M_SQRT2</code>	$\sqrt{2}$
<code>M_PI_2</code>	$\pi/2$	<code>M_SQRT_2</code>	$1/\sqrt{2}$
<code>M_PI_4</code>	$\pi/4$	<code>M_E</code>	$e$
<code>M_1_PI</code>	$1/\pi$	<code>M_LOG2E</code>	$\log_2 e$
<code>M_2_PI</code>	$2/\pi$	<code>M_LOG10E</code>	$\log_{10} e$
<code>M_1_SQRTPI</code>	$1/\sqrt{\pi}$	<code>M_LN2</code>	$\log_e 2$
<code>M_2_SQRTPI</code>	$2/\sqrt{\pi}$	<code>M_LN10</code>	$\log_e 10$

### 16.5 Advanced Concepts

#### 16.5.1 Bitwise Operators

##### Set, Reset and Complement a Specific Bit

Let us now understand how we can set, reset or complement a specific bit at position  $p \dagger$  in a given number, without affecting other bits in it. For this, we need to perform a specific bitwise operation between the given number and another value called a **mask**.

The mask should usually have the same size in bits as that of the number being operated on. One of two masks will be used depending on the operations to be performed - one mask having 1 bit at position  $p$  and 0 bit at all other positions and an other mask with 0 bit at position  $p$  and 1 bit at all other positions. For example, for operations on bit position 2, we require either 0000 0100 or 1111 1011 as the mask.

These masks can be easily obtained using the bitwise left shift operation on number 1 which has a bit pattern of 0000 0001, i. e., 1 bit at bit position 0 and 0 bit at all other positions. Thus, to obtain a mask

with 1 at bit position  $p$  and 0 at all other positions, we use the expression `1 << p`. We can complement this mask to obtain a mask with 0 at bit position  $p$  and 1 at all other positions using the expression `~(1 << p)`.

Now, to set a bit at position  $p$  in a given number `num`, without affecting other bits in it, we use the bitwise *or* value of `a` with a mask having bit 1 at position  $p$  and 0 at all other positions, as shown in Fig. 16.3, using the following statement.

```
num = num |      (1 << p);
```

Note that the parentheses in the above expression are redundant but they improve readability. Alternatively, we can use the compound assignment operator `|=` as shown below.

```
num |= 1 << p; /* set bit at position p */
```

	<b>Set a bit</b>	<b>Reset a bit</b>	<b>Complement a bit</b>
<code>num</code>	0110 0★10	0110 0★10	0110 0110
<code>operator and mask</code>	0000 0100	& 1111 1011	^ 0000 0100
<code>result</code>	0110 0110	0110 0010	0110 0010

**Fig. 16.3 Set, reset and complement a bit at position 2 without affecting other bits in number num (\* indicates any bit value, either 0 or 1).**

We can use the above mask and the bitwise *xor* operator to complement (toggle) a bit at position  $p$  without affecting the other bits as

```
num ^= 1 << p; /* complement bit at position p */
```

To reset a bit at position  $p$  without affecting the other bits in a given number `num`, we need to perform a bitwise *and* operation between `num` and the mask having 0 bit at position  $p$  and 1 at all other positions using the following statement.

```
num &= ~(1 << p); /* reset bit at position p */
```

## Testing Bits

To test the value of a bit at position  $p$  in number `num`, we check the condition `num & (1<<p)`. If this expression is non-zero, the bit at position  $p$  is set; otherwise it is 0.

We can perform this test on all bits of a given number `num` using a loop as shown below.

```
mask = 1;
for(b = 0; b < 8 * sizeof(num); b++) {
    if ((num & mask) > 0) {
        /* bit b is 0, perform desired operation */
```

```

    }
    else {
        /* bit b is 1, perform desired operation */
    }
    mask <= 1;
}

```

Note that the parentheses in the test for the **if** statement are essential as the **>** operator has a higher precedence than the **&** operator. Alternatively, we can write this test simply as

```
if (num & mask)
```

If the **mask** has same size (in bits) as that of **num**, we can alternatively write the above **for** loop in a concise manner by eliminating loop variable **b** as shown below.

```

for(mask = 1; mask > 0; mask <= 1)      {
    if (num & mask) {
        /* bit b is 0, perform desired operation */
    }
    else {
        /* bit b is 1, perform desired operation */
    }
}

```

Sometimes we may wish to process the bits in a given number from left to right. For this, we need to take a mask with a 1 bit in the MS bit position and 0 at all other bit positions. This can be achieved by shifting number 1 left as shown below.

```
mask = 1 << (8 * sizeof(mask) - 1)
```

Note that to shift this bit right one bit position at a time, we have to use a logical right shift operation in which the empty MS bit positions created by the right shift are set to 0. Hence, **mask** should be declared as an **unsigned** variable.

### Example 16.3 Test the bits in a number

- a) Count the number of zeros and ones in a binary representation of a number

A program to count zeros and ones in a binary representation of a number is given below.

```
#include <stdio.h>

int main()
{
```

```

int num;
int zeros, ones;
int b, mask;

printf("Enter an integer number: ");
scanf("%d", &num);

ones = 0;
mask = 1;
for(b = 0; b < 8 * sizeof(num); b++) {
    if (num & mask)
        ones++;
    mask <= 1;
}
zeros = 8 * sizeof(num) - ones;

printf("Zeros: %d Ones: %d\n", zeros, ones);
return 0;
}

```

We can also write a function `count_1bits` to count the number of ones in the binary representation of a number as shown below.

```

int count_1bits(int num)
{
    int ones, mask;

    ones = 0;
    for(mask = 1; mask < 0; mask <= 1) {
        if (num & mask)
            ones++;
    }
    return ones;
}

```

### b) Print a binary representation of a number

To print the binary representation of an integer number, we have to separate the digits starting from the MS bit. This requires a mask of type `unsigned int`. A function to print a binary representation of a given integer number is given below.

```

void print_bin(int num)
{

```

```

    unsigned int mask;
    for(mask = 1 << (8 * sizeof(num) - 1); mask > 0; mask >>= 1)
        printf("%d", num & mask? 1 : 0);
    printf("\n");
}

```

### 16.5.2 Renaming Types with `typedef`

In this section, we see some complex `typedefs`. Recall the rule for a `typedef` from Section 16.3: *To declare a new type name for a particular type, first declare this name as if we were declaring a variable of that type and then precede that declaration with `typedef` keyword.*

#### Declaring Array Type Names Using `typedef`

Consider that we wish to declare a new type name, say `IntArr10`, for an integer array having 10 elements. Using our rule for declaration of a `typedef`, we first declare this name as if it is an integer array having 10 elements as

```
int IntArr10[10];
```

and then insert `typedef` keyword before this declaration to obtain the desired `typedef` as

```
typedef int IntArr10[10];
```

Now we can use this new type name to declare arrays instead of the array declaration syntax as shown below:

```
IntArr10 a;
IntArr10 primes = {2, 3, 5, 7, 11, 13};
```

These declarations are equivalent to the following declarations using the array declaration syntax.

```
int a[10];
int primes[10] = {2, 3, 5, 7, 11, 13};
```

Using `IntArr10`, we can write a function that accepts an `IntArr10` parameter and prints that array as

```
void intarr_print(IntArr10 a, int n)
{
    int i;
    for(i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
```

```
}
```

Note that array elements are accessed within this function using the usual syntax: `a[i]`. Also note that since `IntArr10` is just another name for a type *integer array having 10 elements*, we can use this function to accept arrays declared using `IntArr10` as well as the arrays declared using the usual array syntax. In fact, as the array size in a parameter array is ignored, this function can work with integer arrays having any number of elements. That is why we named the function `intarr_print` instead of `intarr10_print`.

We can declare an array of type `IntArr10` and initialize it as shown below.

```
IntArr10 c[10] = {{1, 2, 3}, {4, 5, 6}};
```

Note that `C` is an array having 10 elements, each of which is again an array having 10 elements. Thus, `C` is a two-dimensional array. The *ij*'th element of this array is represented as `a[i][j]`. We can process this array using nested for loops as illustrated below in function `intarrarr_print`.

```
void intarrarr_print(IntArr10 a[], int m, int n)
{
    int i, j;
    for(i = 0; i < m; i++) {
        for(j = 0; j < n; j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }
}
```

### Declaring Pointer Type Names Using `typedef`

We can also declare a type name for a pointer to `int` type as

```
typedef int *IntPtr;
```

Observe that the code after the `typedef` keyword is how we would have declared a pointer `IntPtr` to an `int` type. Now we can use `IntPtr` to declare pointer variables. For example,

```
int a = 10, b = 20;
IntPtr pa = &a;
IntPtr pb;
scanf("%d", pb);
printf("%d %d", *pa, *pb);
```

Note that we should not use the `*` operator when a pointer is declared using the `IntPtr` type name. Pointer variables `pa` and `pb` can be used as usual. Also note that the use of `typedef` does not

completely eliminate the \* operators in the code. Using this new type name, we can write a function to swap two values as shown below.

```
void swap(IntPtr pa, IntPtr pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
```

### Declaring Function Type Names Using **typedef**

The **typedef** feature can be used to declare a function type name as well. Consider, for example, the typecast used in the `qsort` function call of Section 14.3.3.

```
qsort(a, n, sizeof(a[0]), (int (*)(const void*, const void*)) icmp);
```

Here, the comparison function `icmp` is typecast to a pointer to a function that takes two arguments of type `const void *`. This cast is quite complicated to understand, particularly for a beginner. To simplify this statement, we can use **typedef** as shown below.

```
typedef int (*fcmp)(const void *, const void *);
```

Now the above call to `qsort` function can be simplified as shown below.

```
qsort(a, n, sizeof(a[0]), (fcmp) icmp);
```

### Declaring **typedef** Names from Other **typedef** Names

We can declare a new type name from an already declared type name. Consider the example given below.

```
typedef unsigned short int UShort;
typedef UShort *UShortPtr;
typedef UShort UShortArr10[10];
```

In this example, `UShort` is first declared as a new type name for `unsigned short int` and is then used to declare two type names, `UShortPtr` as a pointer to `UShort` and `UShortArr10` as an array of `UShort` having 10 elements.

### 16.5.3 Unions

As we already know, a structure is an ordered list of elements of either the same or different types. A **union** is similar to a structure but with one major difference that the structure stores all its members

one after another, whereas the union can store only one member at a time, since all the members in a union are stored beginning from the same memory location. Thus, the total memory required to store a union is the same as that required for the largest member in it. The union is useful when we have to store one of the alternative values of different types and yet conserve space.

Consider the union **data** given below.

```
union number {  
    char c;  
    int i;  
    float f;  
};
```

This union allows us to store either a **char**, **int** or **float** number. We can now declare a variable **x** of type union **number** as shown below.

```
union number x;
```

We can access members of this variable using the *dot* operator as **x.c**, **x.i** and **x.f**. Note that if we store a value in a particular member, we should read the value through that member only; otherwise, the results will be unpredictable. Hence, we often use another variable to remember the type of value stored in a union. For example, we can use a char variable, say **type**, which can have one of the three values '**'c'**', '**'i'**' or '**'f'**', for the three data types, **char**, **int** and **float**, respectively. We can also use an enumerate type for this purpose.

It is also a good idea to wrap the union variable and the accompanying type variable for storing the type information in a structure as shown below.

```
struct data {  
    char type;  
    union number val;  
};
```

Alternatively we can include the definition of union **number** in the structure **data** as shown below.

```
struct data {  
    char type;  
    union number {  
        char c;  
        int i;  
        float f;  
    } val;  
};
```

The **main** function given below illustrates the operations on a union. It first reads a value in variable **x**

of type **data** and then prints it.

```
int main()
{
    struct data x;
    printf("Enter type of value (c, i, f): ");
    scanf("%c", &x.type);
    fflush(stdin);

    printf("Enter value: ");
    switch(x.type) {
        case 'c': scanf("%c", &x.val.c); break;
        case 'i': scanf("%d", &x.val.i); break;
        case 'f': scanf("%f", &x.val.f); break;
        default: printf("Error: Invalid number
type ...\\n"); break;
    }

    switch(x.type) {
        case 'c': printf("%c", x.val.c); break;
        case 'i': printf("%d", x.val.i); break;
        case 'f': printf("%f", x.val.f); break;
        default: printf("Error: Invalid number
type ...\\n"); break;
    }
    return 0;
}
```

#### 16.5.4 Writing Multi-File Programs

The programs we have studied so far have the entire C code written in a single source (.c) file. This is fine for small programs. However, as the program size increases, which is typical for most real-world applications, it is much more convenient to divide the program into several source files. Such programs are usually created and managed using the **project** facility provided in the development environments or by using a **make** utility. During the *build* process, in which the program files are compiled to obtain executable file, the compiler will usually compile only those files that have changed since the last build operation. Thus, besides being convenient, the multi-file approach also speeds up the program development.

To create a **multi-file project**, we first logically separate the entire program into several .c files, each having a related functionality. A .c file usually contains function definitions. It will also contain related declarations (**#define**, **typedef**, **enum**, **struct**, **union**, function prototypes, etc.) which are not accessed in other program files. If some functionality is accessed in another .c or .h file, its

declaration or definition is moved to the header file. Thus, the external (or global) declarations are normally written in the header files.

We can use two approaches for writing header files. In the first approach, all the declarations are written in a single header file, whereas in second approach, a separate header file is written for each .c file.

If a functionality from one header file, say "myheader.h", is used in some .c file, that header file is included in the .c file as

```
#include "myheader.h"
```

Note that the header file name is enclosed in double quotes.

A header file is likely to be included in several .c files. If such a file contains `#define`, `enum`, `struct` and `union` declarations, the compiler will report errors as these entities will be encountered multiple times. Hence, the contents of a header file should be included exactly once in a project irrespective of how many times it is actually included. To achieve this, we include a **header guard** in each header file as shown below, for `myheader.h` header file:

```
#ifndef MYHEADER_H_INCLUDED  
#define MYHEADER_H_INCLUDED  
    include the contents of header file here  
#endif
```

We must replace the constant `MYHEADER_H_INCLUDED` with the appropriate name for our header file. When this header file is included for the first time, the constant `MYHEADER_H_INCLUDED` is not defined, and the subsequent statements are read until a `#endif` statement is encountered. Thus, the constant `MYHEADER_H_INCLUDED` is now defined (on the second line) and the contents of the header file have been included in the project. Now if this file is included for the second time in any file in the project, the constant `MYHEADER_H_INCLUDED` is already defined. Thus, the test in `#ifndef` evaluates as false and the subsequent statements are skipped until `#endif` statement is encountered. Thus, the contents of header file are effectively included only once.

#### Example 16.4 Multi-file program to illustrate I/O operations on vectors and matrices

Assume that we have to develop a program to illustrate I/O operations on vectors and matrices. Let us write a multi-file program with a single header file as shown below:

`arrdemo.c` : main project file containing the `main` function

`vector.c` : source file containing definitions of functions for operations with vectors

`matrix.c` : source file containing definition of functions for operations with matrices

`arrdemo.h` : header file containing prototypes of functions from `vector.c` and `matrix.c` files

Let us write two functions for vector I/O, namely, `vec_read` and `vec_print`, and two functions for matrix I/O, namely, `mat_read` and `mat_print`. The contents of `vector.c` and `matrix.c` file are given below (the function definitions are omitted to save space).

`vector.c` file:  
`#include <stdio.h>`

```
void vec_read(int a[], int n)
{
    ...
}
```

```
void vec_print(int a[], int n)
{
    ...
}
```

`matrix.c` file:  
`#include <stdio.h>`

```
void mat_read(int a[][10], int m, int n)
{
    ...
}
```

```
void mat_print(int a[][10], int m, int n)
{
    ...
}
```

The `arrdemo.h` file will contain the declarations of these functions from `vector.c` and `matrix.c` file as shown below along with the header guard.

`arrdemo.h` file:

```
#ifndef ARRDEMO_H_INCLUDED
#define ARRDEMO_H_INCLUDED
void vec_read(int a[], int n);
void vec_print(int a[], int n);
void mat_read(int a[][10], int m, int n);
void mat_print(int a[][10], int m, int n);
#endif
```

Finally, the `arrdemo.c` file will contain the `main` function as shown below.

`arrdemo.c` file:

```
#include <stdio.h>
#include "arrdemo.h"
int main()
{
    ...
}
```

Note that the `arrdemo.h` header file has been included here as it contains the declarations of functions for vector and matrix I/O.

If the project is much bigger, multiple header file approach is more suitable. In this approach, the contents of the `vector.c` and `matrix.c` files remain the same. However, now we have two header files, namely, `vector.h` and `matrix.h`, which will contain the declarations of functions from respective source files along with the header guard as shown below.

**vector.h file:**

```
#ifndef VECTOR_H_INCLUDED
#define VECTOR_H_INCLUDED
void vec_read(int a[], int n);
void vec_print(int a[], int n);
#endif
```

**matrix.h file:**

```
#ifndef MATRIX_H_INCLUDED
#define MATRIX_H_INCLUDED
void mat_read(int a[][10], int m, int n);
void mat_print(int a[][10], int m, int n);
#endif
```

Finally, the `arrdemo.c` file will contain the `main` function as shown below:

**arrdemo.c file:**

```
#include <stdio.h>
#include "vector.h"
#include "matrix.h"
int main()
{
    ...
}
```

Observe that both the header files, `vector.h` and `matrix.h` have now been included in the `arrdemo.c` file.

**Example 16.5** Working with projects in Code::Blocks

Let us now study how to create, build and execute a project in Code::Blocks. This discussion is useful even if you are using some other development environment such as Dev-C++, NetBeans, Eclipse, etc. We will create project `arrdemo` discussed in the previous example.

First, let us create the project `arrdemo.cbp` (`cbp` stands for Code::Blocks project). Start Code::Blocks and select the *File→New→Project...* menu option. In the dialog box displayed, select *Console application* and press the *Go* button. In the *Console application* dialog box, select C language and press the *Next* button. Type the project name (`arrdemo`) in the next dialog box displayed, select the folder where the project is to be created and then press the *Next* button. In the next dialog box, you can select the *Debug* as well as the *Release* configurations and then end up with choosing the *Finish* button. The project is now created and its entry displayed in the *Projects* pane in the *Manager* window (if not visible, this window can be displayed using *View→Manager* option or by pressing *Shift-F2*).

Now we have to add files to the project. For each file to be added, select the *File→New→File...* menu option. In the dialog box displayed, select either *C/C++ source* or *C/C++ header* option as is appropriate, and then press the *Go* button. Follow the instructions in the subsequent dialog boxes (select C language, provide file name and path and add the file to the desired project configurations and finally press the *Finish* button). The file is added to the project, as can be seen in the Manager (under under the *Sources* or *Headers* node in the project tree). Note that a header guard can be automatically generated while adding a header file to a project. Now you can add the required code to these files. Note that when the project was created, a file `main.c` containing the `main` function was also created. We can continue to use this file or delete it and add a new file named `arrdemo.c` as per the original design.

However, if we have the required files ready, we can add them to the project. To add one or more existing files to a project, select *Project→Add Files...* menu option and select the desired files.

Once all the program files are created as required, we can build the program using the *Build→Build and run* menu option (or by pressing *F9*). The errors, if any, should be corrected after which the program will be executed. Finally, note that you should save the project intermittently using *File→Save Project* menu option.

## Exercises

1. Declare the enumerated types specified below (using `enum`) along with suitable enumerated constants for the specified values. Also declare specified variables of this enumerated type
  - a. Operating Systems (MS Windows, Linus, Sun Solaris, Mac OS X); variable `my_os`
  - b. Gender (Male, Female); variable `gender`
  - c. Car manufacturers (Mercedes, BMW, Toyota, Hyundai, Honda); variable `my_car_make`
  - d. Directions (East, West, South, North); variable `dir`
2. Rewrite the answers in Question 1 using `typedef`.

3. Define new types and also declare variables of that type.
  - a. Type **UCHAR** to store unsigned characters; variable **ch1** and **ch2** of type **UCHAR**
  - b. Type **SLONG** to store signed long integer numbers; variable **x** of type **SLONG**
4. Identify the errors, if any, in the declarations given below.
  - a. **#define usint unsigned short int**
  - b. **typedef USHORT = unsigned short;**
  - c. **typedef SLONG signed long int;**
  - d. **typedef long float double;**
  - e. **enum Direction {"Up", "Down", "Left", "Right"};**
  - f. **enum Shape = {Triangle, Square, Rectangle, Circle, Ellipse};**
  - g. **enum taste {sweet, sour, salty, spicy};**
  - h. **typedef enum Lang {Hindi, Marathi, English, French, German};**
  - i. **typedef enum {Apple, Orange, Grape, Banana, Watermelon} Fruit;**
5. State whether the following statements are true or false. Also, rewrite the false statements correctly.
  - a. The **typedef** keyword is used to define new data types in our programs.
  - b. The use of enumerated data types improves the readability of a program.
  - c. The user-defined data types can be used to define new data types that can be used to represent any desired range of values.
  - d. The enumerated constants can be used to define names for floating type constants.

## Exercises (Advanced Concepts)

1. Write functions for the following operations.
  - a. Set a specified bit in a given unsigned integer number.
  - b. Convert a decimal number to a packed BCD (binary coded decimal) representation.
  - c. Convert a packed BCD number to its decimal equivalent.
2. Define new types and also declare variables of that type.
  - a. Type **IntPtrPtr** as a pointer to pointer to **int**; variables **p1** and **p2** of type **IntPtrPtr**
  - b. Type **ULongArr20** as an array having 20 elements of type **unsigned long int**
  - c. Declare type **IntPtr** as a pointer to **int** and using this type declare another pointer named

`intPtrArr` as an array of pointers having 10 elements to `int`

3. Explain the similarities and differences between a structure and a union.
4. Using your favourite IDE, develope a multi-file program for performing I/O and other operations on vectors and matrices as explained in Section 16.5.4.

# 17 Graphics in Turbo C and Turbo C++

In this chapter, we will study the graphics capabilities provided in TC and TC++<sup>1</sup> (referred to simply as TC in the rest of this chapter). We first present the basics of the graphics mode and explain how to invoke the graphics system. Next, we learn how to draw basic graphic entities such as points, lines, rectangles, polygons, circles and ellipses. This chapter also explains how filled entities can be drawn. The next section discusses how to display text. Finally, the techniques for animation, i. e., object motion on the screen, are presented.

Note that the features discussed in this chapter are specific to TC and are not available in other development environments. However, studying them is useful as several basic concepts used here are applicable in other development environments. Also note that unlike the text mode, the graphics mode relies on machine hardware (graphics card and monitor). Hence, a program written for one machine may work differently (or may not work at all) on another machine.

## 17.1 Preliminaries

### 17.1.1 Capabilities of the Graphics Mode

In the graphics mode, we have the ability to manipulate individual dots on the screen. This enables us to display almost anything on the screen. This includes entities such as lines, rectangles, polygons, circles, arcs, ellipses, irregular shapes, charts, plots of equations, etc. These entities can be drawn in various colours and be filled in different colours and patterns. The text can be displayed in various ways (font, style, size, orientation and colour). In addition, we can create animation, i. e., movement of objects on the screen.

### 17.1.2 Graphics Support in TC/TC++

TC/TC++ provide a powerful library for graphics operations. It includes several functions, enumerated constants and structures. Their declarations are provided in the `<graphics.h>` header file. Hence, each graphics program must include it as:

```
#include <graphics.h>
```

The functions provided in the graphic library of TC can be grouped into following broad categories: *invoke graphics system* (initialize and close the graphics system), *setting graphics system* (set colours, line styles, fill styles, etc.), *entity drawing* (drawing points, lines, rectangles, polygons, circles, arcs, ellipses,

etc.), *text display*, *inquire graphics system*, *animation* and *error handling*.

### 17.1.3 Pixels and Resolution

In the graphics mode, the screen is considered to be a large number of closely spaced dots arranged in rows and columns. Each dot is addressed independently. Such an addressable dot on the screen is called a **pixel** (picture element) or **pel**. Note that the horizontal and vertical spacing between the pixels may be different.

On a monochrome (black and white) monitor, each pixel can be turned on or off. Thus, only two colours are available. However, on a colour monitor, a number of colours are available depending on the type of the monitor.

Graphics comprises two parts: *foreground* and *background*. The foreground includes the pixels that form the entities such as lines, arcs, circles, irregular shapes, images, filled regions, etc. The other pixels constitute the background.

The number of pixels available on the monitor's screen is referred to as the **screen resolution**. It is specified in terms of total pixels in the  $x$  and  $y$  directions, as in  $640 \times 480$ ,  $1024 \times 768$ , etc. As the resolution increases, the pixel size and inter-pixel distance decrease, enabling us to draw sharper drawings. Note that the resolution specified for a monitor or a graphics adapter is the maximum resolution. They usually support one or more lower resolutions, providing more colours.

### 17.1.4 Colours and Palettes

In TC, the colours are numbered from 0. The `<graphics.h;>` header file defines several enumerated colour constants shown in Table 17.1. You can remember the colours 0 to 7 using the string `BBGC RMBL`. Note that the colours 8–15 are lighter versions of colours 0–7.

Each pixel may be displayed in one of the colours from a group of colours, often called the **palette**. Thus, the palette determines the colours that can be simultaneously displayed on the screen. A palette is usually a subset of available colours. The maximum available colours, palette size and actual colours available in a palette depend on the graphics adapter and the graphics mode used. For the VGA mode, which is assumed throughout this chapter, the palette provides 16 colours, given in Table 17.1.

**Table 17.1** TC colour constants and the VGA palette

Colour constant	Value
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

### 17.1.5 Graphics View Ports and Pages

The output of graphics functions can be confined to a **view port**, a rectangular region on the screen, without disturbing the remaining screen. This allows us to use the remaining screen to display other information such as instructions to the user, help pages, etc. When the graphics system is initialized, the entire screen becomes the view port. We can clear the contents of a view port using the `clearviewport` function.

Several display adapters support multiple **graphic pages**. This is especially useful in animation. While one page is being displayed, the other pages can be prepared for display. One of these pages can then be displayed instantaneously. For example, the VGA adapter provides one page at a resolution of  $640 \times 480$  pixels, two pages at a resolution of  $640 \times 350$  pixels and four pages at a resolution of  $640 \times 200$  pixels. Unless otherwise mentioned, the VGA adapter is assumed throughout this chapter.

### 17.1.6 Graphics Adapters, Drivers and Modes

A **graphics adapter** is an electronic circuit that handles the display of information (text and graphics) on the screen. Note that as TC is a very old DOS-based IDE, it does not support the advanced adapters available in recent PCs. Some of the adapters supported by TC include CGA ( $640 \times 200$ , 16 colours), EGA ( $640 \times 350$ , 16 colours) and VGA ( $640 \times 480$ , 256 colours). The adapters in modern PCs can also emulate these adapters.

Each video adapter handles graphics differently. Hence, TC provides **drivers** for various video adapters in files having a `.BGI` extensions, e. g., drivers for EGA and VGA adapters are provided in the `EGAVGA.BGI` file. The driver files are usually located in the `\TC` or `\TC\BGI` directory.

The drivers supported by TC are numbered from zero. The `<graphics.h>` header file provides enumerated constants for drivers, e. g., CGA, EGA and VGA with values 1, 4 and 9, respectively. Although we can use these constants to initialize the graphics system in a specific mode, it is better practice to detect the available graphics hardware and use the best mode supported. TC provides a constant `DETECT` with value 0 to perform such autodetection.

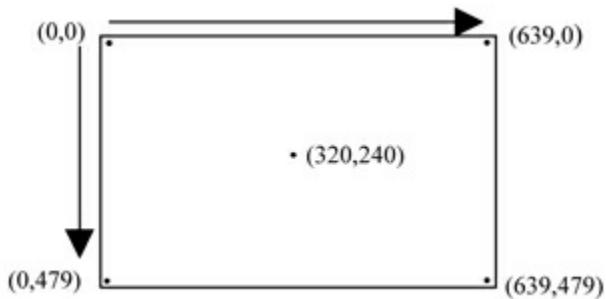
The combination of a particular resolution and a particular colour palette is called a **graphics mode**. A display adapter usually supports various modes. These modes differ in resolution, colours and pages

available. We can use any of these graphics modes at a time. The use of a high-resolution mode allows us to produce sharper graphics, whereas we usually get more colours and pages while working with a lower resolution mode.

The graphics modes available for a graphics driver are numbered from zero. The `<graphics.h>` header file provides the enumerated names for these graphics modes. For example, the modes available for VGA driver are **VGAL0** ( $640 \times 200$  resolution), **VGAMED** ( $640 \times 350$ ) and **VGAHI** ( $640 \times 480$ ).

### 17.1.7 Graphics Coordinate System and CP

In the TC graphics **coordinate system** shown in Fig. 17.1, the pixels are numbered from zero and each pixel is specified by a pair of  $x$  and  $y$  coordinates as  $(x, y)$ . Observe that this coordinate system is quite different from the Cartesian coordinate system: the origin  $(0, 0)$  is located at the top-left corner of the screen and the  $y$  coordinates increase downward. Also, the screen coordinates are non-negative integers and are limited by the resolution of the graphics mode.



**Fig. 17.1** TC screen coordinate system in VGA graphics mode

Fig. 17.1 also shows the coordinates of the corner pixels and the centre pixel in the VGA mode having  $640 \times 480$  resolution. It may be noted that in most graphics functions requiring screen coordinates, we can specify a point outside the screen. TC provides two functions, `getmaxx` and `getmaxy`, to determine the maximum value of the  $x$  and  $y$  coordinate, respectively.

The graphics system supports a notion of **CP**, the **current position**. It is similar to the text mode cursor except that it is not visible. Some drawing functions display the output at the CP. However, most drawing functions directly specify the position at which the output is to be displayed. Note that most graphics drawing functions affect the CP, usually moving it to the last point specified in the function call. TC also provides functions to manipulate the CP directly.

## 17.2 Invoking the Graphics System

By default, the output screen in TC is in the text mode. Before we perform any graphics operations, we must switch to the graphics mode using the **initgraph** function. Similarly, when work with the

graphics mode is over, we close the graphics system using the **closegraph** function. TC also provides the **detectgraph** function to detect the graphics hardware in our machine, i. e., driver and mode to be used.

A call to the **initgraph** function takes the following form:

```
initgraph(&gd, &gm, "c:\\TC\\BGI");
```

where **gd** and **gm** are **int** variables which specify the graphics driver and graphics mode to be used. Before calling the **initgraph** function, we should initialize these variables with the required enumerated constants (discussed earlier). Note that if we specify **DETECT** as the value of **gd**, TC detects the best graphics mode available for use (using the **detectgraph** function) and initializes the graphics system in that mode. The last function parameter specifies the path of the directory containing the driver file, e. g., **EGAVGA.BGI** for the VGA driver.

When the graphics system is initialized, it resets all settings (such as CP, palette, colours, view port, etc.) to their default values. Thus, the screen is cleared, CP is moved to top-left corner, i. e., (0, 0), the entire screen is set as the view port and the default values are initialized for various settings such as drawing colour, background colour, line style, fill style, etc.

We use the **closegraph** function to close the graphics system and return to the text mode. When this function is called, the text mode is initialized and the screen is cleared.

#### Example 17.1 Automatically detect and initialize the graphics system

The skeleton of a typical graphics program is given below. It uses the auto-detect capability of the **initgraph** function to detect the best graphics driver and mode available on the system and initializes the graphics system. Note that the graphics commands should be written between the **initgraph** and **closegraph** functions. A **getch** function is usually used before the **closegraph** function to wait for a key press before closing the graphics system and returning to the TC IDE.

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

int main()
{
    int gd = DETECT, gm; /* graph driver and graph mode */
    initgraph(&gd, &gm, "C:\\TC\\BGI");

    /* add statements to draw graphics */

    getch();
```

```
    closegraph();
}
```

## 17.3 Setting Colours and the Current Position (CP)

TC provides the **setcolor** and **setbkcolor** functions to set the foreground colour (i. e., current drawing colour) and background colour, respectively. Their prototypes are given below.

```
void setcolor(int color);
void setbkcolor(int color);
```

where **color** is either a colour number or name as given in Table 17.1. For example, we can set the foreground colour to green using **setcolor(GREEN)**.

Note that we can use only one colour at a time as the background colour for the entire graphics screen, unlike the text mode in which we can simultaneously use more than one background colour. We can change the background colour any time during the execution of a program without affecting the foreground graphics. The default background colour is **BLACK**.

The colour set by the **setcolor** function is used by subsequent functions that draw various objects such as lines, rectangles, circles, arcs, ellipses, etc. The VGA mode provides a palette with 16 colours as shown in Table 17.1.

The drawing colour can be changed by another call to the **setcolor** function. The default drawing colour is the last colour in the palette, which is **WHITE** for the VGA mode. Note however that we can select a colour from the current palette only. Some graphics modes, such as CGAC0, provide only four colours in the palette. The maximum value of the colours available in a palette can be obtained using the **getmaxcolor** function.

TC provides two functions to set the current position (CP): **moveto** and **moverel**. The **moveto** function moves CP to a specified pixel  $(x, y)$ . Its prototype is given below:

```
void moveto(int x, int y);
```

Note that unlike in the text mode, we can move the CP outside the screen. For example, the function call **moveto(-10, -10)** moves the CP to  $(-10, -10)$  which is outside the screen.

The **moverel** function moves the CP a relative distance from its current position. Its prototype is given below.

```
void moverel(int dx, int dy);
```

If the CP is at  $(x, y)$ , this function will move it to  $(x+dx, y+dy)$ . The values of  $dx$  and  $dy$  may be either positive or negative. Thus, assuming the CP at  $(100, 100)$ , the call **moverel(-20, -10)** moves the CP to  $(80, 90)$ .

## 17.4 Drawing Graphics Entities

TC provides several functions to draw graphics entities such as a point, line, rectangle, polygon, circle, ellipse, arc, pie slice, etc. These objects can be drawn in various colours. We can also draw filled shapes or fill a region with a specified colour and fill style. The functions available in this category are summarized in Table 17.2.

By default, these functions draw graphic entities using the current foreground colour, which is **WHITE** by default in VGA mode. Note that we can change the foreground colour before drawing any graphic entity using the **setcolor** function.

**Table 17.2** *TC functions for drawing graphics entities*

Object	Function	Purpose
Point	<b>putpixel</b>	Draws a pixel in specified colour
Line	<b>line</b>	Draws a line between two specified points
	<b>lineto</b>	Draws a line from CP (current position) to a specified point
	<b>linerel</b>	Draws a line from CP to a point specified relative to CP
Rectangle	<b>rectangle</b>	Draws a rectangle whose top-left and bottom-right corners are specified
	<b>bar</b>	Draws a bar, i. e., a filled rectangle
	<b>bar3d</b>	Draws a three-dimensional bar
Polygon	<b>drawpoly</b>	Draws a polygon, i. e., a sequence of straight lines
	<b>fillpoly</b>	Draws a polygon and fills it
Circle	<b>circle</b>	Draws a circle
	<b>arc</b>	Draws an arc
	<b>pieslice</b>	Draws a pie slice
Ellipse	<b>ellipse</b>	Draws an ellipse
	<b>fillellipse</b>	Draws a filled ellipse
	<b>sector</b>	Draws a sector
Region	<b>floodfill</b>	Fills a region using the current fill style

### 17.4.1 Drawing Points and Lines

TC provides the **putpixel** function to draw a pixel. Its prototype is given below.

```
void putpixel(int x, int y);
```

TC provides three functions to draw lines: **line**, **lineto** and **linerel**. The prototypes for these functions are given below.

```
void line(int x1, int y1, int x2, int y2);
```

```
void lineto(int x, int y);
void linerel(int dx, int dy);
```

The **line** function is used to draw a line between two specified pixels ( $x_1, y_1$ ) and ( $x_2, y_2$ ), whereas the **lineto** and **linerel** functions draw a line from the current position (CP) to a point specified using either absolute and relative coordinates, respectively.

In the case of the **lineto** function, the parameters  $x$  and  $y$  are the coordinates of the endpoint of the line. The **linerel** function draws a line a relative distance from the current position as specified by  $(dx, dy)$ . Thus, if the coordinates of the current position are  $(x, y)$ , the coordinates of end point of the line are obtained as  $(x + dx, y + dy)$ .

These functions use the current settings for the foreground (i. e., drawing) colour and line style. We can change the line style using the **setlinestyle** function. The default line style is a 1-pixel-wide continuous line.

The **lineto** and **linerel** functions (but not the **line** function) modify the current position (CP) such that the end point of the line becomes the new CP. As a result, these functions are very convenient to draw connected line segments and in conjunction with the **moveto** and **moverel** functions, they can be used to simplify the drawing of complex shapes.

Finally, note that the line drawing functions may refer to a point outside the screen. In such a case, the line is clipped at the screen boundary, which is the default view port.

### Example 17.2 Draw a triangle

Let us draw a triangle at the centre of the screen with the coordinates of three vertices as A(320,200), B(360,280) and C(280,280). Assume that the graphics system has been initialized in the **VGA** mode.

The program segment given below uses the **line** function to draw the desired triangle. We require three calls to this function, one for each line segment AB, AC and BC.

```
line(280,280, 360, 280); /* line AB */
line(280,280, 320, 200); /* line AC */
line(360,280, 320, 200); /* line BC */
```

Since we have not set the colours, the default colours are used: black background and white foreground. Alternatively, we can rewrite this code segment using the **moveto** and **lineto** functions as shown below:

```
moveto(280, 280);          /* point A is CP */
lineto(360, 280);          /* draw line AB, B is new CP */
lineto(320, 200);          /* draw line BC, C is new CP */
lineto(280, 280);          /* draw line CA */
```

We first move to point A(280, 280) using the **moveto** function and then use the **lineto** function to draw the three line segments in a sequence. Note that we could have drawn these line segments in

another sequence: AC, CB and BA.

We can also use the relative line drawing function `lineref` to achieve the same result as

```
moveto(280, 280); /* point A is CP */
lineref(80, 0); /* draw line AB, B is new CP */
lineref(-40, -80); /* draw line BC, C is new CP */
lineref(-40, 80); /* draw line CA */
```

This relative drawing approach has an advantage in that we can change starting point (in the `moveto` function) to draw the object at that position.

### 17.4.2 Drawing Rectangles and Polygons

Although we can draw a rectangle or a polygon using several line drawing commands, it is more convenient to directly use the functions provided by TC to draw these shapes.

TC provides the **rectangle function** to draw a rectangle. Its prototypes is as follows:

```
void rectangle(int left, int top, int right, int bottom);
```

This function accepts the coordinates of top-left and bottom-right corners of a rectangle and draws it using the current drawing colour and line style. Note that the `rectangle` function does not affect the CP. For example, we can draw the largest rectangle on the VGA screen as

```
rectangle(0, 0, 639, 479);
```

TC also provides the **drawpoly function** to draw a polygon (i. e., a sequence of connected straight lines). Its prototype is given below.

```
void drawpoly(int numpoints, int *polypoints);
```

The argument `numpoints` specifies the number of points on the polygon and the argument `polypoints` is a pointer to a sequence of `numpoints` × 2 integer numbers. Each pair of integer numbers specifies the *x* and *y* coordinates of a point on the polygon.

As with other entity drawing functions, the `drawpoly` function draws a polygon using the current drawing colour and line style. Note that to draw a close polygon having *n* vertices, we pass *n*+1 points to `drawpoly`, where the last point is the same as that of the first. For example, we can draw a triangle in the previous example by initializing an array `triangle`(at the beginning of the `main` function) with its coordinates and call the `drawpoly` function as shown below.

```
int triangle[] = {280, 280, 360, 280, 320, 200, 280, 280};
drawpoly(4, triangle);
```

### 17.4.3 Drawing Circular and Elliptical Shapes

TC provides three functions to draw circular and elliptical shapes: **`circie`** to draw a complete circle, **`arc`** to draw a circular arc and **`ellipse`** to draw a complete ellipse or an elliptical arc. These functions draw shapes using a solid line and current drawing colour and line thickness. The prototypes of these functions are given below.

```
void circie(int x, int y, int radius);
void arc(int x, int y, int stangle, int endangle, int radius);
void ellipse(int x, int y, int stangle, int endangle, int xradius,
             int yradius );
```

The arguments in these functions are as follows:

<i>x, y</i>	<i>x-</i> and <i>y</i> -coordinates of the object centre
<i>radius</i>	radius of circle or arc (in pixels)
<i>xradius, yradius</i>	radii of an ellipse in <i>x</i> - and <i>y</i> -direction (in pixels)
<i>stangle, endangle</i>	starting and ending angle (in degrees) for arc or ellipse.

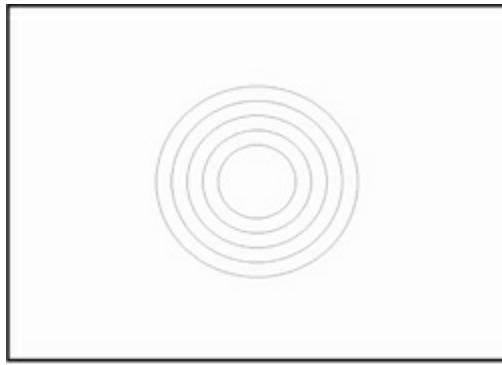
An arc or an ellipse is drawn in counterclockwise direction from *stangle* to *endangle*. These angles are measured from the x axis in the counterclockwise direction. Their values can be negative as well as greater than 360.

The **`ellipse`** function has a limitation that it can draw elliptical shapes in either horizontal or vertical orientation. TC does not provide a function to draw an inclined (or rotated) ellipse. Note that if the circles are not perfectly round, we can use the **`setaspectratio`** function to adjust the aspect ratio.

### Example 17.3 Draw concentric circles and circular and elliptical arcs

The program segment given below draws five concentric circles at the centre of the VGA screen. using a *for* loop with radii 50, 70, 90, 110 and 130, as shown in Fig. 17.2.

```
for (c = 0; c < 5; c++)
    circle (320, 240, 50 + c * 20);
```

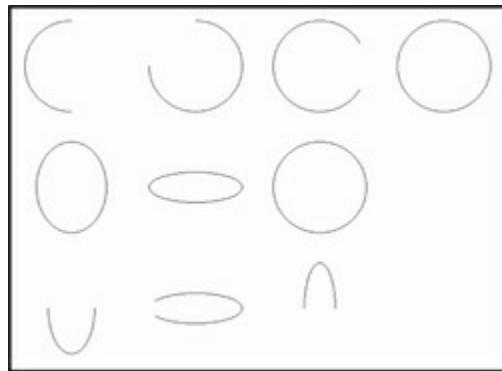


**Fig. 17.2**

We can draw each circle in a different colour by setting a different drawing colour before each circle is drawn, as shown below. Note that as colour 0 is the background colour, we have used colours starting from 1 (using `c + 1` as the colour value in the `setcolor` function).

```
for (c = 0; c < 5; c++) {  
    setcolor(c + 1);  
    circle (320, 240, 50 + c * 20);  
}
```

The program segment given below draws several circular and elliptical arcs as shown in Fig. 17.3. The arc functions draw four arcs. Observe how the shapes are drawn in a counterclockwise direction from *stangle* to *endangle*. The remaining program segment draws elliptical shapes: three complete ellipses followed by three elliptical arcs.



**Fig. 17.3**

```
/*draw arcs in top row */  
arc(80, 80, 90, 270, 60);  
arc(240, 80, 180, 90, 60);  
arc(400, 80, 30, -30, 60);
```

```

arc(560, 80, 0, 360, 60);

/* draw ellipses in middle row */
ellipse(80, 240, 0, 360, 45, 60);
ellipse(240, 240, 0, 360, 60, 20);
ellipse(400, 240, 0, 360, 60, 60);

/* draw elliptical arcs in bottom row */
ellipse(80, 400, -180, 0, 30, 60);
ellipse(240, 400, 210, 150, 60, 20);
ellipse(400, 400, 360, 540, 20, 60);

```

#### 17.4.4 Setting Line Styles and Fill Styles

TC provides four predefined **line styles** that include solid line, dotted line, centre line and dashed line. We can select one of these styles for subsequent drawings using the **setlinestyle** function. This function can also be used to set a user-defined line style. The prototype of this function is given below.

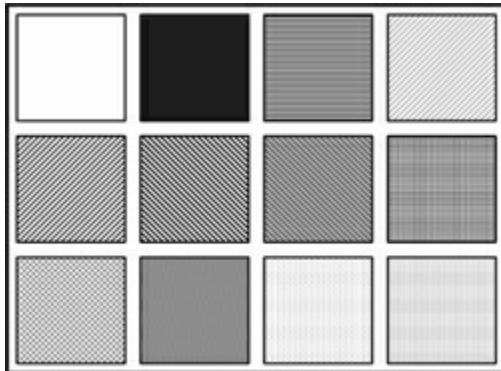
```
void setlinestyle(int linestyle, unsigned upattern, int thickness);
```

The <graphics.h> header file defines five enumerated constants for *linestyle* (values 0-4) as **SOLID\_LINE**, **DOTTED\_LINE**, **CENTER\_LINE**, **DASHED\_LINE** and **USERBIT\_LINE**. It also defines two constants for *thickness* as **NORM\_WIDTH** and **THICK\_WIDTH** for 1- and 3-pixel-wide lines, respectively. Note that while using the **USERBIT\_LINE** line style, the parameter *upattern* specifies the bit pattern of the line, e. g., *upattern* **0xF0F0** sets a dashed line style with a 4-pixel dash followed by a 4-pixel gap.

We may have to fill (or paint) the interior of objects or regions using different fill patterns and colours. TC provides 12 predefined fill patterns, shown in Fig. 17.4. A fill style comprises a fill pattern and a fill colour. We can set a **fill style** using the **setfiiistyle** function whose prototype is given below.

```
void setfiiistyle(int pattern, int colour);
```

The <graphics.h> header file provides 13 enumerated constants for fill patterns with values from 0 to 13. They are **EMPTY\_FILL**, **SOLID\_FILL**, **LINE\_FILL**, **LTSLASH\_FILL** (light slash fill), **SLASH\_FILL**, **BKSLASH\_FILL** (backslash fill), **LTBKSLASH\_FILL** (light backslash fill), **HATCH\_FILL** (light hatch fill), **XHATCH\_FILL** (heavy crosshatch fill), **INTERLEAVE\_FILL** (interleaving line fill), **WIDE\_DOT\_FILL**, **CLOSE\_DOT\_FILL** and **USER\_FILL**. Note that **EMPTY\_FILL** causes filling to be done in the background colour and **USER\_FILL** selects the user-defined fill pattern set using the **setfillpattern** function.



**Fig. 17.4 Built-in fill styles in TC**

#### 17.4.5 Drawing Filled Rectangles and Polygons

TC provides two functions to draw bars (i. e., filled rectangles): **bar** and **bar3d**. The **bar function** draws a filled rectangle and the **bar3d function** draws a three-dimensional bar. The prototypes of these functions are given below:

```
void bar(int left, int top, int right, int bottom);
void bar3d(int left, int top, int right, int bottom, int depth, int t
```

The **bar** function accepts coordinates of the top-left and bottom-right corners of a rectangle and draws a filled rectangle using the current fill style (fill pattern and fill colour). Note that the **bar** function does not draw the boundary of the rectangle and that changing the current line style has no effect on the output. If we wish to draw the boundary of a filled rectangle, we have three alternatives: use the **bar3d** function with 0 depth, use the **filipoly** function or draw a rectangle first and then fill it using the **floodfill** function.

Besides the coordinates of the top-left and bottom-right corners of a rectangular region, the **bar3d** function accepts two additional parameters: **depth** and **topflag**. The **depth** specifies the depth of three-dimensional bar (in pixels) and a **topflag** flag specifies whether the top of the bar should be drawn or not. Note that the **bar3d** function draws the outline of three-dimensional bar using the current line settings (style, thickness and colour) and fills its front side using the current fill style and colour.

In addition to the **drawpoly** function provided to draw a polygon, TC also provides the **filipoly function** to draw a filled polygon. Its prototype is given below.

```
void fillpoly(int numpoints, int *polypoints);
```

Note that the arguments are exactly the same as those used in the **drawpoly** function. The **fillpoly** function draws a polygon using the current line settings (line style, colour and thickness) and fills it using the current fill settings (fill pattern and colour). Note that while drawing a closed filled

polygon having  $n$  vertices, we pass  $n + 1$  points to **fillpoly** where the last point is the same as that of the first point.

#### Example 17.4 Draw two-dimensional and three-dimensional bars

This example illustrates the capabilities of the bar and **bar3d** functions. Its output is given in Fig. 17.5. The program segment given below first selects a white background, blue foreground and fill style comprising **SOLID\_FILL** pattern and cyan colour. It then draws four two-dimensional bars in the upper half of the screen. Observe that the first bar, which is drawn using a bar function, does not have the bounding rectangle. The next three two-dimensional bars, drawn using the **bar3d** function with depth and top flag set to 0, have the bounding rectangle in accordance with the current line style.

```
setbkcolor(WHITE);
setcolor(BLUE);
setfillstyle(SOLID_FILL, CYAN);

bar(10, 10, 150, 150); /* draw a bar */

/* draw filled rectangles using bar3d() with depth=0 & topflag=0 */
bar3d(160, 10, 310, 150, 0, 0); /* default style, thin solid line */

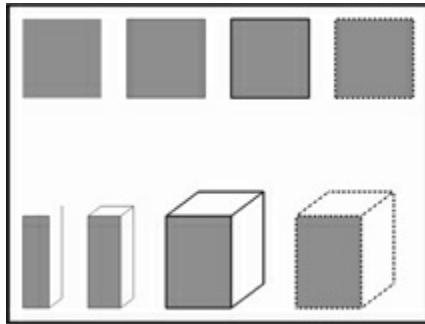
setlinestyle(SOLID_LINE, 0, THICK_WIDTH); /* thick solid line */ bar3d

setiinestyle(DASHED_LINE, 0, THICK_WIDTH); /* thick dashed line */ bar3d
```

The program segment given below, which is a continuation of above program segment, draws four three-dimensional bars in the lower half of the screen. The first two bars are drawn using a 1-pixel-wide solid line, whereas the last two bars are drawn using a 3-pixel-wide line (either continuous or dashed). Also, observe that all three-dimensional bars except the first are drawn with a top.

```
/* draw 3-D bars */
setlinestyle(SOLID_LINE, 0,
    NORM_WIDTH); /* thin solid line */
bar3d(10, 320, 50, 470, 20, 0); /* 20 depth, no top */

bar3d(100, 320, 150, 470, 20, 1); /* 20 depth, top */
setlinestyle(SOLID_LINE, 0, THICK_WIDTH); /* thick solid line */
bar3d(200, 320, 300, 470, 50, 1); /* 50 depth with top */
setlinestyle(DASHED_LINE, 0, THICK_WIDTH); /* thick dashed line */
bar3d(400, 320, 500, 470, 50, 1); /* 50 depth, no top */
```



**Fig. 17.5 Drawing 2-D and 3-D bars**

#### 17.4.6 Drawing Filled Circular and Elliptical Objects

TC provides three functions to draw filled circular and elliptical shapes: **fillellipse** to draw a filled ellipse, **pieslice** to draw a circular pie slice and **sector** to draw an elliptical pie slice. The prototypes of these functions are given below.

```
void    fillellipse(int x, int y, int xradius, int yradius);
void pieslice(int x, int y, int stangle, int endangle, int radius),
void sector(int x, int y, int stangle, int endangle, int xradius, int
```

The arguments in these functions are as follows:

<i>x, y</i>	<i>x</i> - and <i>y</i> -coordinates of the object	centre
<i>radius</i>	radius of pie slice (in pixels)	
<i>xradius, yradius</i>	radii of an elliptical shape in <i>x</i> -	and <i>y</i> -direction (in pixels)
<i>stangle, endangle</i>	starting and ending angle (in degrees) for pie slice or sector	

These functions use the current drawing colour and line thickness to draw the outline of an object. The complete outline is drawn only if the current line style is **SOLID\_LINE**; otherwise, only straight-line segments in the outline are drawn. The interior of the object drawn is filled using the current fill settings (fill pattern and fill colour). To draw a filled shape without an outline, we can set the drawing colour to the same as the fill colour.

TC does not provide a function to draw a filled circle. However, we can use the **fillellipse** function for the same. Alternatively, we can draw a circle and fill it using the **floodfill** function. Note that the **pieslice** and **sector** functions have the same parameters as that of the **arc** and **ellipse** functions, respectively.

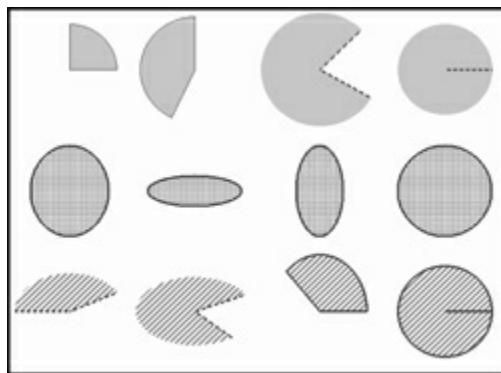
**Example 17.5** Draw filled circular and elliptical shapes

This example illustrates the capabilities and limitations of the `pieslice`, `fillellipse` and `sector` functions by drawing several filled circular and elliptical shapes shown in Fig. 17.6.

The first part of the program segment given below sets a light green solid fill and then draws the first two pie slices in the first row using the default line settings (1-pixel-wide solid line). Then it selects a thick dashed line and draws two more pie slices. Observe that the straight lines are drawn using this dashed line style but the arc is not drawn at all. Also, note that although the `stangle` and `endangle` are specified as -45 and 45 respectively, the `pieslice` is drawn from 45 to -45.

```
/* draw pieslices */
setfillstyle(SOLID_FILL, LIGHTGREEN);
pieslice( 80, 80, 0, 90, 60);
pieslice(240, 80, 245, 90, 70);

setlinestyle(DASHED_LINE, 0, THICK_WIDTH);
pieslice(400, 80, -45, 45, 75);
pieslice(560, 80, 0, 360, 60);
```



**Fig. 17.6** Drawing filled circular and elliptical objects

The second part of the program segment sets a light blue hatch fill and draws four filled ellipses in the middle row. Observe that although the current line setting is a thick dashed line, the ellipses are drawn using thick solid lines.

```
/* draw filled ellipses */
setfillstyle(HATCH_FILL, LIGHTBLUE);
fillellipse( 80, 240, 50,60);
fillellipse(240, 240, 60,20);
fillellipse(400, 240, 30, 60);
fillellipse(560, 240, 60, 60);
```

The last part of the program segment sets the dark gray slash fill style and draws the first two sectors in the last row. Observe that, as in the case of the `pieslice` function, the straight lines are drawn using this line style but the elliptical curve is not drawn at all. The program segment finally selects the thick

solid line and draws the last two sectors.

```
/* draw sectors */
setfillstyle(SLASH_FILL, DARKGRAY);
sector(80, 400, 180, 30, 70, 50);
sector(240, 400, 310, 30, 75, 45);

setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
sector(400, 400, 0, 135, 60, 75);
sector(560, 400, 0, 360, 60, 60);
```

### Program 17.1 Draw a bar graph

Write a program to display a bar graph.

**Solution:** In this program, we present a versatile function named `bar_graph` which allows us to specify most of the parameters for the bar graph being drawn that include

1. data for the bar graph – number of data values ( $n$ ), data array ( $data$ ) and maximum data value ( $ymax$ )
2. size and position parameters – position ( $x, y$ ) of the top-left corner of the bar graph, bar graph size ( $xsize, ysize$ ), bar width ( $w$ ) and margin on all sides ( $m$ )
3. colour parameter – graph background colour, bar colour and fill style and axis colour

We use the structure `bar_graph_data` to store data for the bar graph and structure `bar_graph_params` to store the size, position and colour parameters for the bar graph. The function `bar_graph` accepts pointers to variables of these structures.

The complete program to display the bar graph is given below. It declares `bar_graph_data` and `bar_graph_params` as global structures to store the data and parameters of a bar graph. The main function initializes variables `bg_data` and `bg_params` with the desired parameters of the bar graph. Then it initializes graphics and calls the `bar_graph` function to display a bar graph.

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

/* data for bar graph */
struct bar_graph_data {
    int n;                      /* number of data values */
    int data[20];                /* data array */
    int ymax;                    /* max value of y */
};
```

```

/* parameters for bar graph */ 
struct bar_graph_params {
    int x, y;           /* bar graph position */
    int xsize, ysize;   /* bar graph size */
    int margin;         /* margin on all sides */
    int bar_width;      /* bar width */
    int bgcolor;        /* background colour for bar graph */
    int bar_colour, bar_fill; / * bar colour and fill style */
    int axis_colour;   / * colour of axis */
    int ymax;           /* max y value */
};

void bar_graph(struct bar_graph_data *bgd, struct bar_graph_params *b
int main()
{
    struct bar_graph_data bg_data = {6, {55,67,82,95,67,45},100};

    struct bar_graph_params      bg_params = {
        120, 90,          /* x, y */
        400, 300,         /* xsize, ysize */
        40, 20,          /* margin, bar width */
        YELLOW,           /* background colour for bar graph */
        GREEN,SOLID FILL,/* bar colour and fill style */
        BLUE,             /* axis colour */
        100               /* max y value */
    };

    int gd=DETECT, gm;

    initgraph(&gd, &gm, "C:\\TC\\BGI");
    setbkcolor(WHITE);
    bar_graph(&bg_data, &bg_params);

    getch();
    closegraph();
    return 0;
}

/* draw a bar graph */
void bar_graph(struct bar_graph_data *bgd, struct bar_graph_params *b
{
    float x1, y1, x2, y2; /* coords of LT & RB corners of bar graph re
    float x3, y3, x4, y4; /* coods of LT & RB corners of bar */

```

```

float xscale, yscale;
int i;

/* draw bounding rectangle */
setcolor(bgp->axis_color);
setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
setfillstyle(SOLID_FILL, bgp->bgcolor);
bar3d(bgp->x, bgp->y, bgp->x + bgp->xsize, bgp->y + bgp->ysize, 0,

/* determine coords of bar graph area */ x1 =           bgp->x +
y1 = bgp->y + bgp->margin;
x2 = bgp->x + bgp->xsize - bgp->margin;
y2 = bgp->y + bgp->ysize - bgp->margin;

/* draw axes */
setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
line (x1, y1, x1, y2);      /*      x axis */
line (x1, y2, x2, y2);      /*      y axis */

/* determine xscale and yscale */
xscale = (float) (x2 - x1) / bgd->n;
yscale = (y2 - y1) / bgd->ymax;

/* draw bars */
setcolor(bgp->bar_color);
setfillstyle(bgp->bar_fill, bgp->bar_color);

y4 = y2 - 1;           /* y coord of RB corner of a bar */
for (i = 0; i < bgd->n; i++) {
    x3 = x1 + (i + 0.5) * xscale - bgp->bar_width/2.0;
                           /* x coord of LT corner of a
    y3 = y4 - bgd->data[i] * yscale; /* y coord of LT corner of a b
    x4 = x3 + bgp->bar_width; /*      x coor of RB      corner of a b
    bar3d(x3, y3, x4, y4, 0, 0);      /*      draw bar */
}
}

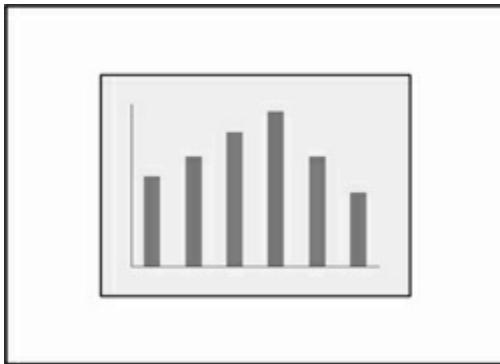
```

The output of this program is shown in Fig. 17.7.

Let us now understand how the bar graph is actually drawn. We assume that the margin  $m$  is the same on all sides and can be used to print the graph title, axis labels, etc. To draw the  $x$  and  $y$  axes, we first determine the coordinates of the top-left corner ( $x_1, y_1$ ) and bottom-right corner ( $x_2, y_2$ ) of the rectangular region for the actual bar graph. With a margin of  $m$  pixels on all sides, these coordinates are given as

$$x1 = x + m \quad x2 = x + xsize - m$$

$$y1 = y + m \quad y2 = y + ysize - m$$



**Fig. 17.7 A bar graph**

Now the  $y$  axis is drawn as a line from  $(x1, y1)$  to  $(x1, y2)$  and the  $x$  axis as a line from  $(x1, y2)$  to  $(x2, y2)$ .

To determine the coordinates of a bar, we require the scale factor in the  $x$  and  $y$  directions, which is obtained as the total pixels available for plotting in that direction divided by the maximum value to be plotted. Thus, we have

$$S_x = (x2 - x1) / n$$

$$S_y = (y2 - y1) / ymax$$

To draw the bars, we set up a **for** loop and in each iteration, we determine the coordinates of the top-left corner  $(x3, y3)$  and bottom-right corner  $(x4, y4)$  of the bar and draw it using the **bar** function. For the  $i$  th bar, the coordinates of the top-left and bottom-right corners are given as

$$x3 = x1 + (i + 0.5) S_x - w / 2 \quad x4 = x3 + w$$

$$y3 = y4 - data[i] * S_y \quad y4 = y2 - 1$$

## 17.5 Displaying Text

TC provides powerful capabilities to display text in the graphics mode. The functions in this category include those to display text (**outtext** and **outtextxy**), to set text characteristics such as font, text direction, character size, text justification (**settextstyle**, **settextjustify** and **setusercharsize**) and functions to find out the display size of a text string and current text settings (**textheight**, **textwidth** and **gettextsettings**).

### 17.5.1 Functions to Display Text

TC provides two functions to display text in the graphics mode: **outtext** and **outtextxy**. The prototypes of these functions are given below.

```
void outtext(char *str);
void outtextxy(int x, int y, char *str);
```

The **outtext** function displays the specified string at the CP, whereas the **outtextxy** function displays it at a specified location ( $x, y$ ). These functions use the current text settings (font, text direction, character size and text justification) to display the text string.

The **outtextxy** function does not modify the CP. However, if the text justification is **LEFT\_TEXT** (left-justified text) and the text direction is **HORIZ\_DIR** (horizontal text), which are the default settings, the **outtext** function advances the  $x$  value of CP by the width of the text string. This simplifies the output of the text using consecutive **outtext** functions.

When the graphics system is initialized, the default text settings are  $8 \times 8$  bit mapped font with left- and top-justified horizontal text. In addition, the default text colour is the same as the default drawing colour, i. e., white, and the CP is at  $(0, 0)$ . With the left- and top-justified horizontal text setting, the text is displayed to the right and below the CP (for the **outtext** function) or the specified point (for the **outtextxy** function). Note that the desired text colour is set using the **setcolor** function and other text characteristics are set using the **settextstyle**, **settextjustify** and **setusercharsize** functions.

### 17.5.2 Setting Text Characteristics

TC provides several text characteristics in the graphics mode. These include text fonts, built-in and user-defined character sizes, text direction and text justification. It provides three functions for setting text characteristics: **settextstyle**, **settextjustify** and **setusercharsize**.

The **settextstyle** function is used to set the text font, text direction and character size. These settings affect the text output by the subsequent **outtext** and **outtextxy** functions. Its prototype is given below.

```
void settextstyle(int font, int direction, int charszie);
```

TC provides five fonts. The `<graphics.h>` file provides enumerated constants for these fonts (with values 0 to 4): **DEFAULT\_FONT**, **TRIPLEX\_FONT**, **SMALL\_FONT**, **SANS\_SERIF\_FONT** and **GOTHIC\_FONT**. The **DEFAULT\_FONT**, which is the default font in graphics mode, is an  $8 \times 8$  bit-mapped font, i. e., each character is represented using  $8 \times 8$  bits. Hence, the magnification for these fonts can be only integer values. The other fonts are stroked fonts, i. e., characters in these fonts are represented using strokes. As a result, we can set non-integral magnification for these fonts using the **setusercharsize** function.

The text direction can be either **HORIZ\_DIR** (horizontal left to right text) or **VERT\_DIR** (vertical bottom to top text). The default text direction is horizontal.

The *charsize* argument specifies the character magnification. The allowed values of *charsize* are 0 to 10. The values 1 to 10 can be used with the bit-mapped as well as the stroked fonts. When used with bitmap fonts, the character size is magnified by that amount. For example, *charsize* of 4 displays a character in  $32 \times 32$  pixel rectangle. When used with stroked fonts, the character size increases as the value of *charsize* increases from 1 to 10. A zero value of *charsize* can be used only with stroked fonts. It magnifies the character size by a default character magnification factor of 4 or by user magnification as specified in a subsequent call to the **setusercharsize** function.

The **settextjustify** function is used to set the horizontal and vertical text justification, i.e., how the text will be aligned with the CP. Its prototype is given below.

```
void settextjustify(int horiz, int vert);
```

TC provides three enumerated constants (with values 0 to 2) for the *horiz* parameter, namely, **LEFT\_TEXT** (left-justified text), **CENTER\_TEXT** and **RIGHT\_TEXT** (right-justified text). It also provides three enumerated constants (with values 0 to 2) for the *vert* parameter, namely, **BOTTOM\_TEXT** (bottom-justified text), **CENTER\_TEXT** and **TOP\_TEXT** (top-justified text). The default settings for text justification are **LEFT\_TEXT** and **TOP\_TEXT**.

The **setusercharsize** function is used to set a user-defined character magnification for stroked fonts. Its prototype is as follows:

```
void setusercharsize(int multx, int divx, int multy, int divy);
```

This causes the default character width and height to be multiplied by *multx* / *divx* and *multy* / *divy*, respectively. Note that for a user-defined character size to be active, the *charsize* argument must be set to 0 in the previous call to the **settextstyle** function.

#### Example 17.6 Display text with font magnification

In the program segment give below, we display strings using different magnifications for various fonts. The output obtained is shown in Fig. 17.8.

```
settextstyle(DEFAULT_FONT, HORIZ_DIR, 2);
outtextxy(0, 30, "Default font. Magnification 2.");

settextstyle(TRIPLEX_FONT, HORIZ_DIR, 6);
outtextxy(0, 60, "Triplex font. Mag 6.");

settextstyle(SMALL_FONT, HORIZ_DIR, 10);
outtextxy(0, 130, "Small font. Mag 10.");

settextstyle(SANS_SERIF_FONT, HORIZ_DIR, 8);
outtextxy(0, 180, "SansSerif font 8.");
```

```
settextstyle(GOTHIC_FONT, HORIZ_DIR, 8);
outtextxy(0, 270, "GOTHIC font 8.");
```



Fig. 17.8

### 17.5.3 Enquiring Text Attributes

TC provides the **textheight** and **textwidth** functions to determine the height and width (in pixels) required to display a text string using the current text settings that include the font, size and magnification. The prototypes of these functions are given below.

```
int textheight (char *str);
int textwidth(char *str);
```

The **textheight function** is used to provide appropriate spacing between lines of text, whereas the **textwidth function** is used to provide appropriate spacing between the words in a line. While displaying multi-line paragraph text, we use the **textheight** function to test whether the next line fits in the screen (or current view port) or not. Similarly, the **textwidth** function is used to test whether the next word fits in the current line or not. We can also use these functions to change the text settings such that the given text string fits in a specified rectangular region.

The **gettextsettings** function obtains information about the current graphic text font that includes text font, direction, size and justification. Its prototype is as follows:

```
void gettextsettings(struct textsettingstype * texttypeinfo );
```

where *texttypeinfo* is a pointer to a **textsettingstype** structure defined in the **graphics.h** header file.

While working with graphics, we may often wish to change the text settings, for example, in a bar graph function, we may want to change the text settings several times to print the title, legends, etc. It is good practice to restore the original text setting so that the subsequent program is not affected by the changed settings. This can be achieved by saving the original text settings before we change them (typically at the beginning of the function) and then restore them after the text output is over (typically at the end of a function).

### Example 17.7 Using the `textheight` function to space lines of text

The program segment given below illustrates the use of the `textheight` function to provide appropriate spacing between the lines of text. It displays string "textheight demo" on several lines with increasing font sizes as shown in Fig. 17.9.

```
y = 0;  
for (s = 1; s <= 10; s++) {  
    settextstyle(SANS_SERIF_FONT, HORIZ_DIR, s);  
    outtextxy(10, y, "textheight demo");  
    y += textheight("textheight demo");  
    line(0, y, 639, y);  
}
```

The default text justification, i. e., left- and top- justified text, is assumed. The variable `y` which is used as the `y` position of the text string is initialized to zero before the `for` loop. The index variable of the `for` loop, which represents the font size, takes values from 1 to 10. For each value of `s`, initially the sans serif font of size `s` is set and string "textheight demo" is displayed at position (10, `y`). Then the value of `y` is increased by the height of the displayed string and a horizontal line is drawn at the current `y` value. Observe how the text lines are properly spaced in the output.



Fig. 17.9

## 17.6 Animation

Movement of graphic objects on screen is called **animation**. To enable animation, the desired object is displayed (and erased) at closely spaced successive locations on the object's path at a sufficiently fast rate. For smooth animation, the object is displayed about 30 or more times per second.

### 17.6.1 Animation of Simple Objects

If an object being moved is simple, we can directly use the graphics drawing commands to draw it at each desired position. Such an object can be erased by redrawing it at the same position in the background colour or by drawing a bar of appropriate size in the background colour. The steps involved in performing the object movement are given below.

1. Determine the next position ( $x, y$ ) for the object.
2. Draw the object at position ( $x, y$ ).
3. Provide a time delay (and/or do some other processing).
4. Erase the object at position ( $x, y$ ).
5. If movement is not yet over, go to step 1.

Two parameters that control the speed and smoothness of object motion are:

1. Distance between consecutive object locations. The speed is directly proportional to this distance. However, for smooth motion, this distance should be as small as possible.
2. The time delay between the drawing and erasing operations. This is the duration for which object is displayed before it is erased. The object speed is inversely proportional to this time delay. We usually use the delay function available in the `<dos.h>` header file of TC to provide the desired delay in milliseconds.

### Example 17.8 Moving a ball

The function `move_ball` given below moves a ball on the screen from the top to the bottom.

```
void move_ball(int x, int ystep, int radius, int color, int bgcolor, :  
{  
    int y;  
    for(y = radius; y < getmaxy() - radius; y += ystep) {  
        /* draw the ball */  
        setcolor(color);  
        setfillstyle(SOLID_FILL, color);  
        fillellipse(x, y, radius, radius);  
        delay(tdelay);  
  
        /* erase the ball */  
        setcolor(bgcolor);  
        setfillstyle(EMPTY_FILL, color);  
        fillellipse(x, y, radius, radius);  
    }  
}
```

The function accepts six arguments:  $x$  ( $x$  position of the ball),  $ystep$  (step size for ball motion),

`radius` (ball radius), `color` (ball colour), `bgcolor` (background colour) and `tdelay` (time delay in milliseconds). It uses a `for` loop with the loop counter `y` taking values from `radius` to `getmaxy() - radius` in steps of `ystep`. Within the loop, we first draw the ball in the specified color using the `fillellipse` function, wait for `tdelay` milliseconds and then erase the ball (i. e., draw it again in background colour). The function is called as shown below:

```
setbkcolor(WHITE);
move_ball(320, 1, 10, RED, WHITE, 10);
```

Note that we can also erase the ball by drawing a bar instead of an ellipse in the background colour as shown below:

```
bar(x - radius, y - radius, x + radius, y + radius);
```

## 17.6.2 Animation of Complex Objects

If an object being moved is complex, we require substantial time to draw it each time and the object motion may not be smooth. To overcome this problem, TC provides three functions specifically for animation: `getimage`, `putimage` and `imagesize`. The `getimage` function stores a bit image of a rectangular screen region in memory, the `putimage` function displays this stored bit image at a desired position on the screen and the `imagesize` function calculates the size of the memory block to store the bit image.

To perform animation, we first draw the desired object on the screen and then store its bit image in memory using the `getimage` function. This bit image is then displayed and erased at closely spaced successive locations on the object's path.

### The `imagesize` Function

The `imagesize` function returns the number of bytes required to store a bit image of a specified rectangular screen region. Its prototype is as follows:

```
unsigned imagesize(int left, int top, int right, int bottom);
```

The arguments `left`, `top`, `right` and `bottom` specify a rectangular region on the screen. The bit image captured by the `getimage` function contains information about every pixel in the desired region. As each pixel requires a fixed number of bits (say  $b$ ), we can calculate the required number of bytes as  $xyb/8$ , where  $x$  and  $y$  are the number of pixels in  $x$  and  $y$  directions, respectively, in the desired region, i. e.,  $x = right - left + 1$  and  $y = bottom - top + 1$ . However, as the number of bits required for each pixel varies greatly depending on the graphics driver and mode, it is a good idea to use the `imagesize` function instead. Note that the number of bytes required to store a bit image does not depend on the amount of drawing details.

### The `getimage` Function

The **getimage** function is used to store a bit image of a rectangular screen region in memory. Its prototype is as follows:

```
unsigned getimage(int left, int top, int right, int bottom,  
                 void *bitmap);
```

The arguments *left*, *top*, *right* and *bottom* specify a rectangular region to be captured and *bitmap* is a void (i. e., typeless) pointer that points to a block of memory where the captured bit image is to be stored.

It is possible to use a static array to store the captured image. However, the array size will either be insufficient or more than necessary. Hence, we use the **imagesize** function to calculate the exact size of the bit image and use dynamic memory allocation to allocate memory. This approach works correctly independent of the graphics driver or mode used and also avoids wastage of memory.

### The **putimage** Function

The **putimage** function displays a bit image previously saved in memory by the **getimage** function. Its prototype is as follows:

```
void putimage(int left, int top, void *bitimage, int op);
```

The arguments *left* and *top* specify the screen location where the **putimage** will position the upper left corner of the stored bit image. The pointer argument *bitimage* points to the saved bit image. The argument *op* is a combination operator that specifies how each pixel in the bit image will be combined with the corresponding pixel on screen when the bit image is displayed. TC provides several enumerated constants for **putimage** operators as shown in Table 17.3.

Table 17.3 *Enumerated constants for **putimage** operators*

Constant	Value	Meaning
COPY_PUT	0	Copies source bit image to the screen
XOR_PUT	1	Exclusive OR source bit image with that already on screen
OR_PUT	2	Inclusive OR source bit image with that already on screen
AND_PUT	3	AND source bit image with that already on screen
NOT_PUT	4	Copies the inverse of the source bit image to the screen

In the **COPY\_PUT** operation, the bit image is directly copied to the screen, overwriting the screen contents in that region. Thus, the displayed image will be exactly same as the bit image. However, while using other operations, the specified operation (e. g., *xor* for **XOR\_PUT**) is performed on bit patterns of corresponding pixels on the screen and bit image. Thus, the colours in the displayed image will often be different from those in the bit image.

The **XOR\_PUT** operation is particularly useful in animation. When we **XOR\_PUT** a bit image on the screen, it is displayed at that position (although the colours in it may change depending on the screen contents). Subsequently, when we **XOR\_PUT** that bit image again at the same screen position, the original screen contents are restored. Thus, to achieve object motion without modifying background, we display the stored bit image twice at each selected location on the object's path using the **XOR\_PUT** operation, with a delay between two **putimage** operations. The steps to be used for moving an object on screen are given below.

1. Determine the next position ( $x, y$ ) for the object.
2. Use the **putimage** function with **XOR\_PUT** operation to display the bit image at the desired position ( $x, y$ ).
3. Provide a time delay (and/or do some other processing).
4. Use the **putimage** function with **XOR\_PUT** operation to display the bit image at position ( $x, y$ ) again. This will actually erase the bit image and restore the original background.
5. If object movement is not yet over, go to step 1.

#### Example 17.9 Moving a ball

A program to move a ball from the top edge of screen to the bottom is given below. It first initializes the graphics system and sets the colours and fill style. Then it draws a ball using the **fillellipse** function, determines the memory required to store its bit image using the **imagesize** function, allocates the required memory using the **malloc** function and then captures the bit image using the **getimage** function.

Then the screen is cleared using the **clearviewport** function and a **for** loop is set up to move the ball using the **putimage** function with **XOR\_PUT** operation as explained above.

Finally the memory used for the bit image is released using the **free** function and the graphics system is closed.

```
#include <stdio.h>
#include<graphics.h>;
#include <stdlib.h>
#include <conio.h>
#include <dos.h>

int main()
{
    char *ball;          /* pointer to bit image */
    unsigned int sz;      /* image size */
    int gd = DETECT, gm;
    int y;
```

```

initgraph(&gd,           &gm, "C:\\TCP\\BGI");

setbkcolor(LIGHTBLUE);
setcolor(YELLOW);
setfillstyle(SOLID_FILL, YELLOW);

fillellipse(15, 15, 10, 10);      /* draw desired object */
sz = imagesize(5, 5, 25, 25);    /* determine image size
ball = (char *) malloc (sz);    /* allocate memory */
if (ball == NULL) {             /* test for memory allocation error
    outtext("Out of memory ...\\n");
    exit (1);
}

getImage(5, 5, 25, 25, ball);   /* store bit image to memory */
clearviewport();

for (y = 0; y < 475; y+=5) {
    putimage (320, y, ball, XOR_PUT); delay(5);
    putimage (320, y, ball, XOR_PUT);
}
getch();
free(ball);
closegraph();
return 0;
}

```

## Exercises

1. Answer the following questions in brief:
  - a. Compare text mode and graphics mode with respect to the text capabilities.
  - b. Define the following terms: pixel, screen resolution, palette, viewport, graphics driver and graphics mode
  - c. Assuming screen resolution of  $1024 \times 768$  pixels, determine the coordinates of four corners and the centre pixel of the screen.
  - d. State the purpose of following functions: `ellipse`, `sector`, `polygon`, `textheight`, `setfillstyle`
  - e. Explain the `XOR_PUT` operation of `putimage` function.

2. State the TC functions to
  - a. determine maximum value  $x$  and  $y$  of screen coordinates
  - b. close the graphics system
  - c. draw a 3-D bar
  - d. fill a region using current fill style
  - e. display text at specified position on screen
  - f. store bit image of rectangular screen region to memory
  - g. draw a line from CP to specified point
  - h. Display a specified pixel
3. Write statements for the following (assume VGA graphics mode, i. e.,  $640 \times 480$ )
  - a. initialize graphics system using autodetect mode
  - b. draw a line from point  $(100, 100)$  to  $(200, 100)$
  - c. move CP right by 10 pixels and down by 20 pixels
  - d. draw a filled square with its top-left corner at  $(50, 50)$  and 30 pixel sides
  - e. draw a circle at the centre of the screen and 30 pixels radius
  - f. set fill style to **HATCH\_FILL** in **RED** colour
  - g. draw a largest ellipse on the screen
4. Write complete C programs to draw simple graphics given below. Assume a  $640 \times 480$  graphics screen, use suitable colours and object sizes. Use loops where multiple objects of same type are required.
  - a. Using line drawing functions, draw a cube and colour the faces with different colours
  - b. Draw two squares (with sides 100 and 50 pixels) at the centre of the screen, join their corners and fill different colours in different regions.
  - c. Draw a rounded rectangle with following specs:  $100 \times 140$  pixel size and rounding radius of 20 pixels.
  - d. Draw a shooting target comprising of concentric circles filled with different colours.
  - e. Draw a chess board of size  $300 \times 300$  pixels. Use green and white colours for alternate squares on the board.
  - f. Draw a line diagram for a mobile or a calculator showing screen and keypad.
  - g. Draw any irregular shaped object (such as map of India, animal, bird, etc.)
5. Write functions to draw the following geometric figures where the lengths are specified in pixels  
(Hint: use trigonometric functions, if required, to determine coordinates of various points)

- a. Angle ABC (length AB, length BC, angle ABC and position of vertex B are function parameters)
  - b. Construction for bisection of an angle specified as in above example.
  - c.  $n$ -sided regular polygon (Function parameters: number of sides, length of side and position)
  - d. Carrom board coins (9 white, 9 black and 1 red) in initial position. (Function parameter: radius of coin)
  - e. Pie chart for  $n$  integer numbers
  - f. Exploded pie chart for  $n$  integer values
5. Write complete C programs for the following:
- a. Draw a table showing countries and their capitals
  - b. Display contents of a text file with pagination control
7. Display a clock showing the current time. The user should be able to customize various parameters for the clock such as size, position, colours, lengths of arms, etc.
3. Develop a program for the popular *bricks* game.

---

<sup>1</sup> As MS Windows 7 does not allow the full screen mode, the TC graphics programs will not run on it. The solution is to use DOSBOX software ([www.dosbox.com](http://www.dosbox.com)).

# Appendix A. The ASCII Character Set

CHAR	DEC	HEX
NUL	0	0
SOH	1	1
STX	2	2
ETX	3	3
EOT	4	4
ENQ	5	5
ACK	6	6
BEL	7	7
BS	8	8
TAB	9	9
LF	10	A
VT	11	B
FF	12	C
CR	13	D
SO	14	E
SI	15	F
DLE	16	10
DC1	17	11
DC2	18	12
DC3	19	13

CHAR	DEC	HEX

DC4	20	14
NAK	21	15
SYN	22	16
ETB	23	17
CAN	24	18
EM	25	19
SUB	26	1A
ESC	27	1B
FS	28	1C
GS	29	1D
RS	30	1E
US	31	1F
SP	32	20
!	33	21
"	34	22
#	35	23
\$	36	24
o	37	25
o		
&	38	26
'	39	27

CHAR	DEC	HEX
(	40	28
)	41	29
*	42	2A
+	43	2B
,	44	2C
-	45	2D
.	46	2E

/	47	2F
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B

CHAR	DEC	HEX
<	60	3C
=	61	3D
>	62	3E
?	63	3F
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49

J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F

CHAR	DEC	HEX
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57
X	88	58
Y	89	59
Z	90	5A
[	91	5B
\	92	5C
]	93	5D
^	94	5E
-	95	5F
`	96	60
a	97	61
b	98	62
c	99	63

CHAR	DEC	HEX

d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77

CHAR	DEC	HEX
x	120	78
y	121	79
z	122	7A
{	123	7B
I	124	7C
}	125	7D
~	126	7E

del	127	7F
-----	-----	----

# Appendix B. Summary of C Operators

Precedence Level	Operator	Description	Associativity
1	( ) [ ] . ->	Function call Array element access Structure member reference Structure member reference (using pointer)	Left to right
2	- ++ -- ! ~ * & sizeof (type)	Unary minus Increment Decrement Logical negation Bitwise complement Pointer reference (indirection) Address Size of an object Type cast (type conversion)	Right to left
3	*	Multiplication	
	/	Division	
	%	Modulus (remainder after integer division)	Left to right
4	+	Addition	
	-	Subtraction	Left to right
5	<< >>	Left shift Right shift	Left to right

6	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right
7	== !=	Equality Inequality	Left to right
8	&	Bitwise AND	Left to right
9	^	Bitwise XOR	Left to right
10		Bitwise OR	Left to right
11	&&	Logical AND	Left to right
12		Logical OR	Left to right
13	? :	Conditional expression	<b>Right to left</b>
14	= * = / = % = + = - = & = ^ =   = << = >> =	Assignment operators	<b>Right to left</b>
15	,	Comma operator	Left to right

# Appendix C. Summary of C Statements

## C. 1 Executable Statements

Syntax	Description
<code>;</code>	<b>Null statement:</b> Used as a placeholder where a statement is required but no operation is to be performed.
<code>variable = expression ;</code>	<b>Assignment statement:</b> Used to assign value of an expression to a variable. The <i>expression</i> may contain function calls.
<code>func_name ( arg1, arg2, ... );</code>  <code>variable = func_name ( arg1, arg2, ... );</code>	<b>Function call:</b> Used to perform activities specified in a function. Data can be passed to a function through arguments. Pointer arguments can be used to return values to calling function. The return value if any, may be ignored or assigned to a <i>variable</i> .
<code>{</code> <i>decl_statement</i> ; <code>...</code> <i>statement</i> ; <code>...</code> <code>}</code>	<b>Block statement:</b> Contains one or more executable C statements enclosed in a pair of braces, optionally preceded by one or more declarative statements. The <i>statement</i> may be any executable C statement including a block statement.
<code>if ( expr )</code> <i>statement1</i> ; <code>else</code> <i>statement2</i> ;	<b>if statement:</b> Used to execute either of two statements depending on the outcome of an expression. The statements may be any of the executable statements or block statements.
<code>for( initial_expr;</code> <i>final_expr</i> ; <code>update_expr )</code> <i>statement</i> ;	<b>for statement:</b> Entry-controlled loop used to repeatedly execute a statement or a block of statements. Usually used in situations where number of iterations are known in advance. One or more expressions within parentheses may be omitted.
<code>while ( expr )</code>	<b>while statement:</b> Entry controlled loop used to repeatedly execute a statement or

<code>statement ;</code>	a block of statements. Usually used in situations where number of iterations are not known in advance.
<code>do   statement ; while (expr);</code>	<b>do ... while statement:</b> Exit-controlled loop used to repeatedly execute a statement or a block of statements depending on the value of <i>expr</i> . Usually used in situations where number of iterations are not known in advance.
<code>switch (expr)   {     case value1:       statement;       ...       break;     case value2:       statement;       ...       break;       ...     default:       statement;       ...       break;   }</code>	<b>switch statement:</b> Used to execute one of the alternative group of statements depending on the value of <i>expr</i> which should be an integral expression. If <b>break</b> statement is not used after a case, the execution falls through to the next case. The <b>default</b> clause is executed when the value of <i>expr</i> does not match with any of the case values.
<code>break;</code>	<b>break statement:</b> Used in a loop to stop its execution and continue with the statement following the loop. Also used in a <b>switch</b> statement at the end of statements within a case, to avoid execution of subsequent cases.
<code>continue;</code>	<b>continue statement:</b> Used in a loop to stop the execution of current iterations in a loop and proceed for next iteration.

## C. 2 Declarative Statements

Syntax	Description
<code>data_type var1, var2,       ... ;        data_type var1 =           expr1, var2 = expr2,</code>	<b>Scalar variables:</b> Used to declare one or more variables of type <i>data_type</i> . Initial values of these variables are garbage.  Initial values can be specified for one or more variables. Constants must be initialized when they are declared.

<pre>...; <b>const</b> <i>data_type</i> <i>var1</i> = <i>expr1</i>, ...;</pre>	
<pre><i>data_type arr1[sz1],</i> arr2[sz2], ... ;</pre>  <pre><i>data_type arr1 [sz] =</i> {<i>val1, val2, ...</i>}, ... ;</pre>	<p><b>1-D arrays:</b> Used to declare one or more one-dimensional arrays of type <i>data_type</i>. Initial values of array elements are garbage. Declaration of scalar variables and arrays can be combined.</p> <p>1-D array can be initialized using a comma separated list of values enclosed in braces. Remaining array elements, if any, are initialized to zero. Array size <i>sz</i> can be omitted. More than one arrays can be initialized using a separate initializer list for each array.</p>
<pre><i>data_type arr1 [sz1]</i> [sz2], ... ;</pre>  <pre><i>data_type arr[sz1]</i> [sz2] = {   <i>val00, val01, ... ,</i>   <i>val10, val11, ...</i> }, ... ;</pre>  <pre><i>data_type arr[sz1]</i> [sz2] = {   {<i>val00, val01, ...</i>},   {<i>val10, val11, ...</i>},   ... }, ... ;</pre>	<p><b>2-D Arrays:</b> Used to declare two dimensional array (i. e. a matrix) arrl of size <i>s1 × s2</i>. Initial values of elements are garbage. More than one arrays can be declared. Declaration of scalar variables and arrays can be combined.</p> <p>2-D array can be initialized (row by row) using a comma separated list of values enclosed in braces. Remaining array elements, if any, are initialized to zero. Array size <i>s1</i> can only be omitted. More than one arrays can be initialized using a separate initializer lists for each array.</p> <p>2-D array can also be initialized by specifying initializer list for each row. Remaining array elements, if any, in matrix or a row are initialized to zero.</p>
<pre><i>data_type *ptr1,</i> *ptr2, ... ;</pre>  <pre><i>data_type var,</i> arr[sz], ... ;</pre> <pre><i>data_type *ptr1 =</i> &amp;<i>var1, *ptr2 = arr,</i> ... ;</pre>	<p><b>Pointer variables:</b> Used to declare one or more pointer variables of type <i>data_type</i>.</p> <p>A pointer variable can be initialized with address of a variable or an array (of same type) declared earlier. Note that address operator is used with scalar variable but not with array name.</p>
<pre><b>enum</b> <i>enum_type</i> {</pre>	<p><b>Enumerated type:</b> Used to declare an enumerated type having enumerated</p>

<pre><code>c1, c2, ... };  <b>enum</b> enum_type {c1,     c2 = val2, ... };  <b>enum</b> enum_type var1, var2, ... ;</code></pre>	<p>constants <math>c1, c2, \dots</math> with values starting from 0. Values can be specified for one or more constants. Subsequent constants are assigned consecutive values. Once an enumerated type is declared, variables of that type can be declared as shown. These variables can also be initialized.</p>
<pre><code><b>struct</b> tag {     data_type member_list;      ... };  <b>struct</b> tag var1,     var2, ... ;  <b>struct</b> tag var1 = {     val1, val2, ... }, ... ;      <b>struct</b> tag {         data_type         member_list;         ...     } var1, var2, ... ;      <b>struct</b> tag {         data_type         member_list;         }     var1 = { val1, val2,         ... }, ... ;</code></pre>	<p><b>Structure:</b> Used to declare a structure. Members of a structure may be scalar variables, arrays or pointers.  Once a structure is declared, structure variables can be declared as shown.  Structure variables can be initialized as well using a comma-separated list of values enclosed in braces.  Structure variables can also be declared in declaration of a structure. These variables may be scalar variables, pointers or arrays.  A structure variable declared in structure declaration can also be initialized by specifying a comma-separated list of values enclosed in braces.</p>
<pre><code><b>typedef</b> type new_type;  new_type var, arr[sz], *ptr;  <b>typedef struct</b> tag {</code></pre>	<p>Type definitions: Declares <math>new\_type</math> as a new name for existing data type <math>type</math>.  This new type name can be used to declare variables, pointers and arrays of that type.  <b>typedef</b> can be used to declare a structure type as shown.</p>

```
    data_type  
    member_list ;  
    ...  
} new_type;  
typedef enum  
enum type {c1, c2,...  
}  
new_type;  
typedef int  
*iarr_ptr[10];
```

It can also be used to declare an enumerated type as shown.

To declare an arbitrary type, first declare new type name as if a variable of that type is being declared and then precede the definition with **typedef** keyword as shown for a type array of pointers to **int**.

# Appendix D. The C Standard Library

## D.1 C Standard Library Header Files

Header file	Explanation
<code>assert.h</code>	<code>assert</code> debugging macro used to detect logical errors and bugs in a program
<code>ctype.h</code>	Character classification and conversion macros
<code>errno.h</code>	Constant mnemonics for error codes
<code>float.h</code>	Constant for implementation-specific properties of floating-point library
<code>limits.h</code>	Constant for implementation-specific properties of integer types
<code>locale.h</code>	Localization functions i. e. functions that provide information specific to languages and countries
<code>math.h</code>	Common mathematical functions
<code>setjmp.h</code>	Macros <code>setjmp</code> and <code>longjmp</code> used for non-local jumps
<code>signal.h</code>	Signal handling functions
<code>stdarg.h</code>	Macros for accessing variable number of arguments passed to functions
<code>stddef.h</code>	Common data types and macros
<code>stdio.h</code>	Stream-level I/O routines, standard I/O predefined streams, types and macros for standard I/O
<code>stdlib.h</code>	Commonly used routines such as conversion, search, sort etc.
<code>string.h</code>	String and memory manipulation routines
<code>time.h</code>	Time conversion and other time-related routines

## D.2 `assert.h` Header file

Function	Prototype	Explanation
assert	void assert(int cond)	Test the value of condition <code>cond</code> and abort if it is false (useful during debugging)

## D.3 ctype.h Header File

Function	Prototype	Explanation
isalnum	int isalnum(int c)	Return true if <i>c</i> is an alphanumeric character (letter or digit)
isalpha	int isalpha(int c)	Return true if <i>c</i> is a letter
iscntrl	int iscntrl(int c)	Return true if <i>c</i> is a control character (0x00 to 0x1F) or delete character (0x7F)
isdigit	int isdigit(int c)	Return true if <i>c</i> is a digit
isgraph	int isgraph(int c)	Return true if <i>c</i> is a printable character other than space
islower	int islower(int c)	Return true if <i>c</i> is a lowercase letter
isprint	int isprint(int c)	Return true if <i>c</i> is a printable character (0x20 to 0x7E)
ispunct	int ispunct(int c)	Returns true if <i>c</i> is a punctuation character i. e. a printing character other than letter, digit or space
isspace	int isspace(int c)	Return true if <i>c</i> is a whitespace character (space, tab, carriage return, new line, vertical tab and formfeed)
isupper	int isupper(int c)	Return true if <i>c</i> is an uppercase letter
isxdigit	int isxdigit(int c)	Return true if <i>c</i> is a hexadecimal digit (0 to 9, A to F, a to f)
tolower	int tolower(int c)	Convert character <i>c</i> to lowercase
toupper	int toupper(int c)	Convert character <i>c</i> to uppercase

## D.4 errno.h Header File

Macro	Explanation
EDOM	Domain Error (function argument values outside the domain over which a function is defined)
ERANGE	Range error (result of a mathematical function outside range of <code>double</code> type)

Variable	Explanation
errno	Set by library functions when an error occurs

## D.5 float.h Header File †

Constant	Explanation
FLT_RADIX	Radix of exponent representation
FLT_DIG†	Number of digits of precision in a <b>float</b>
FLT_EPSILON†	Minimum positive <b>float</b> number $x$ such that $1.0 + x \neq 1.0$
FLT_MANT_DIG†	Number of base FLT_RADIX digits in the mantissa of <b>float</b>
FLT_MAX†	Maximum value of <b>float</b>
FLT_MAX_10_EXP†	Maximum value (base 10) of exponent part of a <b>float</b>
FLT_MAX_EXP†	Maximum value (base FLT_RADIX) of exponent part of a float
FLT_MLN†	Minimum value of <b>float</b>
FLT_MIN_10_EXP†	Minimum value (base 10) of exponent part of a <b>float</b>
FLT_MIN_EXP†	Minimum value (base FLT_RADIX) of exponent part of a float
FLT_ROUNDS	Rounding behavior (-1: indeterminate, 0: towards zero, 1: to nearest, 2: towards +infinity, 3: towards -infinity)

† Only constants for **float** type are given to save space. The constants for **double** and **long double** data types are obtained by replacing **FLT** with **DBL** and **LDBL** respectively (except for **FLT\_RADIX** and **FLT\_ROUNDS** constants)

## D.6 **limits.h** Header File

Constant	Explanation	Allowable value
CHAR_BIT	Number of bits in char type	≥8
CHAR_MAX	Maximum value for char type	UCHAR_MAX (if char is unsigned) SCHAR_MAX (if char is signed)
CHAR_MIN	Minimum value for char type	0 (if char is unsigned) SCHAR_MIN (if char is signed)
INT_MAX	Maximum value for int type	≥+32767
INT_MIN	Minimum value for int type	≤-32767
LONG_MAX	Maximum value for long int type	≥+2147483647
LONG_MIN	Minimum value for long int type	≤-2147483647
SCHAR_MAX	Maximum value for signed char type	≥+127
SCHAR_MIN	Minimum value for signed char type	≤-127
SHRT_MAX	Maximum value for short int type	≥+32767
SHRT_MIN	Minimum value for short int type	≤-32767
UCHAR_MAX	Maximum value for unsigned char type	≥255U
UINT_MAX	Maximum value for unsigned int type	≥65535U
ULONG_MAX	Maximum value for unsigned long type	≥4294967295U
USHRT_MAX	Maximum value of unsigned short type	≥65535U

## D.7 `locale.h` Header File

Function	Prototype	Explanation
localeconv	struct lconv *localeconv(void)	Return a pointer to a structure of type struct lconv set according to current locale
setlocale	char *setlocale(int category, const char *locale)	Set up a locale i. e. country-specific formats for numeric, monetary and time values.

## D.8 `math.h` Header File

Function	Prototype	Explanation
acos	double acos(double x)	$\cos^{-1}(x)$ in range $[0, \pi]$ , $x \in [-1, 1]$
asin	double asin(double x)	$\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$ , $x \in [-1, 1]$
atan	double atan(double x)	$\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$
atan2	double atan2(double y, double x)	$\tan^{-1}(y/x)$ in range $[-\pi, \pi]$
ceil	double ceil(double x)	$\lceil x \rceil$ , Smallest value greater than or equal to $x$
cos	double cos(double x)	cosine of $x$
cosh	double cosh(double x)	hyperbolic cosine of $x$
exp	double exp(double x)	exponential function $e^x$
fabs	double fabs(double x)	$ x $ , Absolute value of $x$
floor	double floor(double x)	$\lfloor x \rfloor$ , Largest value less than or equal to $x$
fmod	double fmod(double x, double y)	$x$ modulo $y$ i.e. Remainder of $x/y$
frexp	double frexp(double x, double *expo)	Split a double number into mantissa and exponent
ldexp	double ldexp(double x, double y)	Return $x \times 2^y$
log	double log(double x)	natural logarithm, $\ln(x)$ , $x > 0$
log10	double log10(double x)	base 10 logarithm, $\log_{10}(x)$ , $x > 0$
modf	double modf(double x, double *ipart)	Split integer and fractional part of $x$ . Store integer part in $*ipart$ and return fractional part.
pow	double pow(double x, double y)	$x^y$ . A domain error occurs if $x = 0$ and $y \leq 0$ or if $x < 0$ and $y$ is not an integer
sin	double sin(double x)	sine of $x$
sinh	double sinh(double x)	hyperbolic sine of $x$
sqrt	double sqrt(double x)	$\sqrt{x}$ , $x \geq 0$ .
tan	double tan(double x)	tangent of $x$
tanh	double tanh(double x)	hyperbolic tangent of $x$

Macro	Explanation
HUGE_VAL	Overflow value for math functions

## D.9 `setjmp.h` Header File

Function	Prototype	Explanation
setjmp	int setjmp(jmp_buf jmpb)	Set a nonlocal goto
longjmp	void longjmp(jmp_buf jmpb, int retval)	Performs a nonlocal goto

Type	Explanation
jmp_buf	Buffer of type jmp_buf is used to save and restore the program task state

## D.10 signal.h Header File

Function	Prototype	Explanation
raise	int raise(int sig)	Send a software signal of type sig to the program
signal	void (*signal(int sig, void (*func) (int sig))) (int)	Specify signal-handling actions

Type	Explanation
sig_atomic_t	Type of object that can be safely modified as an atomic entity, even in presence of asynchronous interrupts

## D.11 stdarg.h Header File

Function	Prototype	Explanation
va_arg	void va_arg(va_list ap, type)	Expands to an expression that has same type and value as the next argument
va_end	void va_end(va_list ap)	Helps perform called function a normal return
va_start	void va_start(va_list ap, lastfix)	Sets ap to point to first of the variable arguments passed to the function

Type	Explanation
va_list	Type for array that holds information needed by va_arg and va_end

## D.12 stddef.h Header File

Type	Explanation
ptrdiff_t	Type used to represent difference between two pointers
size_t	Type used for memory object sizes and repeat counts

wchar_t	Type for wide character constants
---------	-----------------------------------

Constant	Explanation
NULL	Null pointer value

## D.13 stdio.h Header File

Function	Prototype	Explanation
clearerr	void clearerr(FILE *fp)	Reset error and end-of-file indicators for stream <i>fp</i>
fclose	int fclose(FILE *fp)	Close a specified stream
feof	int feof(FILE *fp)	Macro that tests if end-of-file has been reached for stream <i>fp</i>
ferror	int ferror(FILE *fp)	Macro to test if an error has occurred on a stream <i>fp</i>
fflush	int fflush(FILE *fp)	Flush a stream i. e. write the contents of output buffer to associated file
fgetc	int fgetc(FILE *fp)	Get a character from a stream <i>fp</i>
fgetpos	int fgetpos(FILE *fp, fpos_t *pos)	Get current file pointer position
fgets	char *fgets(char *s, int n, FILE *fp)	Get a string from a stream
fopen	FILE *fopen(const char *fname, const char *mode)	Open a stream
fprintf	int fprintf(FILE *fp, const char *fprmat, ...)	Send formatted output to stream
fputc	int fputc(int c, FILE *fp)	Output a character to a stream
fputs	int fputs(const char *s, FILE *fp)	Output a string to a stream

Function	Prototype	Explanation
fread	size_t fread(void *ptr, size_t size, size_t n, FILE *fp)	Read a specified number of equal-sized data items from an input stream into a block
freopen	FILE *fopen(const char *fname, const char *mode, FILE *fp)	Associate a new file with an open stream
fscanf	int fscanf(FILE *fp, const char *format, ...);	Scans and formats input from a stream
fseek	int fseek(FILE *fp, long offset, int whence)	Reposition the file pointer of a stream
fsetpos	int fsetpos(FILE *fp, const fpos_t *pos)	position the file pointer of a stream
ftell	long ftell(FILE *fp)	Return the current file pointer
fwrite	size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp)	Append a specified number of equal-sized data items to an output file
getc	int getc(FILE *fp)	Macro to get one character from a stream
getchar	int getchar(void)	Macro to gets a character from stdin
gets	char *gets(char *str)	Get a string fro stdin
perror	void perror(const char *s)	Print a system error message
printf	int printf(const char *format, ...)	Sends formatted output to stdout
putc	int putc(int c, FILE *fp)	Macro to output a character to a stream
putchar	int putchar(int c)	Macro to output a character on stdout
puts	int puts(const char *s)	Output a string to stdout followed by a newline
remove	int remove(const char *fname)	Macro to remove a file
rename	int rename(const char *oldname, const char *newname)	Rename a file
rewind	void rewind(FILE *fp)	Reposition file pointer to file beginning
scanf	int scanf(const char *format, ...)	Scan and format input from stdin
setbuf	setbuf(FILE *fp, char *buf)	Assign buffering to a stream
setvbuf	int setvbuf(FILE *fp, char *buf, int type, size_t size)	Assign buffering to a stream. It can dynamically allocate and free the buffer.
sprintf	int sprintf(const *buffer, const char *format, ...)	Send formatted output to a string
sscanf	int sscanf(const char *buffer, const char *format, ...)	Scan and format input from a string
tmpfile	FILE *tmpfile(void)	Create a temporary file and open it for update (w+b)
tmpnam	char *tmpnam(char *sptr)	Create a unique file name
ungetc	int ungetc(int c, FILE *fp)	Push a character back into input stream
vfprintf	int vfprintf(FILE *fp, const char *format, va_list arglist)	Send formatted output to a stream, using an argument list
vprintf	int vprintf(const char *format, va_list arglist)	Send formatted output to stdout, using an

Function	Prototype	Explanation
		argument list
vsprintf	int vsprintf(char *buffer, const char *format, va_list arglist)	Send formatted output to a string, using an argument list

Type	Explanation
FILE	Type of object used to contain stream control information
fpos_t	Type of object used to record unique values of a stream's file position indicator

Object	Explanation
stdin stdout stderr	Predefined objects of type (FILE *) referring to standard input, standard output and standard error respectively

Constant	Explanation
_IOFBF _IOLBF _IONBF	Values to control stream buffering in conjunction with <code>setvbuf</code> function (_IOFBF: Fully buffered, _IOLBF: Line buffered, _IONBF: Not buffered)
BUFSIZ	Size of the buffer used by <code>setbuf</code> function (min 256)
EOF	Negative integral constant expression, indicating the end-of-file condition on a file
FILENAME_MAX	Maximum filename length or recommended array size to store a filename
FOPEN_MAX	Minimum number of files that can be concurrently opened
L_tmpnam	Maximum length string generated by <code>tmpnam</code> function
SEEK_CUR SEEK_END SEEK_SET	Constants to control action of <code>fseek</code>
TMP_MAX	Minimum number of unique file names generated by <code>tmpnam</code> function

## D.14 stdlib.h Header File

Function	Prototype	Explanation
abort	void abort(void)	Abnormally terminate a program
abs	int abs(int x)	Return absolute value of an integer <i>n</i>
atexit	int atexit(atexit_t func)	Register termination function
atof	double atof(const char *s)	Convert string <i>s</i> to double and return it
atoi	int atoi(const char *s)	Convert string <i>s</i> to integer and return it
atol	long atol(const char *s)	Convert string <i>s</i> to long integer and return it

# Appendix E. Turbo C

## E.1 Turbo C Menus

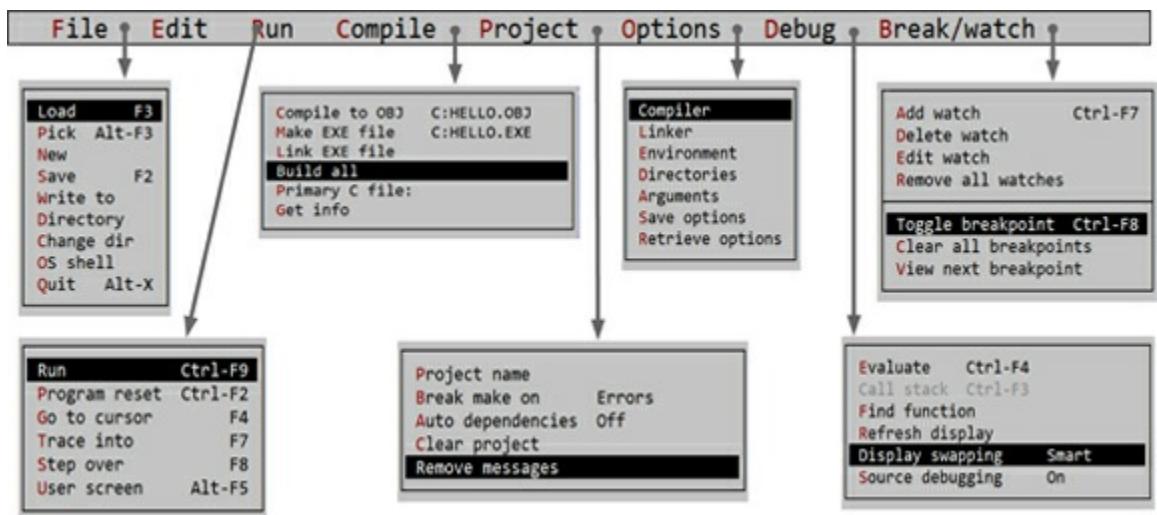


Fig E.1 Turbo C main menu

## E.2 Turbo C Hot keys

Category	Hot key	Function	Meaning
Help system	<i>F1</i>	Help	Display help screen
	<i>Alt-F1</i>	Last help	Display last help screen
	<i>Ctrl-F1</i>	Language help	Display help on item at cursor in <i>Edit</i> window
	<i>Shift-F10</i>	Version	Display TC version screen
File operations	<i>F2</i>	Save	Save file in <i>Edit</i> window to disk
	<i>F3</i>	Load	Load a file to <i>Edit</i> window
	<i>Alt-X</i>	Exit	Exit Turbo C
Window operations	<i>F5</i>	Zoom window	Zoom in/out active window
	<i>F6</i>	Switch window	Switch active window
	<i>Alt-F5</i>	User screen	Display user screen (program output)
	<i>Alt-F6</i>	Switch window contents	Switch <i>Edit</i> window contents between two program files
Program compilation and execution	<i>Alt-F9</i>	Compile	Compile a program to .OBJ file
	<i>F9</i>	Make	Compile/Link a program
	<i>Ctrl-F9</i>	Run	Run program in <i>Edit</i> window
	<i>Ctrl-F2</i>	Program reset	Reset execution of a program
	<i>F4</i>	Go to cursor	Run the program to cursor position
	<i>F7</i>	Trace into	Execute a program tracing into function calls
	<i>F8</i>	Step over	Execute a program skipping function calls
	<i>Alt-F7</i>	Previous error	Locate previous error in message and <i>Edit</i> window
	<i>Alt-F8</i>	Next Error	Locate the next error in message and <i>Edit</i> window
Program Debugging	<i>Ctrl-F3</i>	Call stack	Display call stack window
	<i>Ctrl-F4</i>	Evaluate	Evaluate an expression
	<i>Ctrl-F7</i>	Add watch	Add a watch expression to watch window
	<i>Ctrl-F8</i>	Toggle breakpoint	Toggle breakpoint on/off for a source line
Menu	<i>F10</i>	Menu	Toggles menu and active window
	<i>Alt-B</i>	Break/watch menu	Display Break/watch menu
	<i>Alt-C</i>	Compile menu	Display Compile menu
	<i>Alt-D</i>	Debug menu	Display Debug menu
	<i>Alt-E</i>	Edit file	Display Compile menu
	<i>Alt-F</i>	File menu	Display File menu
	<i>Alt-O</i>	Options menu	Display Options menu
	<i>Alt-P</i>	Project menu	Display Project menu
	<i>Alt-R</i>	Run menu	Display Run menu

### E.3 Commonly-used editor commands in Turbo C

Command	Action	Command	Action
<b>Deletion</b>		<b>Block operations</b>	
<i>Ctrl+Y</i>	Delete line	<i>Ctrl+K B</i>	Mark block beginning
<i>Ctrl+Q Y</i>	Delete to end of line	<i>Ctrl+K K</i>	Marks block end
<i>Ctrl+T</i>	Delete word right	<i>Ctrl+K C</i>	Copy a block
		<i>Ctrl+K V</i>	Move a block
<b>Find and replace</b>		<i>Ctrl+K Y</i>	Delete a block
<i>Ctrl+Q F</i>	Find	<i>Ctrl+K R</i>	Read a block (file) from disk
<i>Ctrl+Q A</i>	Find and replace	<i>Ctrl+K W</i>	Write a block to disk (as a file)
<i>Ctrl+L</i>	Repeat last find	<i>Ctrl+K H</i>	Hide/display block
<i>Esc</i>	Abort operation		

## E.4 Capabilities of various memory models in Turbo C

Memory Model	Memory available for			Pointer type	
	Code	Stack and Static Data	Heap	Function Pointers	Data Pointers
<b>Tiny</b>	Total 64K			Near	Near
<b>Small</b>	64 KB	64 KB		Near	Near
<b>Medium</b>	1 MB	64 KB		Far	Near
<b>Compact</b>	64 KB	64 KB	1 MB	Near	Far
<b>Large</b>	1 MB	64 KB	1 MB	Far	Far
<b>Huge</b>	1 MB	64 KB	Multiple data segments, each 64 KB in size	Far	Far