COMPUTER PROGRAMMING AND UTILIZATION

CC: 110003

(Common for all Engineering Branches)

GTU – June 2011

About the Author

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

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

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

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

Computer Programming and Utilization

CC: 110003

(Common for all Engineering Branches)

GTU – June 2011

E Balagurusamy

Chairman

EBG Foundation

Coimbatore



Tata McGraw Hill Education Private Limited

NEW DELHI

McGraw-Hill Offices

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



Tata McGraw-Hill

Published by Tata McGraw Hill Education Private Limited,
7 West Patel Nagar, New Delhi 110 008.

Computer Programming and Utilization, 2e (For GTU 2011)

Copyright © 2011 by the Tata McGraw Hill Education Private Limited.
No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the author. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

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

ISBN (13): 978-0-07-132992-7
ISBN (10): 0-07-132992-7

Vice President and Managing Director—McGraw-Hill Education, Asia-Pacific Region: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Publishing Manager (SEM and Tech Ed.): *Shalini Jha*
Asst Sponsoring Editor: *Tina Jajoriya*
Asst Development Editor: *Shikha Jain*
Executive—Editorial Services: *Sohini Mukherjee*
Sr Production Manager: *Satinder S Baveja*
Asst Production Manager: *Anjali Razdan*

Dy Marketing Manager (SEM and Tech Ed.): *Biju Ganesan*
Sr Product Specialist (SEM and Tech Ed.): *John Mathews*

General Manager—Production: *Rajender P Ghansela*
Asst General Manager—Production: *B L Dogra*

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

Typeset at *Anvi Composers*, A1 B Block, Paschim Vihar, New Delhi 110063 and printed at Pashupati Printers Pvt. Ltd., 1/429/16, Gali No. 1, Friends Colony Indl. Area, G.T. Road, Shahdara, Delhi 110095

Cover Printer: SDR Printers

RBXLCRACRAZRY

The McGraw·Hill Companies

Contents

Preface *xiii*
Roadmap to the Syllabus *xv*

UNIT 1

1. Introduction to Computer and Programming **1.1–1.25**

1.1 Introduction *1.1*
1.2 Basic Anatomy of a Computer System *1.2*
1.3 Input Devices *1.3*
1.4 Output Devices *1.5*
1.5 Software *1.7*
1.6 Hardware *1.9*
1.7 Programming Languages *1.9*
1.8 Programming Environment *1.12*
1.9 Introduction to the Design and Implementation of Correct, Efficient and Maintainable Programs *1.13*
1.10 Structured Programming *1.14*
1.11 Problem Solving *1.14*
1.12 Algorithms *1.14*
1.13 Flowcharts *1.17*
Just Remember *1.24*
Review Questions *1.24*
Multiple Choice Questions *1.25*

UNIT 2

2. Fundamentals of C **2.1–2.19**

2.1 History of C *2.1*
2.2 Importance of C *2.2*
2.3 Sample Program I: Printing a Message *2.3*
2.4 Sample Program 2: Adding Two Numbers *2.6*
2.5 Sample Program 3: Interest Calculation *2.7*
2.6 Sample Program 5: Use of Math Functions *2.9*
2.7 Basic Structure of C Programs *2.11*
2.8 Programming Style *2.12*
2.9 Executing a 'C' Program *2.13*

Just Remember 2.14
Review Questions 2.14
Programming Exercises 2.16
Debugging Exercises 2.18

3. Data Types in C 3.1–3.27

3.1 Introduction *3.1*
3.2 Character Set *3.1*
3.3 C Tokens *3.3*
3.4 Keywords and Identifiers *3.3*
3.5 Constants *3.4*
3.6 Variables *3.8*
3.7 Data Types *3.9*
3.8 Declaration of Variables *3.12*
3.9 Assigning Values to Variables *3.15*
3.10 Defining Symbolic Constants *3.20*
3.11 Declaring a Variable as Constant *3.22*
3.12 Declaring a Variable as Volatile *3.22*
3.13 Overflow and Underflow of Data *3.22*
Just Remember 3.23
Review Questions 3.23
Programming Exercises 3.25
Debugging Exercises 3.26

4. Operators and their Hierarchy 4.1–4.31

4.1 Introduction *4.1*
4.2 Arithmetic Operators *4.1*
4.3 Relational Operators *4.4*
4.4 Logical Operators *4.5*
4.5 Assignment Operators *4.6*
4.6 Increment And Decrement Operators *4.8*
4.7 Conditional Operator *4.9*
4.8 Bitwise Operators *4.9*
4.9 Special Operators *4.10*
4.10 Arithmetic Expressions *4.12*
4.11 Evaluation of Expressions *4.12*
4.12 Precedence of Arithmetic Operators *4.13*
4.13 Some Computational Problems *4.15*
4.14 Type Conversions In Expressions *4.16*
4.15 Operator Precedence And Associativity *4.20*
4.16 Mathematical Functions *4.22*
Just Remember 4.23
Review Questions 4.23
Programming Exercises 4.27
Debugging Exercises 4.29

5. Input–Output Functions **5.1–5.27**

5.1 Introduction *5.1*
5.2 Reading a Character *5.2*
5.3 Writing a Character *5.5*
5.4 Formatted Input *5.6*
5.5 Formatted Output *5.15*
Just Remember *5.22*
Review Questions *5.23*
Programming Exercises *5.25*

6. Control Structures: Decision Making and Branching **6.1–6.35**

6.1 Introduction *6.1*
6.2 Decision Making with if Statement *6.1*
6.3 Simple if Statement *6.2*
6.4 The **if.....Else** Statement *6.6*
6.5 Nesting of **if....Else** Statements *6.9*
6.6 The **Else if** Ladder *6.12*
6.7 The **Switch** Statement *6.15*
6.8 **The ? :** Operator *6.19*
6.9 The **Goto** Statement *6.21*
Just Remember *6.24*
Review Questions *6.25*
Programming Exercises *6.29*
Debugging Exercises *6.32*

7. Control Structures: Decision Making and Looping **7.1–7.37**

7.1 Introduction *7.1*
7.2 The While Statement *7.3*
7.3 The Do Statement *7.5*
7.4 The for Statement *7.8*
7.5 Jumps in Loops *7.20*
7.6 Concise Test Expressions *7.27*
Just Remember *7.28*
Review Questions *7.28*
Programming Exercises *7.32*
Debugging Exercises *7.35*

8. Arrays **8.1–8.33**

8.1 Introduction *8.1*
8.2 One-dimensional Arrays *8.3*
8.3 Declaration of One-dimensional Arrays *8.4*
8.4 Initialization of One-dimensional Arrays *8.6*
8.5 Two-dimensional Arrays *8.10*

8.6 Initializing Two-dimensional Arrays *8.21*
8.7 Multi-dimensional Arrays *8.24*
8.8 Dynamic Arrays *8.25*
Just Remember 8.26
Review Questions 8.26
Programming Exercises 8.29
Debugging Exercises 8.32

9. Strings 9.1–9.30

9.1 Introduction *9.1*
9.2 Declaring and Initializing String Variables *9.2*
9.3 Reading Strings from Terminal *9.3*
9.4 Writing Strings to Screen *9.8*
9.5 Arithmetic Operations on Characters *9.12*
9.6 Putting Strings Together *9.13*
9.7 Comparison of Two Strings *9.15*
9.8 String-Handling Functions *9.15*
9.9 Table of Strings *9.21*
9.10 Other Features of Strings *9.22*
Just Remember 9.23
Review Questions 9.23
Programming Exercises 9.26
Debugging Exercises 9.27

10. User–Defined Functions and Recursion 10.1–10.49

10.1 Introduction *10.1*
10.2 Need for User-defined Functions *10.1*
10.3 A Multi-function Program *10.2*
10.4 Elements of User-defined Functions *10.5*
10.5 Definition of Functions *10.5*
10.6 Return Values and Their Types *10.8*
10.7 Function Calls *10.9*
10.8 Function Declaration *10.11*
10.9 Category of Functions *10.12*
10.10 No Arguments and no Return Values *10.12*
10.11 Arguments but no Return Values *10.14*
10.12 Arguments with Return Values *10.18*
10.13 No Arguments but Returns a Value *10.21*
10.14 Functions that Return Multiple Values *10.22*
10.15 Nesting of Functions *10.23*
10.16 Recursion *10.24*
10.17 Passing Arrays to Functions *10.25*
10.18 Passing Strings to Functions *10.30*
10.19 The Scope, Visibility and Lifetime of Variables *10.30*

Just Remember *10.40*
Review Questions *10.41*
Programming Exercises *10.45*
Debugging Exercises *10.47*

11. Pointers 11.1–11.32

11.1 Introduction *11.1*
11.2 Understanding Pointers *11.2*
11.3 Accessing the Address of a Variable *11.4*
11.4 Declaring Pointer Variables *11.5*
11.5 Initialization of Pointer Variables *11.6*
11.6 Accessing a Variable through its Pointer *11.7*
11.7 Chain of Pointers *11.9*
11.8 Pointer Expressions *11.10*
11.9 Pointer Increments and Scale Factor *11.11*
11.10 Pointers and Arrays *11.12*
11.11 Pointers and Character Strings *11.15*
11.12 Array of Pointers *11.17*
11.13 Pointers as Function Arguments *11.18*
11.14 Functions Returning Pointers *11.21*
11.15 Pointers to Functions *11.21*
11.16 Pointers and Structures *11.24*
11.17 Troubles with Pointers *11.26*
Just Remember *11.27*
Review Questions *11.27*
Programming Exercises *11.30*
Debugging Exercises *11.31*

12. Structures and Unions 12.1–12.32

12.1 Introduction *12.1*
12.2 Defining a Structure *12.1*
12.3 Declaring Structure Variables *12.3*
12.4 Accessing Structure Members *12.4*
12.5 Structure Initialization *12.6*
12.6 Copying and Comparing Structure Variables *12.7*
12.7 Operations on Individual Members *12.9*
12.8 Arrays of Structures *12.9*
12.9 Arrays within Structures *12.12*
12.10 Structures within Structures *12.13*
12.11 Structures and Functions *12.15*
12.12 Pointers and Structures *12.18*
12.13 Unions *12.20*
12.14 Size of Structures *12.22*
12.15 Bit Fields *12.22*

Just Remember *12.24*
Review Questions *12.25*
Programming Exercises *12.29*
Debugging Exercises *12.31*

UNIT 3

13. Principles of Object-Oriented Programming **13.1–13.15**

13.1 Software Crisis *13.1*
13.2 Software Evolution *13.3*
13.3 A Look at Procedure-oriented Programming *13.4*
13.4 Object-oriented Programming Paradigm *13.5*
13.5 Basic Concepts of Object-oriented Programming *13.6*
13.6 Benefits of OOP *13.10*
13.7 Object-oriented Languages *13.11*
13.8 Applications of OOP *13.13*
Just Remember *13.13*
Review Questions *13.14*

14. Beginning with C++ **14.1–14.13**

14.1 What is C++? *14.1*
14.2 Applications of C++ *14.2*
14.3 A Simple C++ Program *14.2*
14.4 More C++ Statements *14.6*
14.5 An Example with Class *14.8*
14.6 Structure of C++ Program *14.9*
14.7 Creating the Source File *14.10*
14.8 Compiling and Linking *14.10*
Just Remember *14.11*
Review Questions *14.11*
Programming Exercises *14.12*
Debugging Exercises *14.12*

15. Tokens, Expressions and Control Structures **15.1–15.33**

15.1 Introduction *15.1*
15.2 Tokens *15.1*
15.3 Keywords *15.1*
15.4 Identifiers and Constants *15.2*
15.5 Basic Data Types *15.3*
15.6 User-defined Data Types *15.5*
15.7 Derived Data Types *15.7*
15.8 Symbolic Constants *15.8*
15.9 Type Compatibility *15.9*
15.10 Declaration of Variables *15.9*

15.11 Dynamic Initialization of Variables *15.10*
15.12 Reference Variables *15.10*
15.13 Operators in C++ *15.12*
15.14 Scope Resolution Operator 15.13
15.15 Member Dereferencing Operators *15.15*
15.16 Memory Management Operators *15.15*
15.17 Manipulators *15.17*
15.18 Type Cast Operator *15.18*
15.19 Expressions and Their Types *15.19*
15.20 Special Assignment Expressions *15.21*
15.21 Implicit Conversions *15.21*
15.22 Operator Overloading *15.23*
15.23 Operator Precedence *15.23*
15.24 Control Structures *15.24*
Just Remember 15.28
Review Questions 15.29
Programming Exercises 15.30
Debugging Exercises 15.32

16. Functions in C++ 16.10–16.16

16.1 Introduction *16.1*
16.2 The Main Function *16.2*
16.3 Function Prototyping *16.3*
16.4 Call by Reference *16.4*
16.5 Return by Reference *16.5*
16.6 Inline Functions *16.5*
16.7 Default Arguments *16.7*
16.8 **const** Arguments *16.9*
16.9 Function Overloading *16.9*
16.10 Friend and Virtual Functions *16.12*
16.11 Math Library Functions *16.12*
Just Remember 16.12
Review Questions 16.13
Programming Exercises 16.14
Debugging Exercises 16.15

17. Classes and Objects 17.1–17.39

17.1 Introduction *17.1*
17.2 C Structures Revisited *17.1*
17.3 Specifying a Class *17.3*
17.4 Defining Member Functions *17.6*
17.5 A C++ Program with Class *17.8*
17.6 Making an Outside Function Inline *17.10*
17.7 Nesting of Member Functions *17.10*

17.8 Private Member Functions *17.11*
17.9 Arrays within a Class *17.12*
17.10 Memory Allocation for Objects *17.16*
17.11 Static Data Members *17.17*
17.12 Static Member Functions *17.19*
17.13 Arrays of Objects *17.20*
17.14 Objects as Function Arguments *17.23*
17.15 Friendly Functions *17.25*
17.16 Returning Objects *17.30*
17.17 **const** Member Functions *17.31*
17.18 Pointers to Members *17.31*
17.19 Local Classes *17.33*
Just Remember *17.34*
Review Questions *17.34*
Programming Exercises *17.35*
Debugging Exercises *17.36*

18. Constructors and Destructors **18.1–18.23**

18.1 Introduction *18.1*
18.2 Constructors *18.2*
18.3 Parameterized Constructors *18.3*
18.4 Multiple Constructors in a Class *18.6*
18.5 Constructors with Default Arguments *18.8*
18.6 Dynamic Initialization of Objects *18.9*
18.7 Copy Constructor *18.11*
18.8 Dynamic Constructors *18.13*
18.9 Constructing Two-dimensional Arrays *18.14*
18.10 **const** Objects *18.16*
18.11 Destructors *18.16*
Just Remember *18.19*
Review Questions *18.19*
Programming Exercises *18.19*
Debugging Exercises *18.20*

GTU Solved Question Paper–June 2010 **Q.1–Q.16**
GTU Solved Question Paper–Remedial Nov./Dec. 2010 **Q.17–Q.29**
GTU Solved Question Paper–Jan 2011 **Q.30–Q.38**

Preface

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

Rightly so, many institutions and universities in India have introduced a subject covering the fundamentals of computers and programming in C and C++ at the undergraduate and diploma levels of arts, science and engineering disciplines.

This book is designed for the students taking the first-semester paper **Computer Programming and Utilization (CC 110003)** common to all BTech courses in **Gujarat Technological University (GTU).**

Highlights of the Book

The book is for the programmer who wishes to know all about computer fundamentals, C and C++ languages and object-oriented programming. It explains in a simple and easy-to-understand style the what, why and how of C Programming and object-oriented programming with C++.

It ensures that the reader will go from knowing nothing about programming to having a strong grasp of C, C++ and object-oriented programming with the support of numerous programs and exercises for practice. The book will prove be a self-study guide.

Salient Features

- **Independent, up-to-date** and **complete coverage** of entire syllabus of GTU
- **Precise theory** presented in lucid language
- **Codes with comments** provided throughout the book to illustrate the use of various features of the language
- Excellent coverage of **flowcharts** and **algorithms**
- **Supplementary information and notes** that complement but stand apart from the text are included in special boxes
- **Just Remember** given at the end of each chapter for quick recap of the concepts
- Includes latest **Solved Question Paper** of GTU (**June 2010**, **Remedial Nov./Dec. 2010**, **Jan 2011**)
- Rich pedagogy:
  - **258 Review Questions** provide ample opportunities to test the conceptual understanding of features

- **197 Programming Exercises and 68 Debugging Exercises** stimulate interest to practice programming applications
- **71 Fill in the Blanks** and **206 True/False type Questions** serve as effective assessment techniques

Chapter Organisation

The book has been divided into 18 chapters. **Chapter 1** gives an overview of computers, their development, characteristics and evolution. Next, the fundamentals of C are introduced in **Chapter 2. Chapter 3** explains data types in C. Operators and their hierarchy, and the concept of header files and I/O operations are taken up in **Chapters 4 and 5** respectively.

Control structures—decision making, branching and looping—are covered subsequently in **Chapters 6 and 7**. Arrays in ANSI C are presented in **Chapter 8. Chapter 9** deals with strings. Functions and recursion forms the subject of discussion in **Chapter 10**, while pointers, structures and unions are covered in **Chapters 11 and 12** respectively.

The principles of object-oriented programming and C++ are introduced in **Chapter 13**, while **Chapter 14** guides the reader on developing elementary C++ programs. Tokens, instructions and control structures are taken up in **Chapter 15**, while functions in C++, classes, objects, constructors and destructors are covered subsequently in **Chapters 16 to 18**.

The book provides numerous examples, illustrations and complete programs. The sample programs are meant to be both simple and educational. Wherever necessary, pictorial descriptions of concepts are included to improve clarity and facilitate better understanding. The book also presents the concept of oriented approach and discusses briefly the important elements of object programming.

Acknowledgements

I would like to thank all those who provided me with valuable feedback and inputs during the preparation of this book, and especially those at Tata McGraw Hill Education, without whose help and cooperation, this book would not have had a timely release. Special thanks are also due to all my teacher friends and students for their encouragement. I hope everyone who desires to be a part of the next generation of computing will find this book interesting and useful. Further suggestions for improvement will always be welcome.

— E Balagurusamy

Publisher's Note

Constructive suggestions and criticism always go a long way in enhancing any endeavour. We request all readers to email us their valuable comments/views/feedback for the betterment of the book at **tmh.corefeedback@gmail.com**, mentioning the title and author name in the subject line. Also, please feel free to report any piracy of the book spotted by you.

Roadmap to the Syllabus

Computer Programming and Utilization

Unit 1 Fundamentals of Computers

Introduction; Basic block diagram and functions of various components of a computer; Concept of hardware and software;
Concept of basic types of software; Introduction to programming and programming languages; Flow charts and algorithms

GO TO

Chapter 1 Introduction to Computers and Programming

Unit 2 Introduction to C Language

Fundamentals of C—Data types in 'C'; Operators and their hierarchy; Concept of header files, I/O functions; Control structures in C; Looping structures in C; Arrays and strings

GO TO

Chapter 2 Fundamentals of C
Chapter 3 Data Types in C
Chapter 4 Operations and their Hierarchy
Chapter 5 Concept of Header Files and I/O Operation
Chapter 6 Control Structures—Decision Making and Branching
Chapter 7 Control Structures—Decision Making and Looping
Chapter 8 Arrays in Ansi C
Chapter 9 Strings

Functions and Recursion—Concepts of functions with various types of parameters; Various types of parameter-passing mechanisms; Recursive functions and implementation of these concepts in C

GO TO

Chapter 10 Functions and Recursion

Pointers and Structures—Concepts of pointers and simple programs using pointers; Introduction to simple structure and its implementations

GO TO

Chapter 11 Pointers
Chapter 12 Structures and Unions

Unit 3 Introduction To C++ Language

Principles of Object Oriented Programming—Comparison with procedural languages; Tokens, expressions and control structures; Introduction to functions, classes, objects, constructors, destructors in C++

Classes and Objects—Declaring classes; Defining member functions; Making an outside function inline; Nesting of member functions; Private member functions; Arrays with in a class; Memory allocation of objects; Static data members; Static member functions; Arrays of objects; Objects as function arguments; Friend function, Returning of objects; const member function

GO TO

Chapter 13 Principles of Object Oriented Programming
Chapter 14 Beginning With C++
Chapter 15 Tokens, Expressions and Control Structures
Chapter 16 Functions In C++
Chapter 17 Classes and Objects
Chapter 18 Constructors and Destructors



1 Introduction to Computer and Programming

1.1 Introduction

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

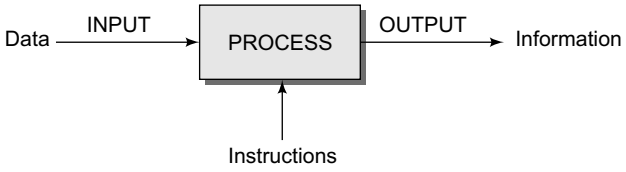


Fig. 1.1 *Input–process–output concept*

The spurt of innovations and inventions in computer technology during the last few decades has led to the development of a variety of computers. They are so versatile that they have become indispensable to engineers, scientists, business executives, managers, administrators, accountants, teachers and students. They have strengthened man's powers in numerical computations and information processing.

Modern computers possess certain characteristics and abilities peculiar to them. They can:

(i) perform complex and repetitive calculations *rapidly* and *accurately,*
(ii) *store large amounts of data* and information for subsequent manipulations,
(iii) *hold a program of a model* which can be explored in many different ways,
(iv) *compare* items and make decisions,
(v) provide information to the user in many different forms,
(vi) *automatically correct* or modify the parameters of a system under control,

(vii) *draw and print graphs*,
(viii) *converse* with users *interactively*, and
(ix) receive and *display audio and video signals.*

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

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

Engineers and scientists make use of the high-speed computing capability of computers to solve complex mathematical models and design problems. Many calculations that were previously beyond contemplation have now become possible. Many of the technological achievements such as landing on the moon would not have been possible without computers.

The areas of computer applications are too numerous to mention. Computers have become an integral part of man's everyday life. They continue to grow and open new horizons of discovery and application such as the electronic office, electronic commerce, and the home computer center.

The microelectronics revolution has placed enormous computational power within the reach of not only every organisation but also individual professionals and businessmen. However, it must be remembered that computers are machines created and managed by human beings. A computer has no brain of its own. Anything it does is the result of human instructions. It is an obedient slave which carries out the master's instructions as long as it can understand them, no matter whether they are right or wrong. A computer has no common sense.

1.2 Basic Anatomy of a Computer System

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

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

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

All the hardware components interact with each other as well as with the software. Similarly, the different types of software interact with each other and with the hardware components. The interaction between various hardware components is illustrated in Fig. 1.2.

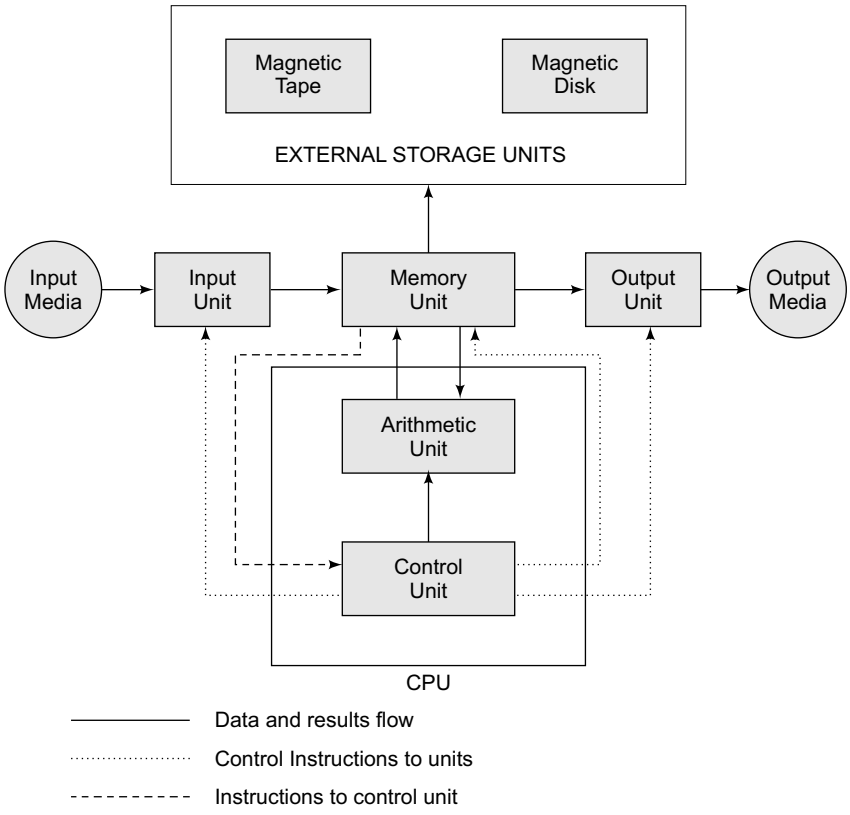


Fig. 1.2 *Interaction among hardware components*

1.3 Input Devices

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

- Keyboard
- Mouse
- Scanner

1.3.1 Keyboard

A standard keyboard includes alphanumeric keys, function keys, modifier keys, cursor movement keys, spacebar, escape key, numeric keypad, and some special keys, such as Page Up, Page Down, Home, Insert, Delete and End. The alphanumeric keys include the number keys and the alphabet keys. The function keys are the keys that help perform a specific task such as searching a file or refreshing a Web page. The modifier keys such as Shift and Control keys modify the casing style of a character or symbol. The cursor movement keys include up, down, left and right keys and are used to modify the direction of the cursor on the screen. The spacebar key shifts the cursor to the right by one position. The numeric keypad uses separate keypads for numbers and mathematical operators. A keyboard is shown in Fig. 1.3.

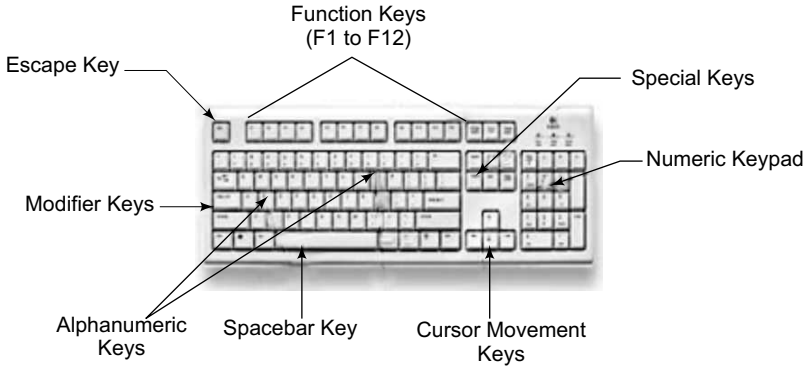


Fig. 1.3 *Keyboard*

1.3.2 Mouse

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

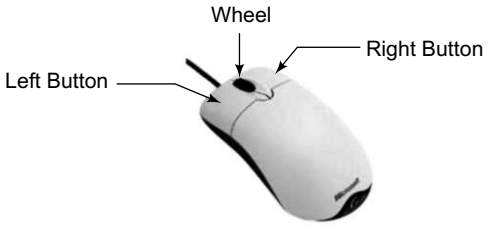


Fig. 1.4 *Mouse*

The mouse consists of two buttons, a wheel at the top and a ball at the bottom of the mouse. When the ball moves, the cursor on the screen moves in the direction in which the ball rotates. The left button of the mouse is used to select an element and the right button, when clicked, displays the special options such as **open** and **explore** and **shortcut** menus. The wheel is used to scroll down in a document or a Web page. A mouse is shown in Fig. 1.4.

1.3.3 Scanner

A scanner is an input device that converts documents and images as the digitized images understandable by the computer system. The digitized images can be produced as black and white images, gray images, or colored images. In case of colored images, an image is considered as a collection of dots with each dot representing a combination of red, green, and blue colors, varying in proportions. The proportions



Fig. 1.5 *Scanner*

of red, green, and blue colors assigned to a dot are together called as *color description*. The scanner uses the color description of the dots to produce a digitized image. Fig. 1.5 shows a scanner.

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

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

1.4 Output Devices

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

- Monitor
- Printer
- Speaker
- Plotter

1.4.1 Monitor

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

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

The inner side of the screen of the CRT contains the red, green, and blue phosphors. When a beam of electrons strike the screen, the beam strikes the red, green and blue phosphors on the screen and irradiates it to produce the image. The process repeats itself for a change in the image, thus refreshing the changing image. To change the color displayed by the monitor, the intensity of the beam striking the screen is varied. If the rate at which the screen gets refreshed is large, then the screen starts flickering, when the images are refreshed.

The LCD monitor is a thin display device that consists of a number of color or monochrome pixels arrayed in front of a light source or reflector. LCD monitors consume a very small amount of electric power.

A monitor can be characterized by its monitor size and resolution. The monitor size is the length of the screen that is measured diagonally. The resolution of the screen is expressed as the number of picture elements or pixels of the screen. The resolution of the monitor is also called the dot pitch. The monitor with a higher resolution produces a clearer image.

1.4.2 Printer

The printer is an output device that transfers the text displayed on the screen, onto paper sheets that can be used by the end user. The various types of printers used in the market are generally categorized as dot matrix printers, inkjet printers, and laser printers. Dot matrix printers are commonly used in low quality and high volume applications like invoice printing, cash registers, etc. However, inkjet printers are slower than dot matrix printers and generate high quality photographic prints. Since laser printers consist of microprocessor, ROM and RAM, they can produce high quality prints in quicker time without being connected to a computer.

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

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

- **Dot matrix printers** — Dot matrix printers are impact printers that use perforated sheet to print the text. The process to print a text involves striking a pin against a ribbon to produce its impression on the paper. As the striking motion of the pins help in making carbon copies of a text, dot matrix printers are used to produce multiple copies of a print out.
- **Inkjet printers** — Inkjet printers are slower than dot matrix printers and are used to generate high quality photographic prints. Inkjet printers are not impact printers. The ink cartridges are attached to the printer head that moves horizontally, from left to right. The print out is developed as the ink of the cartridges is sprayed onto the paper. The ink in the inkjet is heated to create a bubble. The bubble bursts out at high pressure, emitting a jet of the ink on the paper thus producing images.
- **Laser printers** — The laser printer may or may not be connected to a computer, to generate an output. These printers consist of a microprocessor, ROM and RAM, which can be used to store the textual information. The printer uses a cylindrical drum, a toner and the laser beam. The toner stores the ink that is used in generating the output. The fonts used for printing in a laser printer are stored in the ROM or in the cartridges that are attached to the printer. The laser printers are available as gray scale, black and white or color models. To print high quality pages that are graphic intensive, laser printers use the PageMaker software.

1.4.3 Speaker

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

1.4.4 Plotter

The plotter is another commonly used output device that is connected to a computer to print large documents, such as engineering or constructional drawings. Plotters use multiple ink pens or inkjets with color cartridges for printing. A computer transmits binary signals to all the print heads of the plotter. Each binary signal contains the coordinates of where a print head needs to be positioned for printing. Plotters are classified on the basis of their performance, as follows:

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

1.5 Software

In the field of computer science, software is defined as a computer program, which includes logical instructions used for performing a particular task on a computer system using hardware components. The following are the two major categories of software under which different types of computer programs can be classified:

- System software
- Application software

Figure 1.6 shows the relationship among hardware, software, and user.

The figure shows a layered architecture, which represents different components of a computer such as hardware, system software, application software, and user in a hierarchical manner.

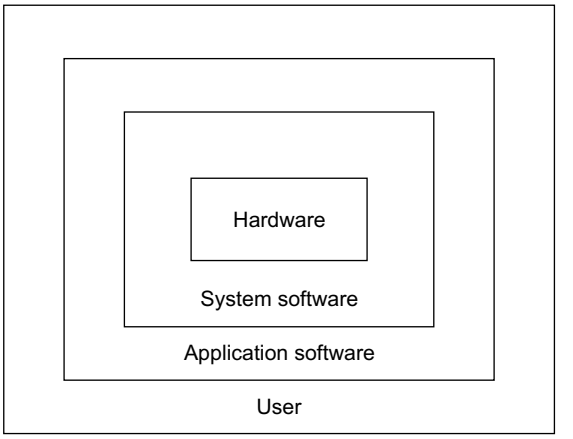


Fig. 1.6 *Relationship among hardware, software, and user*

1.5.1 System Software

System software refers to a computer program that manages and controls hardware components of a computer system. In other words, the system software is responsible for handling the functioning of the computer hardware. The system software is also responsible for the proper functioning of the application software on a computer system. The system software includes general programs, which are written to provide an environment for developing new application software using programming languages. In computer science, there are several types of system software, such as operating systems and utility programs. The operating system is the primary system software, which controls the hardware and software resources of a computer system. It also performs various operations, such as memory allocation, instruction processing, and file management. The most commonly used operating systems are MS DOS, MS Windows, and UNIX. The following are the various functions of system software:

- Process management
- Memory management
- Secondary storage management
- I/O system management
- File management

1.5.2 Application Software

Application software is a computer program that is executed on the system software. It is designed and developed for performing specific tasks and is also known as end-user program. Application software is unable to run without the system software, such as operating system and utility programs. It includes several applications, such as word-processing and spreadsheet. The word-processing application helps in creating and editing a document. Using this application software, we can also format and print the document. For word-processing, many applications are available, such as WordStar, WordPerfect, and Open-source. The most commonly used word-processing application is MS Word, which is a part of the MS Office suite. Spreadsheet application helps in creating a customized ledger, which has number

of columns and rows for entering the data values. The most commonly used spreadsheet application is MS Excel, which is also a part of MS Office suite. It helps in storing and maintaining a database in a structural manner.

1.6 Hardware

The physical devices that make up the computer are called *Hardware*. The hardware units are responsible for entering, storing and processing the given data and then displaying the output to the users. The basic hardware units of a general purpose computer are keyboard, mouse, memory, CPU, monitor and printer. Among these hardware units, keyboard and mouse are used to input data into the computer, memory is used to store the entered data, CPU is used to process the entered data and monitor and printer are used to display the processed data to the users.

CPU is the main component inside the computer that is responsible for performing various operations and also for managing the input and output devices. It includes two components for its functioning, *Arithmetic Logic Unit* (ALU) and *Control Unit* (CU). ALU is used to perform the arithmetic operations such as addition, subtraction, etc. and logic operations such as AND, OR, etc. on the data obtained from the memory. CU is used to control the activities related to the input and output devices. It obtains the instructions from the memory, decodes them and then, executes them so as to deliver output to the users.

The XNOR gate can be logically expressed as:

$$\overline{A \oplus B}$$

Table 1.1 shows the truth table of XNOR gate.

Table 1.1 ***The Truth Table of XNOR Gate***

Input A	Input B	Output Y
0	0	1
0	1	0
1	0	0
1	1	1

1.7 Programming Languages

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

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

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

Three levels of programming languages are available. They are:

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

1.7.1 Machine Language

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

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

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

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

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

1.7.2 Assembly Language

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

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

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

An assembly language program consists of a series of instructions and mnemonics that correspond to a stream of executable instructions. An assembly language instruction consists of a mnemonic code followed by zero or more operands. The mnemonic code is called the *operation code* or *opcode*, which specifies the operation to be performed on the given arguments. Consider the following machine code:
10110000 01100001
Its equivalent assembly language representation is:
mov al, 061h

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

```
ORG 500          /Origin of program is location 500
LDA SUB          /Load subtrahend to AC
```

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

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

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

1.7.3 High-Level Languages

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

```
LOAD A
ADD B
STORE C
```

Using FORTRAN, the above code can be represented as:

$C = A + B$

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

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

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

Procedure-oriented Languages

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

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

Problem-oriented Languages

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

Natural Languages

Natural languages are designed to make a computer to behave like an expert and solve problems. The programmer just needs to specify the problem and the constraints for problem-solving. Natural languages such as LISP and PROLOG are mainly used to develop artificial intelligence and expert systems. These languages are widely known as *fifth generation* languages.

1.8 Programming Environment

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

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

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

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

1.9 Introduction to the Design and Implementation of Correct, Efficient and Maintainable Programs

The design and development of a correct, efficient, and maintainable program depends on the approach followed by the programmer. A programmer should follow standard methodologies throughout the life cycle of program development. The entire program development process is divided into a number of phases, with each phase serving a definite purpose. Also, the output of one phase acts as an input for the next phase. Let us now understand these standard set of phases in the program development process:

1. **Analysis phase** As the name suggests, the first phase of program development involves analyzing the problem in order to ascertain the objectives that the program is supposed to meet. All the identified requirements are documented so as to avoid any doubts or uncertainties pertaining to the functionality of the program. This phase also emphasizes on determining the input and output values of the program.
2. **Designing phase** This phase involves making the plan of action before actually starting the development work. The plan is made on the basis of the program specifications identified in the previous phase. Different programs require different designing patterns depending on the program specifications. Thus, this phase helps in framing the core structure of the program. In addition, the designing phase has an added advantage of modularity. It basically helps to break the program into small modules or chunks. This breaking of the large program into smaller chunks results in the development of a well-organized program. Furthermore, it gives the programmer liberty of planning and creating the algorithm for each module separately.
3. **Development phase** This phase involves writing the instructions or code for the program on the basis of the design document created in the previous phase. The choice of the programming language in which the program will be developed is made on the basis of the type of program. For example, if it is a system program, then it is better to choose C language instead of Visual Basic (VB), which is more suited for applications programming.
4. **Implementation and testing** In this stage, the developed program is implemented in its target environment and its key parameters are closely observed in order to ensure that the program runs correctly. Apart from ensuring the correct functioning of the program this phase primarily focuses on identifying the hidden bugs in the program. No matter how many preventive measures are taken in the development phase there is always the possibility of prevalence of hidden bugs in a program. Thus, to identify such bugs a program needs to be tested using large number of varied input values. Once identified the bugs are removed with the help of software patches.

1.10 Structured Programming

Structured programming is a subset of one of the key programming paradigms, i.e., procedural programming. It helps in making a program easily understandable and debuggable. A program that is not based on the structured programming approach is very difficult to maintain, debug and understand.

Structured programming approach mainly focuses on the order of execution of the statements within a program. It suggests the use of sequential execution of statements in a program. Thus, structured programming approach suggests the use of mainly three types of control structures—sequential, repetitive and selective. Further, it suggests avoiding the use of **goto**, **break** and **continue** statements in a program as all these are unconditional branch statements.

1.11 Problem Solving

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

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

1. **Developing the algorithm** An algorithm is a sequence of steps written in the form of English phrases that specify the tasks that are performed while solving a problem. It involves identifying the variable names and types that would be used for solving the problem.
2. **Drawing the flowchart** A flowchart is the graphical representation of the flow of control and logic in the solution of a problem. The flowchart is a pictorial representation of an algorithm.

1.12 Algorithms

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

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

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

Fig. 1.7

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

1.12.1 Characteristics of Algorithms

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

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

1.12.2 Advantages of Algorithms

Some of the key advantages of algorithms are the following:

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

1.12.3 Disadvantages of Algorithms

Apart from the advantages, algorithms also posses certain limitations, which are the following:

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

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

Solution

```
Algorithm
  Step 1 - Start
  Step 2 - Accept a number from the user (num)
  Step 3 - Initialize looping counter i = 2
```

Contd.

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

Fig. 1.8

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

Solution

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

Fig. 1.9

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

Solution

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

Fig. 1.10

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

Solution

```
Algorithm
     Step 1 - Start
     Step 2 – Accept a number from the user (num)
     Step 3 – If remainder of num divided by 2 (num/2) is Zero then goto
     Step 4 else goto Step 5
```

Contd.

```
Step 4 – Display "num is an even number" and goto Step 6
Step 5 – Display "num is an odd number"
Step 6 - Stop
```

Fig. 1.11

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

Solution

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

Fig. 1.12

1.13 Flowcharts

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

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

- **Start and end** It is represented by an oval or a rounded rectangle. It represents the starting and the ending of a process. Every process starts and ends at some point so a flowchart always contains one start as well as one end symbol. Figure 1.13 shows the start and the end symbols used in a flowchart.

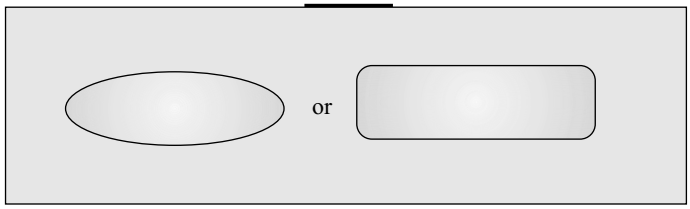


Fig. 1.13 *Start and end symbol*

- **Input or output** It is represented by a parallelogram. It represents the inputs given by the user to the process and the outputs given by the process to the user. Figure 1.14 shows the input or output symbol.



Fig. 1.14 *Input or output symbol.*

- **Action or process** It is represented by a rectangle. It represents the actions, logics and calculations taking place in a process. Figure 1.15 shows the action or process symbol.



Fig. 1.15 *Action or process symbol.*

- **Decision or condition** It is represented by a rhombus or a diamond shape. It represents the condition or the decision-making step in a flowchart. The result of the decision is a Boolean value, which is either true or false. Each of these values takes the flow of the program to a certain point, which is shown with the help of arrows. Figure 1.16 shows the decision or condition symbol.

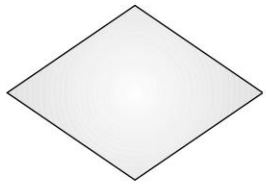


Fig. 1.16 *Decision or condition symbol.*

- **Arrow** It is represented by a directed line. It represents the flow of process and the sequence of steps in a flowchart. It guides the process about the direction and the sequence, which is to be followed while performing the various steps in the process. Figure 1.17 shows the arrow symbol.

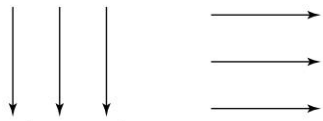


Fig. 1.17 *Arrow symbol.*

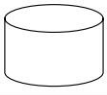
- **Connector** It is represented by a circle in a flowchart. It represents the continuation of the flow of steps when a flowchart continues to the next page. A character such as an alphabet (a to z) or a symbol (α, β or χ), etc. can be placed in the circle at the position where the flow is broken and the same character is also placed in the circle at the position from where the flowchart continues. Figure 1.18 shows the connector symbol.



Fig. 1.18 *Connector symbol.*

In addition to the above-mentioned basic flowchart symbols, certain other symbols are also used inside flowcharts for depicting advanced operations. Table 1.2 shows some of these advanced flowchart symbols:

Table 1.2

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

Contd.

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

In order to understand how a flowchart represents the flow of information; consider an example of a flowchart for adding two numbers, as shown in Fig. 1.19.

Flowchart

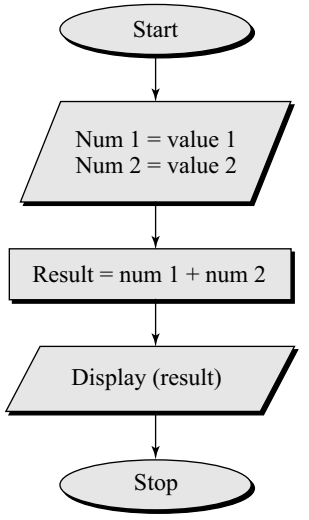


Fig. 1.19 *Flowchart for addition of two numbers.*

1.13.1 Flowchart Design Rules

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

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

1.13.2 Advantages of Flowcharts

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

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

1.13.3 Disadvantages of Flowcharts

Flowcharts also have certain limitations, such as the following:

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

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

Solution

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

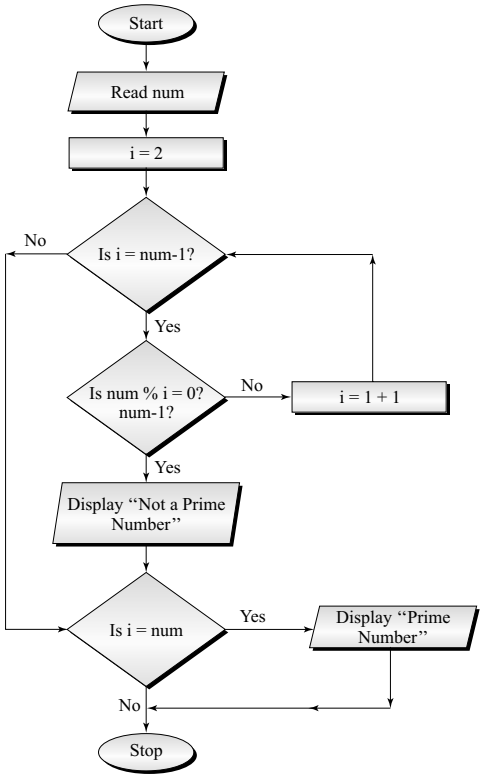


Fig. 1.20

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

Solution

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

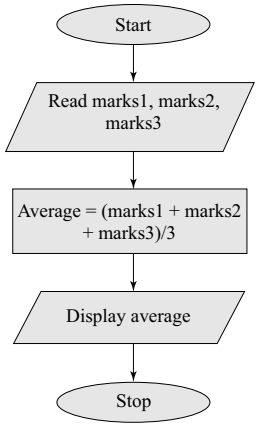


Fig. 1.21

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

Solution

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

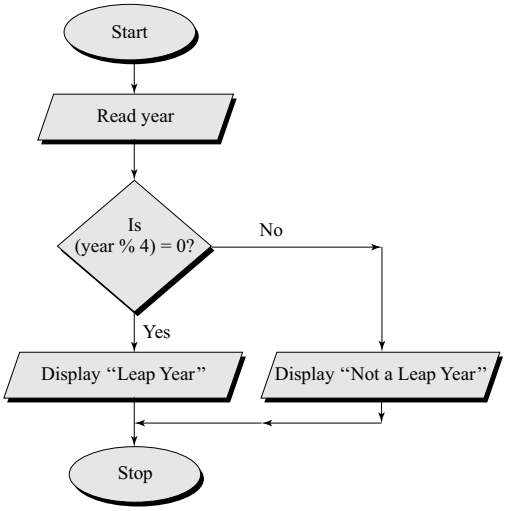


Fig. 1.22

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

Solution

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

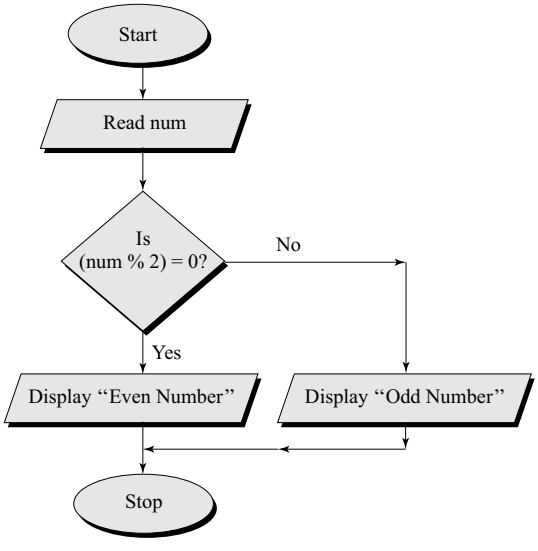


Fig. 1.23

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

Solution

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

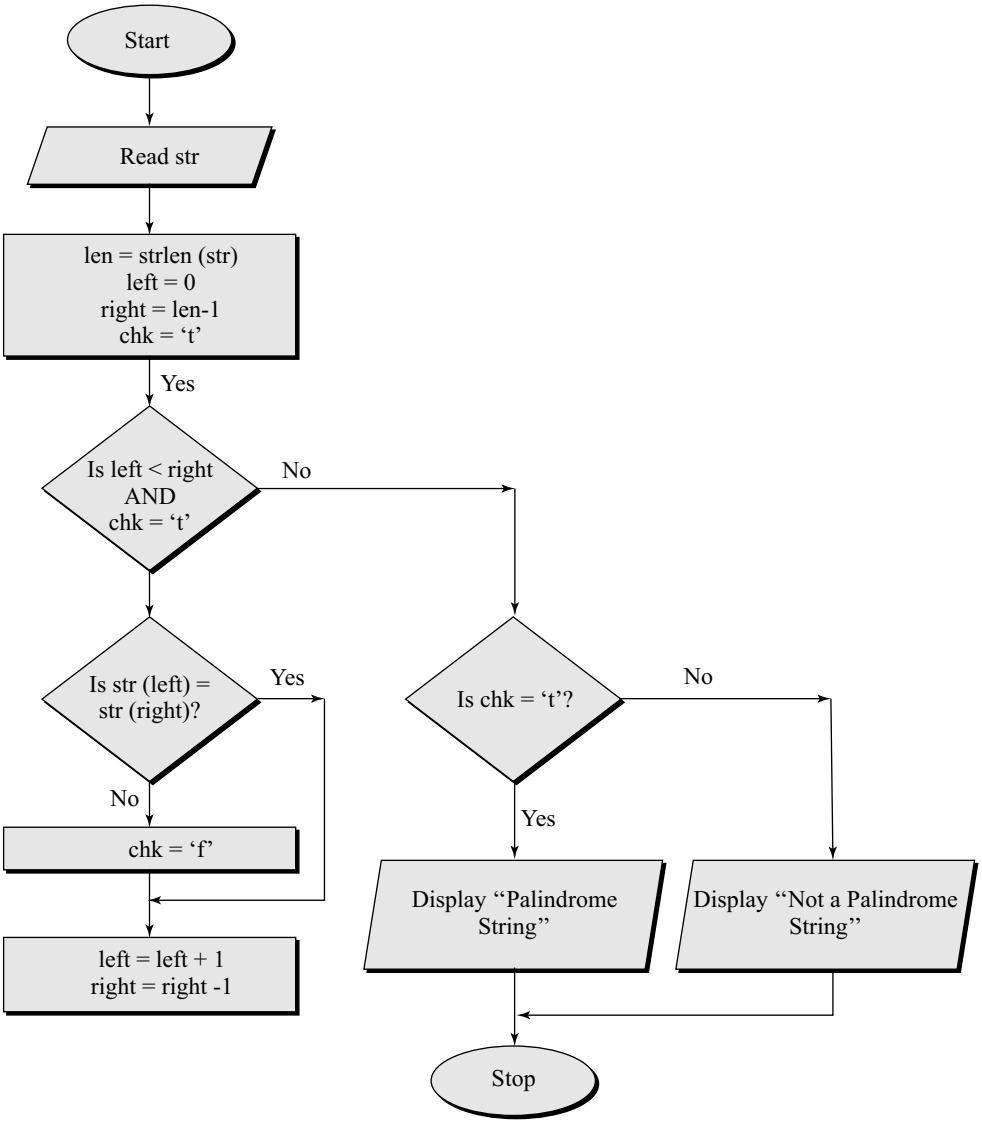


Fig. 1.24

Just Remember

- **Computer**: It is an electronic device that takes data and instructions as input from the user, processes the data, and generates useful information as an output.
- **Input devices**: Input devices accept the data from the end users on which the operations are to be performed.
- **Output devices**: Output devices are used for providing the output of a program that is obtained after performing the operations specified in a program.
- **CPU**: It is the heart of a computer that is used to process the data entered through the input device.
- **Memory**: It is used for storing the input data as well as the output of a program that is obtained after performing the operations in a program.
- **Scanner**: It is an input device that converts documents and images as the digitized images understandable by the computer system.
- **Motherboard**: It is a device used for connecting the CPU with the input and output devices.
- **Monitor**: It is an output device that produces visual displays generated by the computer.
- **Printer**: It is an output device that prints the computer generated information onto the paper sheets.
- **Speaker**: It is an electromechanical transducer that converts an electrical signal into sound.
- **Plotter**: It is an output device that is connected to a computer to print large documents, such as engineering and constructional drawings.
- **System software**: It refers to a computer program that manages and controls hardware components.
- **Application software**: It is a computer program that is designed and developed for performing specific utility tasks; it is also known as end-user program.
- **Operating System**: It is the system software that helps in managing the resources of a computer system. It also provides a platform for the application programs to run on the computer system.
- **Algorithm**: It is a complete, detailed, and precise step-by-step method for solving a problem independently.
- **Flowchart**: It is a pictorial representation of an algorithm depicting the flow of various steps.

Review Questions

1.1 State whether the following statements are *true* or *false*.

(i) Dot matrix printers are slower than inkjet printers and are used to generate high quality photographic prints.
(ii) Describing the process step by step is called as flowchart.
(iii) Algorithm involves very complex process.
(iv) When we break up a big task into smaller steps, what we actually do is to create an algorithm.
(v) Each step in an algorithm can be called as an instruction.
(vi) In general, the steps in an algorithm can be divided in five basic categories.
(vii) Avoiding a *goto* in a program, makes it a goto-less or top-down or structured.
(viii) goto-less programs are difficult to understand.
(ix) Sequence, selection and iteration form the building blocks for writing any algorithm.

1.2 What are input devices? Briefly explain some popular input devices.
1.3 What is the purpose of an output device? Explain various types of output devices.
1.4 What is assembly language? What are its main advantages?
1.5 What is high-level language? What are the different types of high-level languages?
1.6 What is a flow chart? How is it different from an algorithm?
1.7 What are the functions of a flow chart?
1.8 Point down the differences between an algorithm and a flowchart.
1.9 Write an algorithm for withdrawing Rs. 1000 from the bank.
1.10 Draw a flowchart for the above.

Multiple Choice Questions

1. Which of the following is used to perform computations on the entered data?
   (a) Memory (b) Processor
   (c) Input device (d) Output device
2. Which of the following is not an input device?
   (a) Plotter (b) Scanner
   (c) Keyboard (d) Mouse
3. Which of the following is not an output device?
   (a) Plotter (b) Scanner
   (c) Printer (d) Speaker
4. Which of the following is defined as a computer program for performing a particular task on the computer system?
   (a) Hardware (b) Software
   (c) Processor (d) Memory
5. In which of the following languages, the instructions are written in the form of 0s and 1s?
   (a) Assembly language (b) Programming language
   (c) High-level language (d) Machine language
6. Which one of the following is known as the 'language of the computer'?
   (a) Programming language (b) High-level language
   (c) Machine language (d) Assembly language
7. What are the three main categories of high-level language?
   (a) Low-level languages (b) Procedure oriented languages
   (c) Mechanical languages (d) Natural languages
   (e) Problem oriented languages
8. Any C program
   (a) Must contain at least one function (b) Need not contain any function
   (c) Needs input data (d) None of the above



2 Fundamentals of C

2.1 History of C

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

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the conscepts of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway. During 1970s, C had evolved into what is now known as *"traditional C"*.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.

During 1990's, C++, a language entirely based on C, underwent a number of improvements and changes and became an ANSI/ISO approved language in November 1977. C++ added several new features to C to make it not only a true object-oriented language but also a more versatile language. During the same period, Sun Microsystems of USA created a new language **Java** modelled on C and C++.

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig. 2.1.

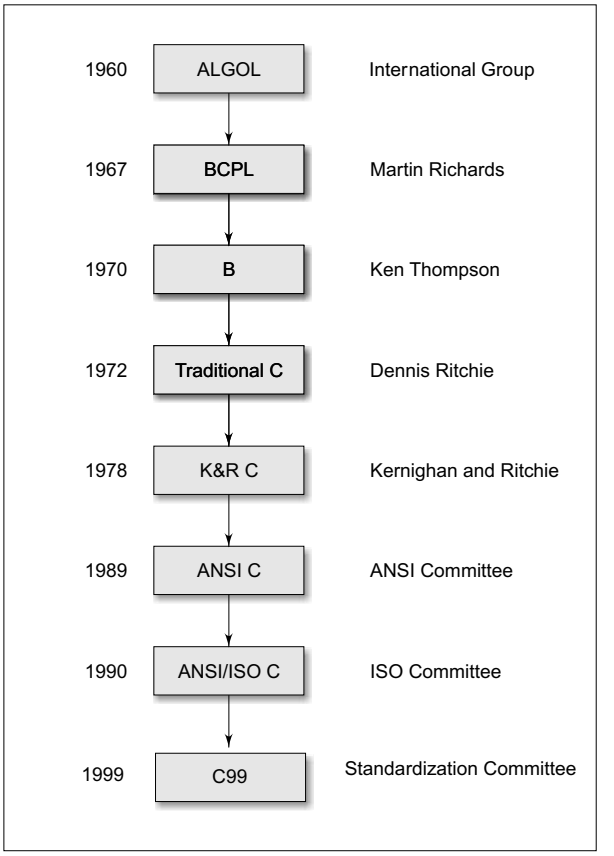


Fig. 2.1 *History of ANSI C*

Although C99 is an improved version, still many commonly available compilers do not support all of the new features incorporated in C99. We, therefore, discuss all the new features added by C99 in an appendix separately so that the readers who are interested can quickly refer to the new material and use them wherever possible.

2.2 Importance of C

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

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

There are only 32 keywords in ANSI C and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

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

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

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

2.3 Sample Program I: Printing a Message

Consider a very simple program given in Fig. 2.2.

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

Fig. 2.2 *A program to print one line of text*

This program when executed will produce the following output:

```
I see, I remember
```

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

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 10).

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

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

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

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

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

is not valid and therefore results in an error.

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

Let us now look at the **printf( )** function, the only executable statement of the program.

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

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

```
I see, I remember
```

Note that the print line ends with a semicolon. *Every statement in C should end with a semicolon (;) mark.*

Suppose we want to print the above quotation in two lines as

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

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

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

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

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

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

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

This is similar to the output of the program in Fig. 2.2. However, note that there is no space between , and I.

It is also possible to produce two or more lines of output by one **printf** statement with the use of newline character at appropriate places. For example, the statement

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

will output

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

while the statement

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

will print out

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

Note *Some authors recommend the inclusion of the statement*

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

at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions ***printf*** and ***scanf*** which have been defined as a part of the C language.

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

The above example that printed **I see, I remember** is one of the simplest programs. Figure 2.3 highlights the general format of such simple programs. All C programs need a **main** function.



Fig. 2.3 *Format of simple C programs*

The main Function

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

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

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

2.4 Sample Program 2: Adding Two Numbers

Consider another program, which performs addition on two numbers and displays the result. The complete program is shown in Fig. 2.4.

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

Fig. 2.4 *Program to add two numbers*

This program when executed will produce the following output:

```
100
106.10
```

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

The words **number** and **amount** are *variable names* that are used to store numeric data. The numeric data may be either in *integer* form or in *real* form. In C, *all variables should be declared* to tell the compiler what the ***variable names*** are and what ***type of data*** they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

```
int number;
float amount;
```

tell the compiler that **number** is an integer (**int**) and **amount** is a floating(**float**) point number. Declaration statements must appear at the beginning of the functions as shown in Fig. 2.4. All declaration statements end with a semicolon; C supports many other data types and they are discussed in detail in Chapter 3.

The words such as **int** and **float** are called the *keywords* and cannot be used as *variable* names. A list of keywords is given in Chapter 3.

Data is stored in a variable by *assigning* a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable **number** and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount.** The statements

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

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.
The next statement is an output statement that prints the value of **number**. The print statement

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

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

The last statement of the program

```
printf("%5.2f", amount);
```

prints out the value of **amount** in floating point format. The format specification *%*5.2*f* tells the compiler that the output must be in *floating point,* with five places in all and two places to the right of the decimal point.

2.5 Sample Program 3: Interest Calculation

The program in Fig. 2.5 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in Fig. 2.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = Value at start of year (1 + interest rate)

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement

```
amount = value ;
```

makes the value at the end of the *current* year as the value at start of the *next* year.

```
/*———————— INVESTMENT PROBLEM ————————*/
#define PERIOD   10
#define PRINCIPAL   5000.00
/*———————— MAIN PROGRAM BEGINS ————————*/
main()
{ /*———————— DECLARATION STATEMENTS ————————*/
          int year;
          float amount, value, inrate;
/*———————— ASSIGNMENT STATEMENTS ————————*/
         amount = PRINCIPAL;
         inrate = 0.11;
         year = 0;
/*———————— COMPUTATION STATEMENTS ————————*/
/*———————— COMPUTATION USING While LOOP ————————*/
         while(year <= PERIOD)
         {  printf("%2d  %8.2f\n",year, amount);
            value  = amount + inrate * amount;
                   year   = year + 1;
            amount= value;
         }
/*———————— while LOOP ENDS ————————*/
}
/*———————— PROGRAM ENDS ————————*/
```

Fig. 2.5 *Program for investment problem*

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

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

Fig. 2.6 *Output of the investment program*

The #define Directive

A **#define** is a preprocessor compiler directive and not a statement. Therefore **#define** lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names. **#define** instructions are usually placed at the beginning before the **main()** function. Symbolic constants are not declared in declaration section.

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

```
PRINCIPAL = 10000.00;
```

is illegal.

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

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

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

All computations and printing are accomplished in a **while** loop. **while** is a mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of **year** is less than or equal to the value of **PERIOD**, the four statements that follow **while** are executed. Note that these four statements are grouped by braces. We exit the loop when **year** becomes greater than **PERIOD**. The concept and types of loops are discussed in Chapter 7

C supports the basic four arithmetic operators (–, +, *, /) along with several others. They are discussed in Chapter 4.

2.6 Sample Program 5: Use of Math Functions

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

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

math.h is the filename containing the required function. Figure 2.7 illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20…….180 and prints out the results with headings.

```
/*———————— PROGRAM USING COSINE FUNCTION ———————— */
#include  <math.h>
#define    PI3.1416
#define    MAX180
main ( )
{
   int angle;
```

Contd.

```
                    float x,y;
                    angle = 0;
                    printf(" Angle     Cos(angle)\n\n");
                    while(angle <= MAX)
                    {
                          x = (PI/MAX)*angle;
                          y = cos(x);
                          printf("%15d %13.4f\n", angle, y);
                          angle = angle + 10;
               }
          }
Output
                       Angle              Cos(angle)
                         0                1.0000
                        10                0.9848
                        20                0.9397
                        30                0.8660
                        40                0.7660
                        50                0.6428
                        60                0.5000
                        70                0.3420
                        80                0.1736
                        90                -0.0000
                       100                -0.1737
                       110                -0.3420
                       120                -0.5000
                       130                -0.6428
                       140                -0.7660
                       150                -0.8660
                       160                -0.9397
                       170                -0.9848
                       180                -1.0000
```

Fig. 2.7 *Program using a math function*

Another **#include** instruction that is often required is

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

stdio.h refers to the *standard* I/O header file containing standard input and output functions

The #include Directive

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

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

#include<filename>

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

2.7 Basic Structure of C Programs

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more *statements* designed to perform a *specific task*. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 2.8.

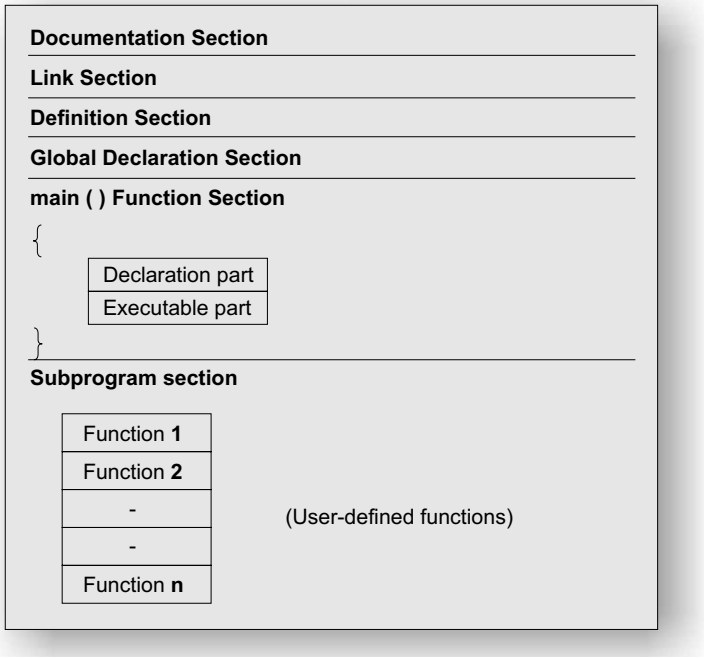


Fig. 2.8 *An overview of a C program*

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

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

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

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

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

2.8 Programming Style

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

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

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of Fig. 2.5.

Since C is a free-form language, we can group statements together on one line. The statements

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

can be written on one line as

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

The program

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

may be written in one line like

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

However, this style makes the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

2.9 Executing a 'C' Program

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

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

Figure 2.9 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the *operating system,* system commands for implementing the steps and conventions for naming *files* may differ on different systems.

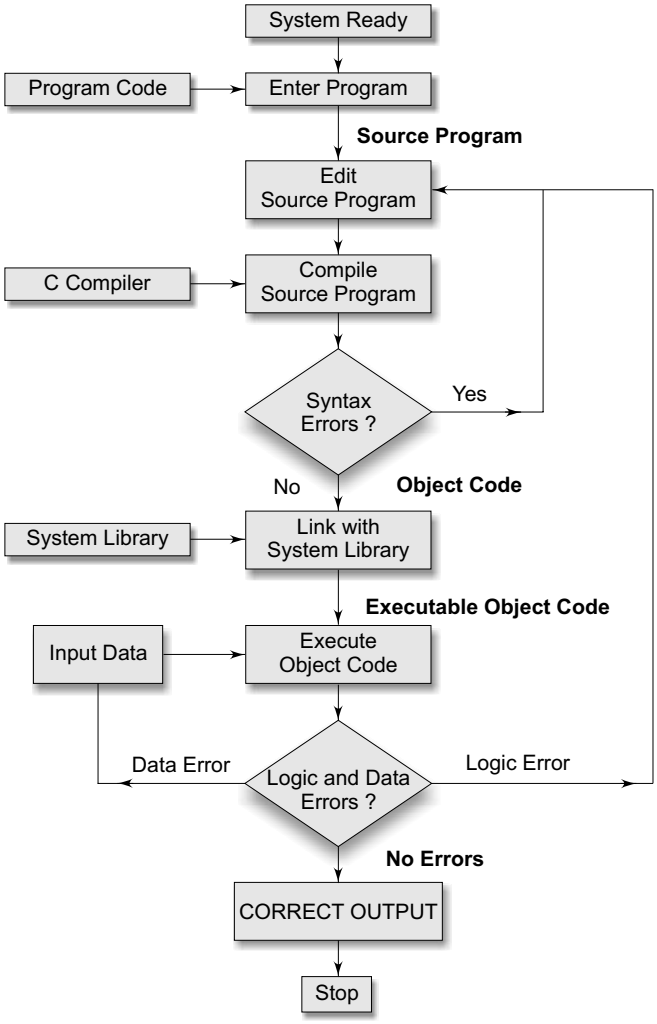


Fig. 2.9 *Process of compiling and running a C program*

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

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers).

Just Remember

- Every C program requires a **main()** function (Use of more than one **main()** is illegal). The place **main** is where the program execution begins.
- The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace.
- C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings.
- All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark.
- Every program statement in a C language must end with a semicolon.
- All variables must be declared for their types before they are used in the program.
- We must make sure to include header files using **#include** directive when the program refers to special names and functions that it does not define.
- Compiler directives such as **define** and **include** are special instructions to the compiler to help it compile a program. They do not end with a semicolon.
- The sign # of compiler directives must appear in the first column of the line.
- When braces are used to group statements, make sure that the opening brace has a corresponding closing brace.
- C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program.
- A comment can be inserted almost anywhere a space can appear. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and */ appropriately.

Review Questions

2.1 State whether the following statements are *true* or *false*.

(a) Every line in a C program should end with a semicolon.
(b) In C language lowercase letters are significant.
(c) Every C program ends with an END word.
(d) **main( )** is where the program begins its execution.
(e) A line in a program may have more than one statement.
(f) A **printf** statement can generate only one line of output.
(g) The closing brace of the **main( )** in a program is the logical end of the program.
(h) The purpose of the header file such as **stdio.h** is to store the source code of a program.

(i) Comments cause the computer to print the text enclosed between /* and */ when executed.
(j) Syntax errors will be detected by the compiler.

2.2 Which of the following statements are *true*?
(a) Every C program must have at least one user-defined function.
(b) Only one function may be named **main( )**.
(c) Declaration section contains instructions to the computer.

2.3 Which of the following statements about comments are *false*?
(a) Use of comments reduces the speed of execution of a program.
(b) Comments serve as internal documentation for programmers.
(c) A comment can be inserted in the middle of a statement.
(d) In C, we can have comments inside comments.

2.4 Fill in the blanks with appropriate words in each of the following statements.
(a) Every program statement in a C program must end with a _____
(b) The _____ Function is used to display the output on the screen.
(c) The _____ header file contains mathematical functions.
(d) The escape sequence character _____ causes the cursor to move to the next line on the screen.

2.5 Remove the semicolon at the end of the **printf** statement in the program of Fig. 1.2 and execute it. What is the output?

2.6 In the Sample Program 2, delete line-5 and execute the program. How helpful is the error message?

2.7 Modify the Sample Program 3 to display the following output:

Year	**Amount**
1	5500.00
2	6160.00
-	_____
-	_____
10	14197.11

2.8 Find errors, if any, in the following program:

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

2.9 Find errors, if any, in the following program:

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

2.10 Find errors, if any, in the following program:

```
Include <math.h>
main { }
(
```

```
        FLOAT X;
        X = 2.5;
        Y = exp(x);
        Print(x,y);
)
```

2.11 Why and when do we use the **#define** directive?
2.12 Why and when do we use the **#include** directive?
2.13 What does **void main(void)** mean?
2.14 Distinguish between the following pairs:
(a) main( ) and void main(void)
(b) int main( ) and void main( )
2.15 Why do we need to use comments in programs?
2.16 Why is the look of a program is important?
2.17 Where are blank spaces permitted in a C program?
2.18 Describe the structure of a C program.
2.19 Describe the process of creating and executing a C program under UNIX system.
2.20 How do we implement multiple source program files?

Programming Exercises

2.1 Write a program that will print your mailing address in the following form:
First line : Name
Second line : Door No, Street
Third line : City, Pin code
2.2 Modify the above program to provide border lines to the address.
2.3 Write a program using one print statement to print the pattern of asterisks as shown below:

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

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



2.5 Given the radius of a circle, write a program to compute and display its area. Use a symbolic constant to define the π value and assume a suitable value for radius.
2.6 Write a program to output the following multiplication table:

$5 \times 1 = 5$
$5 \times 2 = 10$
$5 \times 3 = 15$
. .
. .
$5 \times 10 = 50$

2.7 Given two integers 20 and 10, write a program that uses a function add( ) to add these two numbers and sub( ) to find the difference of these two numbers and then display the sum and difference in the following form:

$20 + 10 = 30$

$20 - 10 = 10$

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

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

Execute your program for the following values:

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

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

Comment on the output in each case.

2.9 Relationship between Celsius and Fahrenheit is governed by the formula

$$F = \frac{9C}{5} + 32$$

Write a program to convert the temperature

(a) from Celsius to Fahrenheit and

(b) from Fahrenheit to Celsius.

2.10 Area of a triangle is given by the formula

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

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

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

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

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

2.12 A point on the circumference of a circle whose center is (o, o) is (4,5). Write a program to compute perimeter and area of the circle. (Hint: use the formula given in the Ex. 2.11)

2.13 The line joining the points (2,2) and (5,6) which lie on the circumference of a circle is the diameter of the circle. Write a program to compute the area of the circle.

2.14 Write a program to display the equation of a line in the form

$$ax + by = c$$

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

2.15 Write a program to display the following simple arithmetic calculator

x =		y =	
sum =		Difference =	
Product =		Division =	

Debugging Exercises

2.1 Identify the error in the following program:

```
#include <stdio.h>
#include <math.h>
   void main()
   {
   long sum;
   int n,i;
   printf("Enter the value of n: ");
   scanf("%d",n);/*Reading value of n*/
   sum=1;
   for(i=1;i<=n;i++)
   {
    sum=sum+pow(2,i);/*Calculating the sum of the series*/
 }
 printf("Sum of series = %ld\n",sum);
}
```

2.2 Identify the error in the following program:

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

void main()
{
 int x,y;
 long z;
 printf("Enter the value of x: ");
 scanf("%d",&x);/*Reading the value of x*/
 printf("Enter the value of y: ");
 scanf("%d",&y);/*Reading the value of y*/
 z=power(x,y);/*Calling the built-in function*/
 printf("%d to the power of %d is equal to %ld", x, y, z);
}
```

2.3 Identify the error in the following program:

```
#include <stdio.h>

void main()
{
 int num;

 printf("\nEnter a number: ");
 scanf("%d",&num);/*Reading an Integer*/

 if(num%2=0)/*Using % operator to compute the remainder value*/
```

```
  printf("\n%d is an even number",num);
 else
  printf("\n%d is an odd number",num);
}
```

2.4 Identify the error in the following statements:

(a)

```
void x;
```

(b)

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

(c)

```
int i = 10;
char *ptr = &i;
```

(d)

```
char ch = 'j';
printf("\n\n%d",ch);
```

(e)

```
int i = 106;
printf("\n\n%c",i);
```

(f)

```
int int i;
```



3 Data Types in C

3.1 Introduction

A programming language is designed to help process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information.* The task of processing of data is accomplished by executing a sequence of precise instructions called a *program.* These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must confirm precisely to the syntax rules of the language.

Like any other language, C has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to C programming language.

3.2 Character Set

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

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

The entire character set is given in Table 3.1.

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

3.2.1 Trigraph Characters

Many non-English keyboards do not support all the characters mentioned in Table 3.1. ANSI C introduces the concept of "trigraph" sequences to provide a way to enter certain characters that are not available on

some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 3.2. For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraphs ??(and ??).

Table 3.1 ***C Character Set***

Letters			**Digits**
Uppercase A.....Z			All decimal digits 0 9
Lowercase a.....z			
		Special Characters	
	, comma		& ampersand
	. period		^ caret
	; semicolon		* asterisk
	: colon		– minus sign
	? question mark		+ plus sign
	‘ apostrophe		< opening angle bracket
	“ quotation mark		(or less than sign)
	! exclamation mark		> closing angle bracket
	| vertical bar		(or greater than sign)
	/ slash		(left parenthesis
	\ backslash		) right parenthesis
	~ tilde		[left bracket
	_ under score		] right bracket
	$ dollar sign		{ left brace
	% percent sign		} right brace
			# number sign
		White Spaces	
		Blank space	
		Horizontal tab	
		Carriage return	
		New line	
		Form feed	

Table 3.2 ***ANSI C Trigraph Sequences***

Trigraph sequence	**Translation**
??=	# number sign
??(	[left bracket

Contd.

Table 3.2 *(Contd)*

Trigraph sequence	Translation
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vertical bar
??/	\ back slash
??/	^ caret
??-	~ tilde

3.3 C Tokens

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 3.1. C programs are written using these tokens and the syntax of the language.

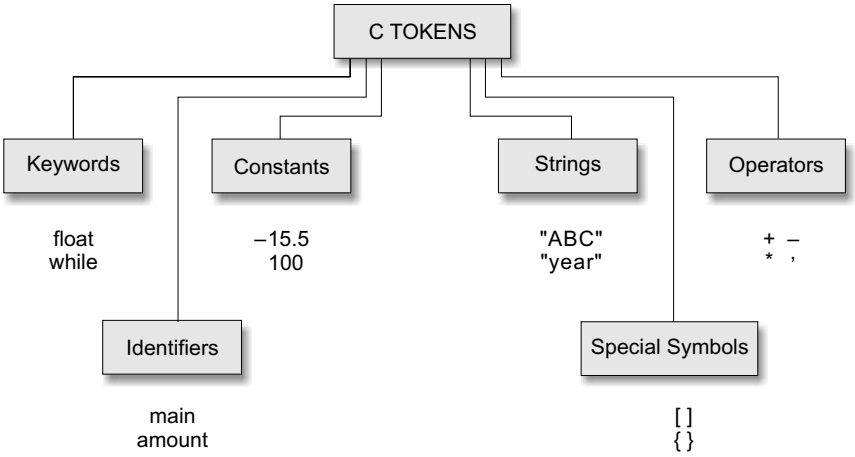


Fig. 3.1 *C tokens and examples*

3.4 Keywords and Identifiers

Every C word is classified as either a *keyword* or an *identifier*. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of ANSI C are listed in Table 3.3. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

Note *C99 adds some more keywords.*

Table 3.3 ***ANSI C Keywords***

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

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

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

3.5 Constants

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 3.2.

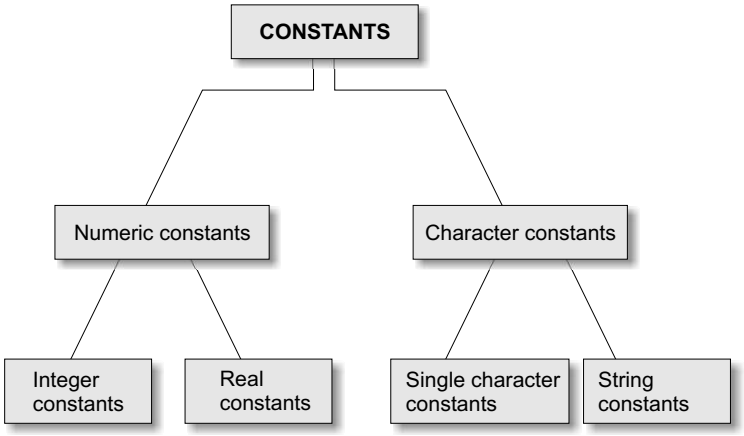


Fig. 3.2 *Basic types of C constants*

3.5.1 Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123 – 321 0 654321 +78

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750 20,000 $1000

are illegal numbers.

Note *ANSI C supports unary plus which was not defined earlier.*

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

037 0 0435 0551

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

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

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

Examples:

56789U	or 56789u	(unsigned integer)
987612347UL	or 98761234ul	(unsigned long integer)
9876543L	or 98765431	(long integer)

The concept of unsigned and long integers are discussed in detail in Section 3.7.

Example 3.1 Representation of integer constants on a 16-bit computer.

The program in Fig.3.3 illustrates the use of integer constants on a 16-bit machine. The output in Fig. 3.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

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

Contd.

```
Output
  Integer values
  32767 -32768 -32759
  Long integer values
  32767 32768 32777
```

Fig. 3.3 *Representation of integer constants on 16-bit machine*

3.5.2 Real Constants

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

0.0083 –0.75 435.36 +247.0

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

215. .95 –.71 +.5

are all valid real numbers.

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

mantissa e exponent

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

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

Embedded white space is not allowed.

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

Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and l or L to extend double precision further.

Some examples of valid and invalid numeric constants are given in Table 3.4.

Table 3.4 ***Examples of Numeric Constants***

Constant	Valid ?	Remarks
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	

Contd.

Table 3.4 ***(Contd)***

Constant	Valid ?	Remarks
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

3.5.3 Single Character Constants

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

'5' 'X' ';' ' '

Note that the character constant '5' is not the same as the *number 5*. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

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

would print the number 97, the ASCII value of the letter a. Similarly, the statement

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

would output the letter 'a'.

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

3.5.4 String Constants

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

"Hello!" "1987" "WELL DONE" "?...!" "5+3" "X"

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs.

3.5.5 Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 3.5. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as *escape sequences*.

Table 3.5 ***Backslash Character Constants***

Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space

Contd.

Table 3.5 ***(Contd)***

Constant	Meaning
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

3.6 Variables

A *variable* is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. In Chapter 2, we used several variables. For instance, we used the variable **amount** in Sample Program 3 to store the value of money at the end of each year (after adding the interest earned during that year).

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

Average
height
Total
Counter_1
class_strength

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

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers. (In C99, at least 63 characters are significant.)
3. Uppercase and lowercase are significant. That is, the variable **Total** is not the same as **total** or **TOTAL**.
4. It should not be a keyword.
5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

Further examples of variable names and their correctness are given in Table 3.6.

Table 3.6 ***Examples of Variable Names***

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

If only the first eight characters are recognized by a compiler, then the two names

average_height
average_weight

mean the same thing to the computer. Such names can be rewritten as

```
avg_height and avg_weight
```

or

```
ht_average and wt_average
```

without changing their meanings.

3.7 Data Types

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

ANSI C supports three classes of data types:

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

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

All C compilers support five fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), double-precision floating point (**double**) and **void**. Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given in Fig. 3.4. The range of the basic four types are given in Table 3.7. We discuss briefly each one of them in this section.

Note *C99 adds three more data types, namely* **_Bool**, **_Complex**, *and* **_Imaginary**.

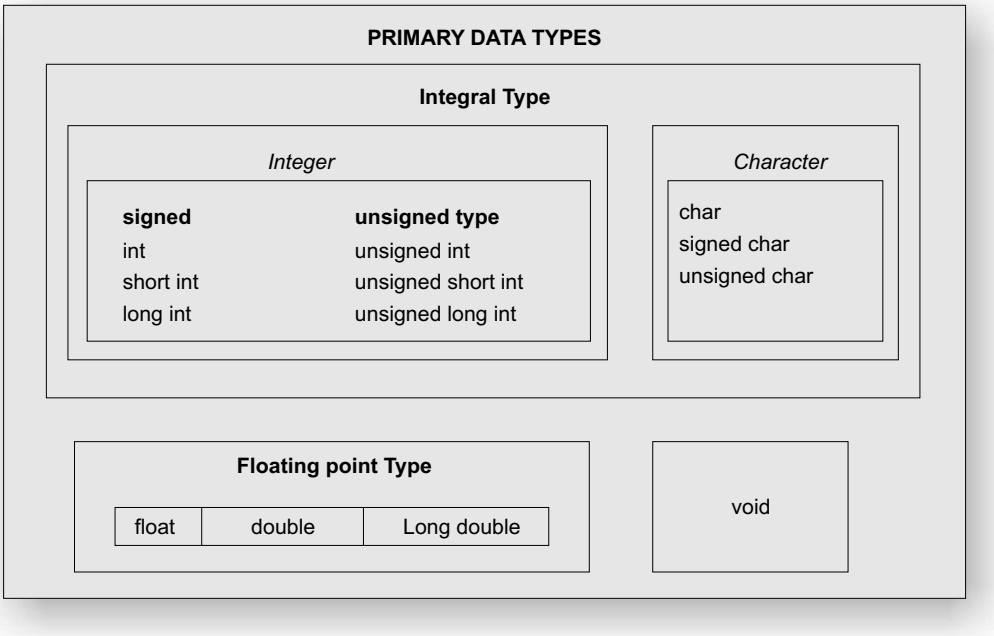


Fig. 3.4 *Primary data types in C*

Table 3.7 ***Size and Range of Basic Data Types on 16-bit Machines***

Data type	Range of values
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

3.7.1 Integer Types

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

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int,** and **long int,** in both **signed** and **unsigned** forms. ANSI C defines these types so that they can be organized from the smallest to the largest, as shown in Fig. 3.5. For example, **short int** represents fairly small integer values and requires half the amount of storage as

a regular **int** number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

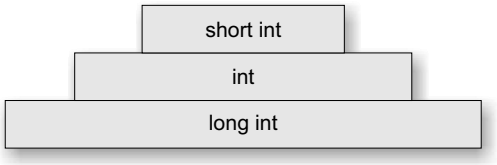


Fig. 3.5 *Integer types*

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. Table 3.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Note *C99 allows* **long long** *integer types.*

Table 3.8 ***Size and Range of Data Types on a 16-bit Machine***

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

3.7.2 Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float.** When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as *double precision* numbers. Remember that double type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, we may use **long double** which uses 80 bits. The relationship among floating types is illustrated in Fig. 3.6.

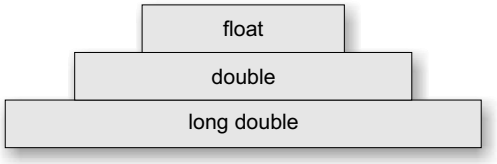


Fig. 3.6 *Floating-point types*

3.7.3 Void Types

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

3.7.4 Character Types

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

3.8 Declaration of Variables

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

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

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

3.8.1 Primary Type Declaration

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

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

v1, v2, ...vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example, valid declarations are:

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

int and **double** are the keywords to represent integer type and real type data values respectively. Table 3.9 shows various data types and their keyword equivalents.

Table 3.9 ***Data Types and Their Keywords***

Data type	Keyword equivalent
Character	char
Unsigned character	unsigned char

Contd.

Table 3.9 ***(Contd)***

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

The program segment given in Fig. 3.7 illustrates declaration of variables. **main**() is the beginning of the program. The opening brace { signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the **main** function. The importance of place of declaration will be dealt in detail later while discussing functions.

Note *C99 permits declaration of variables at any point within a function or block, prior to their use.*

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

Fig. 3.7 *Declaration of variables*

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

Default values of Constants

Integer constants, by default, represent **int** type data. We can override this default by specifying unsigned or long after the number (by appending U or L) as shown below:

Literal	Type	Value
+111	int	111
–222	int	–222
45678U	unsigned int	45,678
–56789L	long int	–56,789
987654UL	unsigned long int	9,87,654

Similarly, floating point constants, by default represent **double** type data. If we want the resulting data type to be **float** or **long double**, we must append the letter f or F to the number for **float** and letter l or L for **long double** as shown below:

Literal	Type	Value
0.	double	0.0
.0	double	0.0
12.0	double	12.0
1.234	double	1.234
–1.2f	float	–1.2
1.23456789L	long double	1.23456789

3.8.2 User-Defined Type Declaration

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

typedef *type* **identifier;**

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

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

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

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

batch1 and batch2 are inclared as **int** variable and name1[50] and name 2[50] are declared as 50 element floating point array variables. The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

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

enum identifier {*value1, value2, ... valuen*};

The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants*). After this definition, we can declare variables to be of this 'new' type as below:

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

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

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

An example:

```
enum day {Monday, Tuesday, ... Sunday};
enum day week_st, week_end;
week_st = Monday;
week_end = Friday;
if (week_st == Tuesday)
week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example:

```
enum day {Monday = 1, Tuesday, ... Sunday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. Example:

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

3.9 Assigning Values to Variables

Variables are created for use in program statements such as,

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

In the first statement, the numeric value stored in the variable **inrate** is multiplied by the value stored in **amount** and the product is added to **amount**. The result is stored in the variable value. This process is possible only if the variables **amount** and inrate have already been given values. The variable **value** is called the *target variable*. While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal (=) sign of a computational statement) *must* be assigned values before they are encountered in the program. Similarly, the variable **year** and the symbolic constant **PERIOD** in the **while** statement must be assigned values before this statement is encountered.

3.9.1 Assignment Statement

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

variable_name = constant;

We have already used such statements in Chapter 1. Further examples are:

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

C permits multiple assignments in one line. For example

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

are valid statements.

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The statement

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

means that the 'new value' of **year** is equal to the 'old value' of **year** plus 1.

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

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

data-type variable_name = constant;

Some examples are:

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

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

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

are valid. The first statement initializes the variables **p, q,** and **s** to zero while the second initializes **x, y,** and **z** with **MAX.** Note that **MAX** is a symbolic constant defined at the beginning.

Remember that external and static variables are initialized to zero by *default.* Automatic variables that are not initialized explicitly will contain garbage.

Example 3.2 Program in Fig. 3.8 shows typical declarations, assignments and values stored in various types of variables.

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

The variable **m** that has been declared as **int** is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an **int** variable can store is only 32767. However, the variable **k** (declared as **unsigned**) has stored the value 54321 correctly. Similarly, the **long int** variable **n** has stored the value 1234567890 correctly.

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

Program

```
main()
{
/*..........DECLARATIONS.............................*/
      float     x, p ;
      double    y, q ;
      unsigned  k ;
/*..........DECLARATIONS AND ASSIGNMENTS.............*/
      int       m = 54321 ;
      long int  n = 1234567890 ;
/*..........ASSIGNMENTS.............................*/
      x = 1.234567890000 ;
      y = 9.87654321 ;
      k = 54321 ;
      p = q = 1.0 ;
/*..........PRINTING.................................*/
      printf("m = %d\n", m) ;
      printf("n = %ld\n", n) ;
      printf("x = %.12lf\n", x) ;
      printf("x = %f\n", x) ;
      printf("y = %.12lf\n",y) ;
      printf("y = %lf\n", y) ;
      printf("k = %u p = %f q = %.12lf\n", k, p, q) ;
}
```

Output

```
      m = -11215
```

Contd.

```
n = 1234567890
x = 1.234567880630
x = 1.234568
y = 9.876543210000
y = 9.876543
k = 54321 p = 1.000000 q = 1.000000000000
```

Fig. 3.8 *Examples of assignments*

3.9.2 Reading Data from Keyboard

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

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

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

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

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

Example 3.3 The program in Fig. 3.9 illustrates the use of **scanf** function.

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

Enter an integer number

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

Your number is smaller than 100

is printed on the screen. Otherwise, the message

Your number contains more than two digits

is printed. Outputs of the program run for two different inputs are also shown in Fig. 3.9.

Program

```
main()
{
        int number;

        printf("Enter an integer number\n");
        scanf ("%d", &number);
```

Contd.

```
            if ( number < 100 )
               printf("Your number is smaller than 100\n\n");
            else
               printf("Your number contains more than two digits\n");
   }

Output
   Enter an integer number
   54
   Your number is smaller than 100
   Enter an integer number
   108
   Your number contains more than two digits
```

Fig. 3.9 *Use of **scanf** function for interactive computing*

Some compilers permit the use of the 'prompt message' as a part of the control string in **scanf**, like

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

We discuss more about **scanf** in Chapter 5.

In Fig. 3.9 we have used a decision statement **if...else** to decide whether the number is less than 100. Decision statements are discussed in depth in Chapter 6.

Example 3.4 Sample program 3 discussed in Chapter 2 can be converted into a more flexible interactive program using **scanf** as shown in Fig. 3.10.

In this case, computer requests the user to input the values of the amount to be invested, interest rate and period of investment by printing a prompt message

Input amount, interest rate, and period

and then waits for input values. As soon as we finish entering the three values corresponding to the three variables **amount, inrate,** and **period,** the computer begins to calculate the amount at the end of each year, up to 'period' and produces output as shown in Fig. 3.10.

```
Program
        main()
        {
              int year, period ;
              float amount, inrate, value ;
              printf("Input amount, interest rate, and period\n\n") ;
              scanf ("%f %f %d", &amount, &inrate, &period) ;
              printf("\n") ;
              year = 1 ;
              while( year <= period )
              {
                        value = amount + inrate * amount ;
                        printf("%2d Rs %8.2f\n", year, value) ;
```

Contd.

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

Fig. 3.10 *Interactive investment program*

Note that the **scanf** function contains three variables. In such cases, care should be exercised to see that the values entered match the *order* and *type* of the variables in the list. Any mismatch might lead to unexpected results. The compiler may not detect such errors.

3.10 Defining Symbolic Constants

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant **"pi".** Another example is the total number of students whose mark-sheets are analysed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. We face two problems in the subsequent use of such programs. These are

1. problem in modification of the program and
2. problem in understanding the program.

3.10.1 Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

3.10.2 Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

Assignment of such constants to a *symbolic name* frees us from these problems. For example, we may use the name **STRENGTH** to define the number of students and **PASS_MARK** to define the pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names **STRENGTH** and **PASS_MARK** in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is defined as follows:

#define symbolic-name value of constant

Valid examples of constant definitions are:

```
#define STRENGTH 100
#define PASS_MARK 50
#define MAX 200
#define PI 3.14159
```

Symbolic names are sometimes called *constant identifiers.* Since the symbolic names are constants (not variables), they do not appear in declarations. The following rules apply to a **#define** statement which define a symbolic constant:

1. Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letters. This is only a convention, not a rule.)
2. No blank space between the pound sign **'#'** and the word **define** is permitted.
3. **'#'** must be the first character in the line.
4. A blank space is required between **#define** and *symbolic name* and between the ***symbolic name*** and the *constant.*
5. **#define** statements must not end with a semicolon.
6. After definition, the *symbolic name* should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. **#define** statements may appear *anywhere* in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

#define statement is a *preprocessor* compiler directive and is much more powerful than what has been mentioned here. More advanced types of definitions will be discussed later. Table 3.10 illustrates some invalid statements of **#define.**

Table 3.10 ***Examples of Invalid #define Statements***

Statement	Validity	Remark
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No white space between # and define

Contd.

Table 3.10 *(Contd)*

Statement	Validity	Remark
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

3.11 Declaring a Variable as Constant

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier **const** at the time of initialization. Example:

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

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

3.12 Declaring a Variable as Volatile

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

```
volatile int date;
```

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

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

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

Note *C99 adds another qualifier called* ***restrict****.*

3.13 Overflow and Underflow of Data

Problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine. Since floating-point values are rounded off to the number of significant digits allowed (or specified), an overflow normally results in the largest possible real value, whereas an underflow results in zero.

Integers are always exact within the limits of the range of the integral data types used. However, an overflow which is a serious problem may occur if the data type does not match the value of the constant.

C does not provide any warning or indication of integer overflow. It simply gives incorrect results. (Overflow normally produces a negative number.) We should therefore exercise a greater care to define correct data types for handling the input/output values.

Just Remember

- Do not use the underscore as the first character of identifiers (or variable names) because many of the identifiers in the system library start with underscore.
- Use only 31 or less characters for identifiers. This helps ensure portability of programs.
- Do not use keywords or any system library names for identifiers.
- Use meaningful and intelligent variable names.
- Do not create variable names that differ only by one or two letters.
- Each variable used must be declared for its type at the beginning of the program or function.
- All variables must be initialized before they are used in the program.
- Integer constants, by default, assume **int** types. To make the numbers **long** or **unsigned**, we must append the letters L and U to them.
- Floating point constants default to **double**. To make them to denote **float** or **long double**, we must append the letters F or L to the numbers.
- Do not use lowercase l for long as it is usually confused with the number 1.
- Use single quote for character constants and double quotes for string constants.
- A character is stored as an integer. It is therefore possible to perform arithmetic operations on characters.
- Do not combine declarations with executable statements.
- A variable can be made constant either by using the preprocessor command **#define** at the beginning of the program or by declaring it with the qualifier **const** at the time of initialization.
- Do not use semicolon at the end of **#define** directive.
- The character # should be in the first column.
- Do not give any space between **#** and **define.**
- C does not provide any warning or indication of overflow. It simply gives incorrect results. Care should be exercised in defining correct data type.
- A variable defined before the main function is available to all the functions in the program.
- A variable defined inside a function is local to that function and not available to other functions.

Review Questions

3.1 State whether the following statements are *true* or *false*.
(a) Any valid printable ASCII character can be used in an identifier.
(b) All variables must be given a type when they are declared.
(c) Declarations can appear anywhere in a program.
(d) ANSI C treats the variables **name** and **Name** to be same.
(e) The underscore can be used anywhere in an identifier.
(f) The keyword **void** is a data type in C.

(g) Floating point constants, by default, denote **float** type values.
(h) Like variables, constants have a type.
(i) Character constants are coded using double quotes.
(j) Initialization is the process of assigning a value to a variable at the time of declaration.
(k) All **static** variables are automatically initialized to zero.
(l) The **scanf** function can be used to read only one value at a time.

3.2 Fill in the blanks with appropriate words.
(a) The keyword _____ can be used to create a data type identifier.
(b) _____ is the largest value that an unsigned short int type variable can store.
(c) A global variable is also known as _____ variable.
(d) A variable can be made constant by declaring it with the qualifier _____ at the time of initialization.

3.3 What are trigraph characters? How are they useful?

3.4 Describe the four basic data types. How could we extend the range of values they represent?

3.5 What is an unsigned integer constant? What is the significance of declaring a constant unsigned?

3.6 Describe the characteristics and purpose of escape sequence characters.

3.7 What is a variable and what is meant by the "value" of a variable?

3.8 How do variables and symbolic names differ?

3.9 State the differences between the declaration of a variable and the definition of a symbolic name.

3.10 What is initialization? Why is it important?

3.11 What are the qualifiers that an **int** can have at a time?

3.12 A programmer would like to use the word DPR to declare all the double-precision floating point values in his program. How could he achieve this?

3.13 What are enumeration variables? How are they declared? What is the advantage of using them in a program?

3.14 Describe the purpose of the qualifiers **const** and **volatile**.

3.15 When dealing with very small or very large numbers, what steps would you take to improve the accuracy of the calculations?

3.16 Which of the following are invalid constants and why?

0.0001	5 × 1.5	99999
+100	75.45 E-2	"15.75"
–45.6	–1.79 e + 4	0.00001234

3.17 Which of the following are invalid variable names and why?

Minimum	First.name	n1+n2	&name
doubles	3rd_row	n$	Row1
float	Sum Total	Row Total	Column-total

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

```
Int x;
float letter,DIGIT;
double = p,q
exponent alpha,beta;
```

```
m,n,z: INTEGER
short char c;
long int m; count;
long float temp;
```

3.19 What would be the value of x after execution of the following statements?

```
int x, y = 10;
char z = 'a';
x = y + z;
```

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

```
#define PI 3.14159
 main()
 {
       int R,C;          /* R-Radius of  circle
       float perimeter;  /* Circumference of circle */
       float area;       /* Area of circle */
       C = PI
       R = 5;
       Perimeter   = 2.0 * C *R;
       Area        = C*R*R;
       printf("%f", "%d",&perimeter,&area)
       }
```

Programming Exercises

3.1 Write a program to determine and print the sum of the following harmonic series for a given value of n:

$$1+ 1/2 +1/3 +....+ 1/n$$

The value of n should be given interactively through the terminal.

3.2 Write a program to read the price of an item in decimal form (like 15.95) and print the output in paise (like 1595 paise).

3.3 Write a program that prints the even numbers from 1 to 100.

3.4 Write a program that requests two float type numbers from the user and then divides the first number by the second and display the result along with the numbers.

3.5 The price of one kg of rice is Rs. 16.75 and one kg of sugar is Rs. 15. Write a program to get these values from the user and display the prices as follows:

*** LIST OF ITEMS ***

Item	Price
Rice	Rs 16.75
Sugar	Rs 15.00

3.6 Write program to count and print the number of negative and positive numbers in a given set of numbers. Test your program with a suitable set of numbers. Use **scanf** to read the numbers. Reading should be terminated when the value 0 is encountered.

3.7 Write a program to do the following:
(a) Declare x and y as integer variables and z as a short integer variable.
(b) Assign two 6 digit numbers to x and y
(c) Assign the sum of x and y to z
(d) Output the values of x, y and z
Comment on the output.

3.8 Write a program to read two floating point numbers using a **scanf** statement, assign their sum to an integer variable and then output the values of all the three variables.

3.9 Write a program to illustrate the use of **typedef** declaration in a program.

3.10 Write a program to illustrate the use of symbolic constants in a real-life application.

Debugging Exercises

3.1 Identify the error in the following program:

```
#include <stdio.h>
#DEFINE CP 50
void main( )
{
 int SP, profit;
 printf("Enter the SELLING PRICE ");
 scanf("%d", &SP);
 profit=SP-CP; /*Using Symbolic Constant CP in an expression*/
 printf("\nThe profit earned is %d", profit);
}
```

3.2 Identify the error in the following variable declaration statements:

```
(a)
      int dept-sales;
(b)
      char _flag01;
(c)
      int Int;
(d)
      int char;
(e)
char 27item;
(f)
char item27;
```

3.3 Identify the error in the following program:

```
#include <stdio.h>
void main()
{
 int a=b=c=3;
 int x;
 x=a-b/3+c*2-1;/*Expression evaluation*/
 printf("Value of x = %d",x);/*Printing the resultant value x*/
}
```

3.4 Identify the error in the following program:

```
#include <stdio.h>
void main()
{
 const int x=5;
 x=5;
 printf("Value of x = %d",x);
 getch();
}
```



4 Operators and their Hierarchy

4.1 Introduction

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

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

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

An expression is a sequence of operands and operators that reduces to a single value. For example,

10 + 15

is an expression whose value is 25. The value can be any type other than ***void***.

4.2 Arithmetic Operators

C provides all the basic arithmetic operators. They are listed in Table 4.1. The operators +, –, *, and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by –1. Therefore, a number preceded by a minus sign changes its sign.

Table 4.1 ***Arithmetic Operators***

Operator	Meaning
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

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

a – b a + b
a * b a / b
a % b –a * b

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

4.2.1 Integer Arithmetic

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

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

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

6/7 = 0 and –6/–7 = 0

but –6/7 may be zero or –1. (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

–14 % 3 = –2
–14 % –3 = –2
14 % –3 = 2

Example 4.1 The program in Fig. 4.1 shows the use of integer arithmetic to convert a given number of days into months and days.

Program

```
main ()
{
```

Contd

```
    int months, days ;
    printf("Enter days\n") ;
    scanf("%d", &days) ;
    months = days / 30 ;
    days = days % 30 ;
    printf("Months = %d Days = %d", months, days) ;
}
```

Output

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

Fig. 4.1 *Illustration of integer arithmetic*

The variables months and days are declared as integers. Therefore, the statement

```
months = days/30;
```

truncates the decimal part and assigns the integer part to months. Similarly, the statement

```
days = days%30;
```

assigns the remainder part of the division to days. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

4.2.2 Real Arithmetic

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

$$x = 6.0/7.0 = 0.857143$$
$$y = 1.0/3.0 = 0.333333$$
$$z = -2.0/3.0 = -0.666667$$

The operator % cannot be used with real operands.

4.2.3 Mixed-mode Arithmetic

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

$$15/10.0 = 1.5$$

whereas

$$15/10 = 1$$

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

4.3 Relational Operators

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

$$a < b \text{ or } 1 < 20$$

containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*. It is *one* if the specified relation is *true* and *zero* if the relation is *false*. For example

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

but

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

C supports six relational operators in all. These operators and their meanings are shown in Table 4.2.

Table 4.2 ***Relational Operators***

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

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

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

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

4.5 <= 10 TRUE

4.5 < –10 FALSE

–35 >= 0 FALSE

10 < 7+5 TRUE

a+b = c+d TRUE only if the sum of values of a and b is equal to the sum of values of c and d.

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

Relational expressions are used in *decision statements* such as **if** and **while** to decide the course of action of a running program. We have already used the **while** statement in Chapter 2. Decision statements are discussed in detail in Chapters 6 and 7.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

We can simplify an expression involving the **not** and the **less than** operators using the complements as shown below:

Actual one	Simplified one
!(x<y)	x >= y
!(x>y)	x <= y
!(x!=y)	x == y
!(x<=y)	x > y
!(x>=y)	x < y
!(x == y)	x != y

4.4 Logical Operators

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

&& meaning logical AND
|| meaning logical OR
! meaning logical NOT

The logical operators && and || are used when we want to test more than one condition and make decisions. An example is:

a > b && x == 10

An expression of this kind, which combines two or more relational expressions, is termed as a *logical expression* or a *compound relational expression*. Like the simple relational expressions, a logical expression also yields a value of *one* or *zero*, according to the truth table shown in Table 4.3. The logical expression given above is true only if **a** > **b** is *true* and **x** == 10 is true. If either (or both) of them are false, the expression is *false*.

Table 4.3 ***Truth Table***

op-1	op-2	Value of the expression	
		op-1 && op-2	op-1 || op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)
2. if (number < 0 || number > 100)

We shall see more of them when we discuss decision statements.

NOTE: Relative precedence of the relational and logical operators is as follows:

```
Highest !
        > >= < <=
        == !=
        &&
Lowest  ||
```

It is important to remember this when we use these operators in compound expressions.

4.5 Assignment Operators

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='. In addition, C has a set of *'shorthand'* assignment operators of the form

```
v op= exp;
```

where *v* is a variable, *exp* is an expression and *op* is a C binary arithmetic operator. The operator **op**= is known as the shorthand assignment operator.

The assignment statement

```
v op= exp;
```

is equivalent to

```
v = v op (exp);
```

with v evaluated only once. Consider an example

```
x += y+1;
```

This is same as the statement

```
x = x + (y+1);
```

The shorthand operator += means 'add y+1 to x' or 'increment x by y+1'. For y = 2, the above statement becomes

```
x += 3;
```

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 4.4.

Table 4.4 ***Shorthand Assignment Operators***

Statement with simple assignment operator	Statement with shorthand operator
a = a + 1	a += 1
a = a – 1	a –= 1
a = a * (n+1)	a *= n+1
a = a / (n+1)	a /= n+1
a = a % b	a %= b

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement like

```
value(5*j-2) = value(5*j-2) + delta;
```

With the help of the += operator, this can be written as follows:

```
value(5*j-2) += delta;
```

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

Example 4.2 Program of Fig. 4.2 prints a sequence of squares of numbers. Note the use of the shorthand operator *= .

The program attempts to print a sequence of squares of numbers starting from 2. The statement

```
a *= a;
```

which is identical to

```
a = a*a;
```

replaces the current value of a by its square. When the value of **a** becomes equal or greater than **N** (=100) the **while** is terminated. Note that the output contains only three values 2, 4 and 16.

```
Program
    #define  N  100
    #define  A  2
    main()
    {
            int a;
            a = A;
            while( a < N )
            {
                    printf("%d\n", a);
                    a *= a;
            }
    }
Output
    2
    4
    16
```

Fig. 4.2 *Use of shorthand operator* *=

4.6 Increment And Decrement Operators

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

```
++ and – –
```

The operator ++ adds 1 to the operand, while – – subtracts 1. Both are unary operators and takes the following form:

```
++m; or m++;
– –m; or m– –;
```

```
++m; is equivalent to m = m+1; (or m += 1;)
```

```
– –m; is equivalent to m = m–1; (or m –= 1;)
```

We use the increment and decrement statements in **for** and **while** loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
```

```
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
```

```
y = m++;
```

then, the value of y would be 5 and m would be 6. *A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.*

Similar is the case, when we use ++ (or – –) in subscripted variables. That is, the statement

```
a[i++] = 10;
```

is equivalent to

```
a[i] = 10;
```

```
i = i+1;
```

The increment and decrement operators can be used in complex statements. Example:

```
m = n++ –j+10;
```

Old value of n is used in evaluating the expression. n is incremented after the evaluation. Some compilers require a space on either side of n++ or ++n.

Rules for ++ and – – Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or – –) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++(or – –) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of ++ and – – operators are the same as those of unary + and unary –.

4.7 Conditional Operator

A ternary operator pair "? :" is available in C to construct conditional expressions of the form

```
exp1 ? exp2 : exp3
```

where *exp1*, *exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example, consider the following statements.

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the **if..else** statements as follows:

```
if (a > b)
    x = a;
else
  x = b;
```

4.8 Bitwise Operators

C has a distinction of supporting special operators known as *bitwise operators* for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to **float** or **double**. Table 4.5 lists the bitwise operators and their meanings.

Table 4.5 ***Bitwise Operators***

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

4.9 Special Operators

C supports some special operators of interest such as comma operator, **sizeof** operator, pointer operators (**&** and *****) and member selection operators (. and –>). The comma and **sizeof** operators are discussed in this section while the pointer operators are discussed in Chapter 11. Member selection operators which are used to select members of a structure are discussed in Chapters 12 and 11. ANSI committee has introduced two preprocessor operators (# and ##).

4.9.1 The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated *left to right* and the value of *right-most* expression is the value of the combined expression. For example, the statement

```
value = (x = 10, y = 5, x+y);
```

first assigns the value 10 to **x**, then assigns 5 to **y**, and finally assigns 15 (i.e. 10 + 5) to **value**. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

In **for** loops:

```
for ( n = 1, m = 10, n <=m; n++, m++)
```

In **while** loops:

```
while (c = getchar( ), c != '10')
```

Exchanging values:

```
t = x, x = y, y = t;
```

4.9.2 The sizeof Operator

The *sizeof* is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

m = **sizeof** (sum);
n = **sizeof** (long int);
k = **sizeof** (235L);

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Example 4.3 In Fig. 4.3, the program employs different kinds of operators. The results of their evaluation are also shown for comparison.

Notice the way the increment operator ++ works when used in an expression. In the statement

```
c = ++a - b;
```

new value of **a** (= 16) is used thus giving the value 6 to c. That is, a is incremented by 1 before it is used in the expression. However, in the statement

```
d = b++ + a;
```

the old value of **b** (=10) is used in the expression. Here, b is incremented by 1 after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement

```
printf("a%%b = %d\n", a%b);
```

The program also illustrates that the expression

```
c > d ? 1 : 0
```

assumes the value 0 when c is less than d and 1 when c is greater than d.

```
Program
  main()
  {
       int a, b, c, d;
       a = 15;
       b = 10;
       c = ++a - b;
       printf("a = %d b = %d c = %d\n",a, b, c);
       d = b++ +a;
       printf("a = %d b = %d d = %d\n",a, b, d);
       printf("a/b = %d\n", a/b);
       printf("a%%b = %d\n", a%b);
       printf("a *= b = %d\n", a*=b);
       printf("%d\n", (c>d) ? 1 : 0);
       printf("%d\n", (c<d) ? 1 : 0);
  }
Output
     a = 16b = 10  c = 6
     a = 16b = 11  d = 26
     a/b = 1
```

Contd

```
a%b = 5
a *= b = 176
0
1
```

Fig. 4.3 *Further illustration of arithmetic operators*

4.10 Arithmetic Expressions

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 4.6. Remember that C does not have an operator for exponentiation.

Table 4.6 ***Expressions***

Algebraic expression	C expression
a x b - c	a * b - c
(m+n) (x+y)	(m+n) * (x+y)
$\left(\frac{ab}{c}\right)$	a * b/c
$3x^2+2x+1$	3 * x * x + 2 * x + 1
$\left(\frac{x}{y}\right)+c$	x/y+c

4.11 Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form:

```
variable = expression;
```

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

```
x = a * b - c;

y = b / c * a;

z = a - b / c + d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c, and d must be defined before they are used in the expressions.

Example 4.4 The program in Fig. 4.4 illustrates the use of variables in expressions and their evaluation.

Output of the program also illustrates the effect of presence of parentheses in expressions. This is discussed in the next section.

```
Program
   main()
   {
            float a, b, c, x, y, z;
         a = 9;
         b = 12;
         c = 3;
         x = a - b / 3 + c * 2 - 1;
         y = a - b / (3 + c) * (2 - 1);
         z = a - (b / (3 + c) * 2) - 1;
         printf("x = %f\n", x);
         printf("y = %f\n", y);
         printf("z = %f\n", z);
   }
Output
         x = 10.000000
         y = 7.000000
         z = 4.000000
```

Fig. 4.4 *Illustrations of evaluation of expressions*

4.12 Precedence of Arithmetic Operators

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High priority * / %
Low priority + –

The basic evaluation procedure includes 'two' left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program of Fig. 4.4.

x = a–b/3 + c*2–1

When a = 9, b = 12, and c = 3, the statement becomes

x = 9–12/3 + 3*2–1

and is evaluated as follows

First pass

Step1: x = 9–4+3*2–1
Step2: x = 9–4+6–1

Second pass

Step3: x = 5+6–1
Step4: x = 11–1
Step5: x = 10

These steps are illustrated in Fig. 4.5. The numbers inside parentheses refer to step numbers.

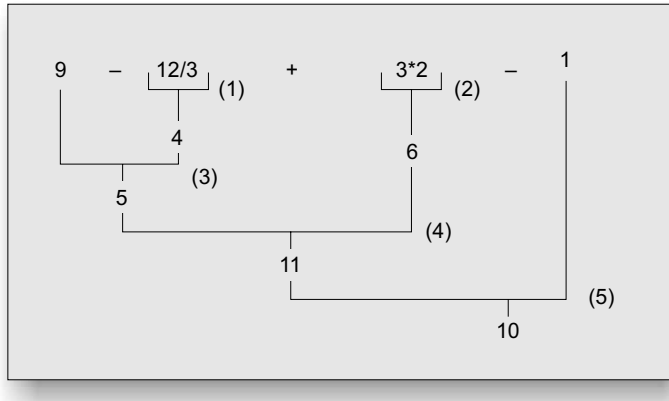


Fig. 4.5 *Illustration of hierarchy of operations*

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

9–12/(3+3)*(2–1)

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

Step 1: 9-12/6 * (2-1)
Step 2: 9-12/6 * 1

Second pass

Step 3: 9-2 * 1
Step 4: 9-2

Third pass

Step 5: 7

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e equal to the number of arithmetic operators).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis. For example

$$9 - (12/(3+3) * 2) - 1 = 4$$

whereas

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

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

Rules for Evaluation of Expression

- First, parenthesized sub-expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

4.13 Some Computational Problems

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:

```
a = 1.0/3.0;
b = a * 3.0;
```

We know that (1.0/3.0) 3.0 is equal to 1. But there is no guarantee that the value of **b** computed in a program will equal 1.

Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.

The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that operands are of the correct type and range, and the result may not produce any overflow or underflow.

Example 4.5 Output of the program in Fig. 4.6 shows round-off errors that can occur in computation of floating point numbers.

```
Program
  /*———————— Sum of n terms of 1/n ————————————*/
  main()
  {
      float sum, n, term ;
      int count = 1 ;
      sum = 0 ;
      printf("Enter value of n\n") ;
         scanf("%f", &n) ;
      term = 1.0/n ;
      while( count <= n )
      {
            sum = sum + term ;
            count++ ;
      }
      printf("Sum = %f\n", sum) ;
  }
Output
  Enter value of n
  99
  Sum = 1.000001
  Enter value of n
  143
  Sum = 0.999999
```

Fig. 4.6 *Round-off errors in floating point computations*

We know that the sum of n terms of 1/n is 1. However, due to errors in floating point representation, the result is not always 1.

4.14 Type Conversions In Expressions

4.14.1 Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as *implicit type conversion*.

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in Fig. 4.7.

Given below is the sequence of rules that are applied while evaluating expressions.

All **short** and **char** are automatically converted to **int;** then

1. if one of the operands is **long double**, the other will be converted to **long double** and the result will be **long double**;

2. else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double**;
3. else, if one of the operands is **float**, the other will be converted to **float** and the result will be **float;**
4. else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int;**
5. else, if one of the operands is long int and the other is **unsigned int**, then
   (a) if **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**;
   (b) else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int**;
6. else, if one of the operands is **long int**, the other will be converted to **long int** and the result will be **long int**;
7. else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and the result will be **unsigned int**.

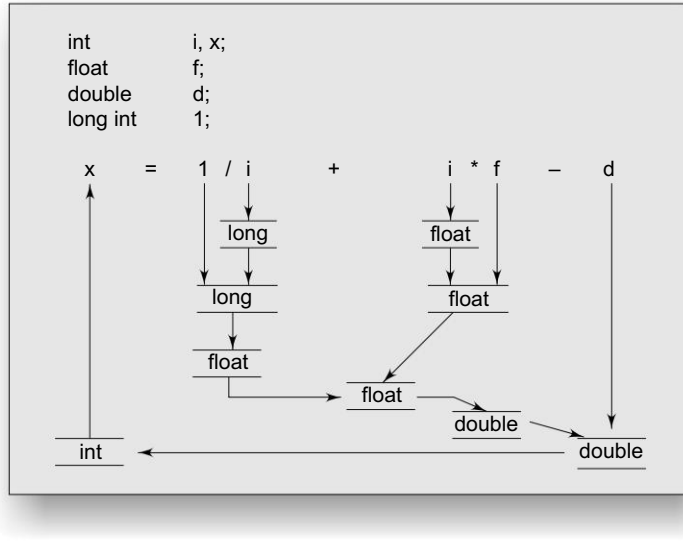
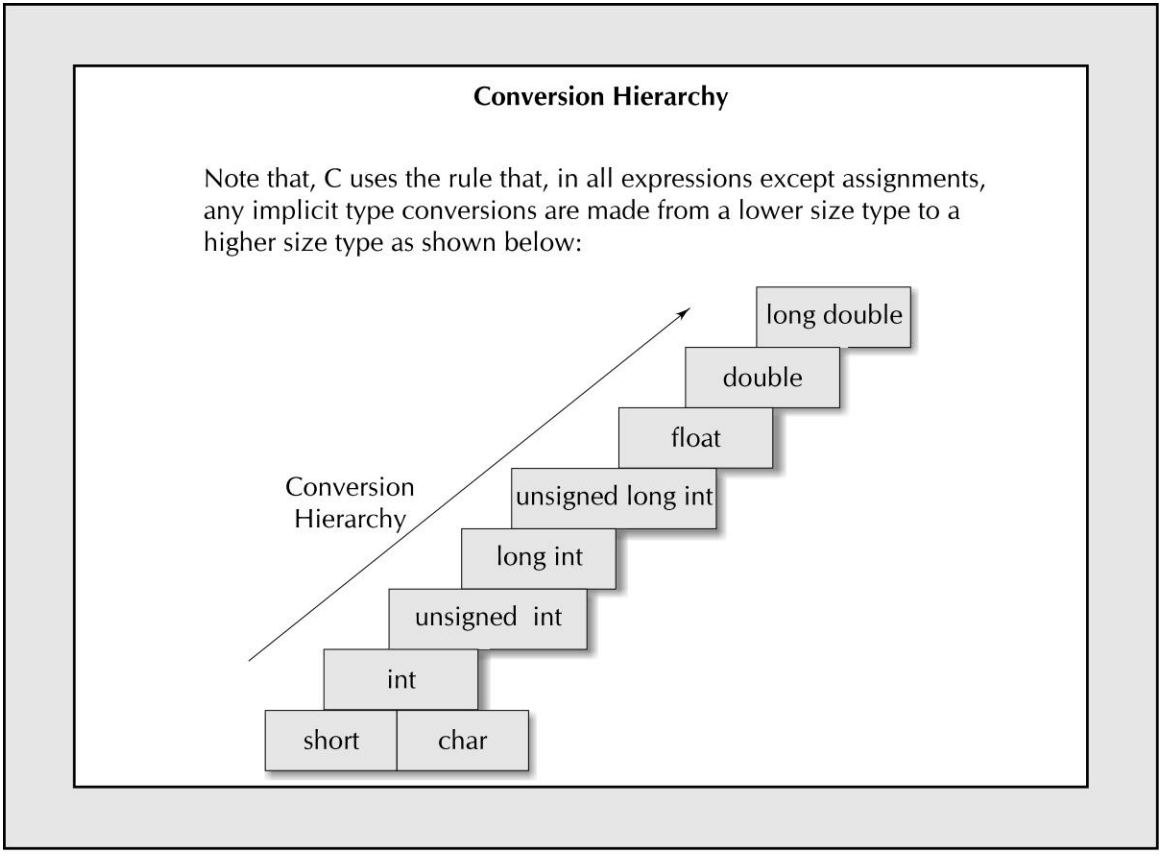


Fig. 4.7 *Process of implicit type conversion*

Note that some versions of C automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. **float** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.



4.14.2 Explicit Conversion

We have just discussed how C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

ratio = female_number/male_number

Since **female_number** and **male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

ratio = (**float**) female_number/male_number

The operator (**float**) converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (**float**) affect the value of the variable **female number**. And also, the type of **female number** remains as **int** in the other parts of the program.

The process of such a local conversion is known as *explicit conversion* or *casting a value*. The general form of a cast is:

```
(type-name)expression
```

where *type-name* is one of the standard C data types. The expression may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 4.7.

Table 4.7 ***Use of Casts***

Example	Action
x = (**int**) 7.5	7.5 is converted to integer by truncation.
a = (**int**) 21.3/(int)4.5	Evaluated as 21/4 and the result would be 5.
b = (**double**)sum/n	Division is done in floating point mode.
y = (**int**) (a+b)	The result of a+b is converted to integer.
z = (**int**)a+b	a is converted to integer and then added to b.
p = cos((**double**)x)	Converts x to double before using it.

Casting can be used to round-off a given value. Consider the following statement:

x = (**int**) (y+0.5);

If **y** is 27.6, **y**+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to **x**. Of course, the expression, being cast is not changed.

Example 4.6 Figure 4.8 shows a program using a cast to evaluate the equation sum = $\sum_{i=1}^{n}(1/i)$

```
Program
   main()
   {
         float   sum ;
         int     n ;
         sum = 0 ;
         for( n = 1 ; n <= 10 ; ++n )
         {
            sum = sum + 1/(float)n ;
            printf("%2d %6.4f\n", n, sum) ;
   }
}
Output
 1  1.0000
 2  1.5000
 3  1.8333
 4  2.0833
 5  2.2833
 6  2.4500
 7  2.5929
```

Contd

```
8   2.7179
9   2.8290
10  2.9290
```

Fig. 4.8 *Use of a cast*

4.15 Operator Precedence And Associativity

As mentioned earlier each operator, in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct *levels of precedence* and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from 'left to right' or from 'right to left', depending on the level. This is known as the *associativity* property of an operator. Table 4.8 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and 15 the lowest. The list also includes those operators, which we have not yet discussed.

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

if (x == 10 + 15 && y < 10)

The precedence rules say that the ***addition*** operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

if (x == 25 && y < 10)

The next step is to determine whether **x** is equal to 25 and **y** is less than 10. If we assume a value of 20 for x and 5 for y, then

x == 25 is FALSE (0)
y < 10 is TRUE (1)

Note that since the operator < enjoys a higher priority compared to ==, y < 10 is tested first and then x == 25 is tested.

Finally we get:

if (FALSE && TRUE)

Because one of the conditions is FALSE, the complex condition is FALSE.

In the case of **&&**, it is guaranteed that the second operand will not be evaluated if the first is zero and in the case of ||, the second operand will not be evaluated if the first is non-zero.

Table 4.8 ***Summary of C Operators***

Operator	Description	Associativity	Rank
( )	Function call	Left to right	1
[ ]	Array element reference		
+	Unary plus		

Contd.

Table 4.8 *(Contd)*

Operator	Description	Associativity	Rank
–	Unary minus	Right to left	2
++	Increment		
– –	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference (indirection)		
&	Address		
sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
–	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
|=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
=	Assignment operators	Right to left	14
* = /= %=			
+= –= &=			
^= |=			
<<= >>=			
,	Comma operator	Left to right	15

Rules of Precedence and Associativity

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied

4.16 Mathematical Functions

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Most of the C compilers support these basic math functions. However, there are systems that have a more comprehensive math library and one should consult the reference manual to find out which functions are available. Table 4.9 lists some standard math functions.

Table 4.9 ***Math functions***

Function	Meaning
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power (e^x)
fabs(x)	Absolute value of x.
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, x > 0
log10(x)	Base 10 log of x, x > 0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x > = 0

Note

1. **x** and **y** should be declared as **double**.
2. In trigonometric and hyperbolic functions, **x** and **y** are in radians.
3. All the functions return a **double**.
4. C99 has added **float** and **long double** versions of these fuctions.
5. C99 has added many more mathematical functions.
6. See the Appendix "C99 Features" for details.

As pointed out earlier in Chapter 2, to use any of these functions in a program, we should include the line:

 # include <math.h>

in the beginning of the program.

Just Remember

- Use *decrement* and *increment* operators carefully. Understand the difference between **postfix** and **prefix** operations before using them.
- Add parentheses wherever you feel they would help to make the evaluation order clear.
- Be aware of side effects produced by some expressions.
- Avoid any attempt to divide by zero. It is normally undefined. It will either result in a fatal error or in incorrect results.
- Do not forget a semicolon at the end of an expression.
- Understand clearly the precedence of operators in an expression. Use parentheses, if necessary.
- Associativity is applied when more than one operator of the same precedence are used in an expression. Understand which operators associate from right to left and which associate from left to right.
- Do not use *increment* or *decrement* operators with any expression other than a *variable identifier*.
- It is illegal to apply modules operator % with anything other than integers.
- Do not use a variable in an expression before it has been assigned a value.
- Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary.
- The result of an expression is converted to the type of the variable on the left of the assignment before assigning the value to it. Be careful about the loss of information during the conversion.
- All mathematical functions implement *double* type parameters and *return* double type values.
- It is an error if any space appears between the two symbols of the operators ==, !=, <= and >=.
- It is an error if the two symbols of the operators !=, <= and >= are reversed.
- Use spaces on either side of binary operator to improve the readability of the code.
- Do not use increment and decrement operators to floating point variables.
- Do not confuse the equality operator == with the assignment operator =.

Review Questions

4.1 State whether the following statements are *true* or *false.*

(a) All arithmetic operators have the same level of precedence.

(b) The modulus operator % can be used only with integers.

(c) The operators <=, >= and != all enjoy the same level of priority.

(d) During modulo division, the sign of the result is positive, if both the operands are of the same sign.

(e) In C, if a data item is zero, it is considered false.

(f) The expression !(x<=y) is same as the expression x>y.

(g) A unary expression consists of only one operand with no operators.
(h) Associativity is used to decide which of several different expressions is evaluated first.
(i) An expression statement is terminated with a period.
(j) During the evaluation of mixed expressions, an implicit cast is generated automatically.
(k) An explicit cast can be used to change the expression.
(l) Parentheses can be used to change the order of evaluation expressions.

4.2 Fill in the blanks with appropriate words.
(a) The expression containing all the integer operands is called _____ expression.
(b) The operator _____ cannot be used with real operands.
(c) C supports as many as _____ relational operators.
(d) An expression that combines two or more relational expressions is termed as _____ expression.
(e) The _____ operator returns the number of bytes the operand occupies.
(f) The order of evaluation can be changed by using _____ in an expression.
(g) The use _____ of on a variable can change its type in the memory.
(h) _____ is used to determine the order in which different operators in an expression are evaluated.

4.3 Given the statement
int a = 10, b = 20, c;
determine whether each of the following statements are true or false.
(a) The statement a = + 10, is valid.
(b) The expression a + 4/6 * 6/2 evaluates to 11.
(c) The expression b + 3/2 * 2/3 evaluates to 20.
(d) The statement a + = b; gives the values 30 to a and 20 to b.
(e) The statement ++a++; gives the value 12 to a
(f) The statement a = 1/b; assigns the value 0.5 to a

4.4 Declared **a** as *int* and **b** as *float*, state whether the following statements are true or false.
(a) The statement a = 1/3 + 1/3 + 1/3; assigns the value 1 to a.
(b) The statement b = 1.0/3.0 + 1.0/3.0 + 1.0/3.0; assigns a value 1.0 to b.
(c) The statement b = 1.0/3.0 * 3.0 gives a value 1.0 to b.
(d) The statement b = 1.0/3.0 + 2.0/3.0 assigns a value 1.0 to b.
(e) The statement a = 15/10.0 + 3/2; assigns a value 3 to a.

4.5 Which of the following expressions are true?
(a) !(5 + 5 >=10)
(b) 5 + 5 = = 10 || 1 + 3 = = 5
(c) 5 > 10 || 10 < 20 && 3 < 5
(d) 10 ! = 15 && !(10<20) || 15 > 30

4.6 Which of the following arithmetic expressions are valid ? If valid, give the value of the expression; otherwise give reason.
(a) 25/3 % 2
(b) +9/4 + 5
(c) 7.5 % 3
(d) 14 % 3 + 7 % 2
(e) –14 % 3
(f) 15.25 + – 5.0
(g) (5/3) * 3 + 5 % 3
(h) 21 % (int)4.5

4.7 Write C assignment statements to evaluate the following equations:

(a) Area = $\pi r^2 + 2 \pi rh$

(b) Torque = $\frac{2m_1m_2}{m_1+m_2}.g$

(c) Side = $\sqrt{a^2+b^2 - 2ab\cos(x)}$

(d) Energy = mass $\left[\text{acceleration} \times \text{height} + \frac{(\text{velocity})^2}{2}\right]$

4.8 Identify unnecessary parentheses in the following arithmetic expressions.

(a) ((x–(y/5)+z)%8) + 25
(b) ((x–y) * p)+q
(c) (m*n) + (–x/y)
(d) x/(3*y)

4.9 Find errors, if any, in the following assignment statements and rectify them.

(a) `x = y = z = 0.5, 2.0. -5.75;`
(b) `m = ++a * 5;`
(c) `y = sqrt(100);`
(d) `p * = x/y;`
(e) `s = /5;`
(f) `a = b++ -c*2`

4.10 Determine the value of each of the following logical expressions if a = 5, b = 10 and c = –6

(a) a > b && a < c
(b) a < b && a > c
(c) a == c || b > a
(d) b > 15 && c < 0 || a > 0
(e) (a/2.0 == 0.0 && b/2.0 != 0.0) || c < 0.0

4.11 What is the output of the following program?

```
main ( )
{
        char x;
        int y;
        x = 100;
        y = 125;
        printf ("%c\n", x) ;
        printf ("%c\n", y) ;
        printf ("%d\n", x) ;
}
```

4.12 Find the output of the following program.

```
main ( )
{
```

```
        int x = 100;
        printf("%d/n", 10 + x++);
        printf("%d/n", 10 + ++x);
}
```

4.13 What is printed by the following program?

```
main
{
    int x = 5, y = 10, z = 10 ;
    x = y == z;
    printf("%d",x ) ;
}
```

4.14 What is the output of the following program?

```
main ( )
{
    int x = 100, y = 200;
    printf ("%d", (x > y)? x : y);
}
```

4.15 What is the output of the following program?

```
main ( )
{
    unsigned x = 1 ;
    signed char y = -1 ;
    if(x > y)
        printf(" x > y");
    else
        printf("x<= y") ;
}
Did you expect this output? Explain.
```

4.16 What is the output of the following program? Explain the output.

```
main ( )
{
    int x = 10 ;
    if(x = 20) printf("TRUE") ;
    else printf("FALSE") ;
}
```

4.17 What is the error in each of the following statements?

(a) if (m == 1 & n ! = 0)
    printf("OK");

(b) if (x = < 5)
    printf ("Jump");

4.18 What is the error, if any, in the following segment?

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

4.19 What is printed when the following is executed?

```
for (m = 0; m <3; ++m)
printf("%d/n", (m%2) ? m: m+2);
```

4.20 What is the output of the following segment when executed?

```
int m = - 14, n = 3;
printf("%d\n", m/n * 10) ;
n = -n;
printf("%d\n", m/n * 10);
```

Programming Exercises

4.1 Given the values of the variables x, y and z, write a program to rotate their values such that x has the value of y, y has the value of z, and z has the value of x.

4.2 Write a program that reads a floating-point number and then displays the right-most digit of the integral part of the number.

4.3 Modify the above program to display the two right-most digits of the integral part of the number.

4.4 Write a program that will obtain the length and width of a rectangle from the user and compute its area and perimeter.

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

First line : all digits
Second line : all except first digit
Third line : all except first two digits
…….
Last line : The last digit

For example, the number 5678 will be displayed as:

5 6 7 8
6 7 8
7 8
8

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

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

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

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

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

4.8 The total distance travelled by a vehicle in t seconds is given by

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

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

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

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

and the optimal Time Between Orders

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

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

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

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

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

4.11 Write a program to read a four digit integer and print the sum of its digits.
Hint: Use / and % operators.

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

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

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

4.15 Write a program to read three values using **scanf** statement and print the following results:
(a) Sum of the values
(b) Average of the three values
(c) Largest of the three
(d) Smallest of the three

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

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

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

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

Number	Square-root	Square
0	0	0
100	10	10000

4.19 Write a program that determines whether a given integer is odd or even and displays the number and description on the same line.

4.20 Write a program to illustrate the use of cast operator in a real life situation.

Debugging Exercises

4.1 Identify the error in the following program; or if there is no error, give its output:

```
(#include <stdio.h>
void main()
{
 if((10/2-4)&&(7%3)||(0/10))
  printf("True");
 else
  printf("False");
}
```

4.2 Identify the error in the following program; or if there is no error, give its output:

```
#include <stdio.h>
void main()
{
 long num, temp;
 int sum=0;
 printf("\nEnter an integer value: ");
 scanf("%ld",&num);
```

```
 temp=num;
 while(temp!==0)
 {
 sum = sum+temp%10;
 temp=temp/10;
 }

 printf("\n\nThe sum of digits of %ld is %d",num,sum);/*Displaying
result*/
}
```

4.3 Identify the error in the following program that prints the sum of odd numbers between 1 and 50:

```
/*Program to print sum of odd numbers between 1 and 50*/
#include<stdio.h>
void main()
{
   int i,sum=0;
   for (i=1;i<50;i++)
   {
   if(i%2==0)
      break;
      sum=sum+i;
   }
   printf("\nSum of odd numbers between 1 to 50 is = %d", sum);
}
```

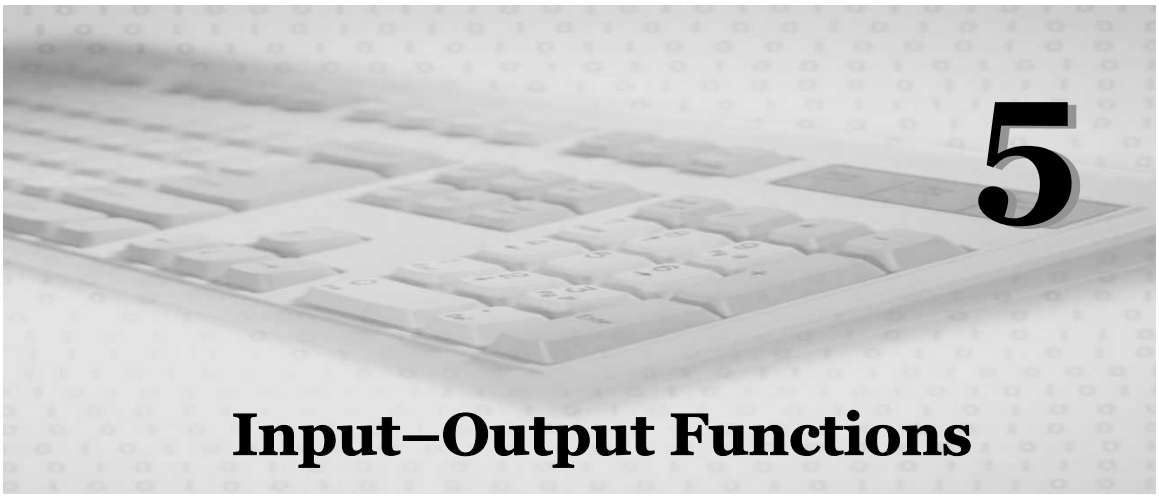
4.4 Identify the error in the following code; or if there is no error, give its output:

```
void main()
{
 const int a=5;
 int b=4;
 int c;
 c=a==b;
 printf("c=%d",c);
}
```

4.5 Identify the error in the following program that demonstrates the use of bitwise operations AND and OR:

```
#include <stdio.h>
void main()
{
```

```
    char a='A',b='B';
    printf("a AND b = %c",a&b);
    printf("\na OR b = %c",a^b);
}
```



5 Input–Output Functions

5.1 Introduction

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

Unlike other high-level languages, C does not have any built-in input/output statements as part of its syntax. All input/output operations are carried out through function calls such as **printf** and **scanf**. There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the standard I/O library. In this chapter we shall discuss some common I/O functions that can be used on many machines without any change. However, one should consult the system reference manual for exact details of these functions and also to see what other functions are available.

To use a particular function from the I/O library we must include its corresponding header file at the beginning of the program; otherwise the compiler won't recognise the functions and result in errors. A header file contains the source code of the built-in library functions. For instance, the stdio.h header file contains the source code of standard input/output functions while math.h contains the source code of built-in mathematical functions. A header file is included at the beginning of the program with the help of a pre-processor directive, as shown below:

#include <stdio.h>

It may be recalled that we have included a statement

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

in the Sample Program 5 in Chapter 2. where a math library function cos (x) has been used.

This is to instruct the compiler to fetch the function cos(x) from the math library, and that it is not a part of C language. Similarly, each program that uses a standard input/output function must contain the statement

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

at the beginning. However, there might be exceptions. For example, this is not necessary for the functions **printf** and **scanf** which have been defined as a part of the C language.

The file name **stdio.h** is an abbreviation for *standard input-output header* file. The instruction **#include <*stdio.h*>** tells the compiler 'to search for a file named **stdio.h** and place its contents at this point in the program'. The contents of the header file become part of the source code when it is compiled.

5.2 Reading a Character

The simplest of all input/output operations is reading a character from the 'standard input' unit (usually the keyboard) and writing it to the 'standard output' unit (usually the screen). Reading a single character can be done by using the function **getchar.** (This can also be done with the help of the **scanf** function which is discussed in Section 5.4.) The **getchar** takes the following form:

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

variable_name is a valid C name that has been declared as **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function. Since **getchar** is used on the right-hand side of an assignment statement, the character value of **getchar** is in turn assigned to the variable name on the left. For example

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

Will assign the character 'H' to the variable **name** when we press the key H on the keyboard. Since **getchar** is a function, it requires a set of parentheses as shown.

Example 5.1 The program in Fig. 5.1 shows the use of **getchar** function in an interactive environment.

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

My name is BUSY BEE

otherwise, outputs

You are good for nothing

Note *There is one line space between the input text and output message.*

Program

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

Contd.

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

Fig. 5.1 *Use of* ***getchar*** *function to read a* ***character*** *from keyboard*

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

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

Warning

The **getchar()** function accepts any character keyed in. This includes RETURN and TAB. This means when we enter single character input, the newline character is waiting in the input queue after **getchar()** returns. This could create problems when we use **getchar()** in a loop interactively. A dummy **getchar()** may be used to 'eat' the unwanted newline character. We can also use the **fflush** function to flush out the unwanted characters.

Note *We shall be using decision statements like* **if, if…else** *and* **while** *extensively in this chapter. They are discussed in detail in Chapters 6 and 7.*

Example 5.2 The program of Fig. 5.2 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

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

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

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

Program

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

Output

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

Fig. 5.2 *Program to test the* ***character*** *type*

C supports many other similar functions, which are given in Table 5.1. These character functions are contained in the file **ctype.h** and therefore the statement

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

must be included in the program.

Table 5.1 ***Character Test Functions***

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

5.3 Writing a Character

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

```
putchar (variable_name);
```

where ***variable_name*** is a type **char** variable containing a character. This statement displays the character contained in the ***variable_name*** at the terminal. For example, the statements

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

will display the character Y on the screen. The statement

```
putchar ('\n');
```

would cause the cursor on the screen to move to the beginning of the next line.

Example 5.3 A program that reads a character from keyboard and then prints it in reverse case is given in Fig. 5.3. That is, if the input is upper case, the output will be lower case and vice versa.

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

Program

```
#include <stdio.h>
#include <ctype.h>
main()
{
   char alphabet;
   printf("Enter an alphabet");
```

Contd.

```
            putchar('\n'); /* move to next line */
            alphabet = getchar();
            if (islower(alphabet))
            putchar(toupper(alphabet));/* Reverse and display */
         else
            putchar(tolower(alphabet)); /* Reverse and display */
      }
Output
            Enter an alphabet
            a
            A
            Enter an alphabet
            Q
            q
            Enter an alphabet
            z
            Z
```

Fig. 5.3 *Reading and writing of alphabets in reverse case*

5.4 Formatted Input

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

```
15.75 123 John
```

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

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

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

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

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

- Field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an *optional* number, specifying the field width.
- Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional. The discussions that follow will clarify these concepts.

5.4.1 Inputting Integer Numbers

The field specification for reading an integer number is:

```
% w sd
```

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

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

Data line:

```
50  31426
```

The value 50 is assigned to **num1** and 31426 to **num2**. Suppose the input data is as follows:

```
31426 50
```

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

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

will read the data

```
31426 50
```

correctly and assign 31426 to **num1** and 50 to **num2.**

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

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

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

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

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

will assign the data

```
123 456 789
```

as follows:

```
123 to a
456 skipped (because of *)
789 to b
```

The data type character **d** may be preceded by '**l**' (letter ell) to read long integers and **h** to read short integers.

Note *We have provided white space between the field specifications. These spaces are not necessary with the numeric input, but it is a good practice to include them.*

Example 5.4 Various input formatting options for reading integers are experimented in the program shown in Fig. 5.4.

```
Program
   main()
   {
      int a,b,c,x,y,z;
      int p,q,r;
      printf("Enter three integer numbers\n");
      scanf("%d %*d %d",&a,&b,&c);
      printf("%d %d %d \n\n",a,b,c);
      printf("Enter two 4-digit numbers\n");
      scanf("%2d %4d",&x,&y);
      printf("%d %d\n\n", x,y);
      printf("Enter two integers\n");
      scanf("%d %d", &a,&x);
      printf("%d %d \n\n",a,x);
      printf("Enter a nine digit number\n");
      scanf("%3d %4d %3d",&p,&q,&r);
      printf("%d %d %d \n\n",p,q,r);
      printf("Enter two three digit numbers\n");
      scanf("%d %d",&x,&y);
      printf("%d %d",x,y);
   }
Output
      Enter three integer numbers
      1 2 3
      1 3 -3577
      Enter two 4-digit numbers
      6789 4321
      67 89
      Enter two integers
      44 66
      4321 44
```

Contd.

```
Enter a nine-digit number
123456789
66 1234 567
Enter two three-digit numbers
123 456
89 123
```

Fig. 5.4 *Reading integers using* ***scanf***

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

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

Note *It is legal to use a non-whitespace character between field specifications. However, the* **scanf** *expects a matching character in the given location. For example,*

```
scanf("%d-%d", &a, &b);
```

accepts input like

```
123-456
```

to assign 123 to a and 456 to b.

5.4.2 Inputting Real Numbers

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

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

with the input data

```
475.89 43.21E-1 678
```

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

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

Example 5.5 Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 5.5.

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

Fig. 5.5 *Reading of* ***real*** *numbers*

5.4.3 Inputting Character Strings

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

```
%ws  or  %wc
```

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

Example 5.6 Reading of strings using **%wc** and **%ws** is illustrated in Fig. 5.6.

The program in Fig. 5.6 illustrates the use of various field specifications for reading strings. When we use **%wc** for reading a string, the system will wait until the w^{th} character is keyed in. **Note** that the specification **%s** terminates reading at the encounter of a blank space. Therefore, **name2** has read only the first part of "New York" and the second part is automatically assigned to **name3**. However, during the second run, the string "New-York" is correctly assigned to **name2**.

```
Program
      main()
      {
      int no;
```

Contd.

```
        char name1[15], name2[15], name3[15];
        printf("Enter serial number and name one\n");
        scanf("%d %15c", &no, name1);
        printf("%d %15s\n\n", no, name1);
        printf("Enter serial number and name two\n");
        scanf("%d %s", &no, name2);
        printf("%d %15s\n\n", no, name2);
        printf("Enter serial number and name three\n");
        scanf("%d %15s", &no, name3);
        printf("%d %15s\n\n", no, name3);
    }
Output
        Enter serial number and name one
        1 123456789012345
        1 123456789012345r
        Enter serial number and name two
        2 New York
        2               New
        Enter serial number and name three
        2               York
        Enter serial number and name one
        1 123456789012
        1 123456789012r
        Enter serial number and name two
        2 New-York
        2               New-York
        Enter serial number and name three
        3 London
        3               London
```

Fig. 5.6 *Reading of strings*

Some versions of **scanf** support the following conversion specifications for strings:

```
%[characters]
%[^characters]
```

The specification **%[characters]** means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification **%[^characters]** does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

Example 5.7 The program in Fig. 5.7 illustrates the function of %[ ] specification.

```
Program-A
      main()
      {
         char address[80];
         printf("Enter address\n");
         scanf("%[a-z]", address);
         printf("%-80s\n\n", address);
         }
Output
         Enter address
         new delhi 110002
         new delhi
Program-B
      main()
      {
         char address[80];
         printf("Enter address\n");
         scanf("%[^\n]", address);
         printf("%-80s", address);
      }
Output
         Enter address
         New Delhi 110 002
         New Delhi 110 002
```

Fig. 5.7 *Illustration of conversion specification%* [ ] *for strings*

Reading Blank Spaces

We have earlier seen that %s specifier cannot be used to read strings with blank spaces. But, this can be done with the help of %[ ] specification. Blank spaces may be included within the brackets, thus enabling the ***s*canf** to read strings with spaces. Remember that the lowercase and uppercase letters are distinct. See Fig. 5.7.

5.4.4 Reading Mixed Data Types

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

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

will read the data

```
15 p 1.575 coffee
```

correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.

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

5.4.5 Detection of Errors in Input

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

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

will return the value 3 if the following data is typed in:

```
20 150.25 motor
```

and will return the value 1 if the following line is entered

```
20 motor 150.25
```

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

Example 5.8 The program presented in Fig. 5.8 illustrates the testing for correctness of reading of data by **scanf** function.

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

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

Note *The character '2' is assigned to the character variable c.*

Program

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

Output

```
Enter values of a, b and c
   12 3.45 A
```

Contd.

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

Fig. 5.8 *Detection of errors in* ***scanf*** *input*

Commonly used **scanf** format codes are given in Table 5.2

Table 5.2 ***Commonly used scanf Format Codes***

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

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

h for short integers
l for long integers or double
L for long double

Note *C99 adds some more format codes.*

5.4.6 Points to Remember while Using scanf

If we do not plan carefully, some 'crazy' things can happen with **scanf**. Since the I/O routines are not a part of C language, they are made available either as a separate module of the C library or as a part of

the operating system (like UNIX). New features are added to these routines from time to time as new versions of systems are released. We should consult the system reference manual before using these routines. Given below are some of the general points to keep in mind while writing a **scanf** statement.

1. All function arguments, except the control string, *must* be pointers to variables.
2. Format specifications contained in the control string should match the arguments in order.
3. Input data items must be separated by spaces and must match the variables receiving the input in the same order.
4. The reading will be terminated, when **scanf** encounters a 'mismatch' of data or a character that is not valid for the value being read.
5. When searching for a value, **scanf** ignores line boundaries and simply looks for the next appropriate character.
6. Any unread data items in a line will be considered as part of the data input line to the next scanf call.
7. When the field width specifier *w* is used, it should be large enough to contain the input data size.

Rules for scanf

- Each variable to be read must have a filed specification.
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a matching character in the user input.
- Never end the format string with whitespace. It is a fatal error!
- The scanf reads until:
  - A whitespace character is found in a numeric specification, or
  - The maximum number of characters have been read or
  - An error is detected, or
  - The end of file is reached

5.5 Formatted Output

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

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

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

Control string consists of three types of items:

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

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

A simple format specification has the following form:

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

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

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

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

5.5.1 Output of Integer Numbers

The format specification for printing an integer number is:

```
% w d
```

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

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

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

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

Example 5.9 The program in Fig. 5.9 illustrates the output of integer numbers under various formats.

```
Program
      main()
      {
            int m = 12345;
            long n = 987654;
            printf("%d\n",m);
            printf("%10d\n",m);
            printf("%010d\n",m);
            printf("%-10d\n",m);
            printf("%10ld\n",n);
            printf("%10ld\n",-n);
      }
Output
            12345
               12345
            0000012345
            12345
                  987654
                - 987654
```

Fig. 5.9 *Formatted output of integers*

5.5.2 Output of Real Numbers

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

```
% w.p f
```

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

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

```
% w.p e
```

The display takes the form

```
[ - ] m.nnnne[ ± ]xx
```

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

$$w \geq p+7$$

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

Padding the leading blanks with zeros and printing with *left-justification* are also possible by using flags 0 or – before the field width specifier **w**.

The following examples illustrate the output of the number y = 98.7654 under different format specifications:

Format	*Output*
printf("%7.4f",y)	9 8 . 7 6 5 4
printf("%7.2f",y)	␣ ␣ 9 8 . 7 7
printf("%-7.2f",y)	9 8 . 7 7 ␣ ␣
printf("%f",y)	9 8 . 7 6 5 4
printf("%10.2e",y)	␣ ␣ 9 . 8 8 e + 0 1
printf("%11.4e",-y)	– 9 . 8 7 6 5 e + 0 1
printf("%-10.2e",y)	9 . 8 8 e + 0 1 ␣ ␣
printf("%e",y)	9 . 8 7 6 5 4 0 e + 0 1

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

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

In this case, both the field width and the precision are given as arguments which will supply the values for **w** and **p**. For example,

```
printf("%*.*f",7,2,number);
```

is equivalent to

```
printf("%7.2f",number);
```

The advantage of this format is that the values for *width* and *precision* may be supplied at run time, thus making the format a *dynamic* one. For example, the above statement can be used as follows:

```
int width = 7;
int precision = 2;
........
........
printf("%*.*f", width, precision, number);
```

Example 5.10 All the options of printing a real number are illustrated in Fig. 5.10.

```
Program
  main()
  {
     float y = 98.7654;
```

Contd.

```
    printf("%7.4f\n", y);
    printf("%f\n", y);
    printf("%7.2f\n", y);
    printf("%-7.2f\n", y);
    printf("%07.2f\n", y);
    printf("%*.*f", 7, 2, y);
    printf("\n");
    printf("%10.2e\n", y);
    printf("%12.4e\n", -y);
    printf("%-10.2e\n", y);
    printf("%e\n", y);
}
Output
    98.7654
    98.765404
    98.77
    98.77
    0098.77
    98.77
    9.88e+001
    -9.8765e+001
    9.88e+001
    9.876540e+001
```

Fig. 5.10 *Formatted output of* ***real*** *numbers*

5.5.3 Printing of a Single Character

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

```
%wc
```

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

5.5.4 Printing of Strings

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

```
%w.ps
```

where *w* specifies the field width for display and *p* instructs that only the first p characters of the string are to be displayed. The display is *right-justified.*

The following examples show the effect of variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including blanks).

Specification *Output*

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

Example 5.11 Printing of characters and strings is illustrated in Fig. 5.11.

Program

```
main()
{
char x = 'A';
char name[20] = "ANIL KUMAR GUPTA";
printf("OUTPUT OF CHARACTERS\n\n");
printf("%c\n%3c\n%5c\n", x,x,x);
printf("%3c\n%c\n", x,x);
printf("\n");
printf("OUTPUT OF STRINGS\n\n");
printf("%s\n", name);
printf("%20s\n", name);
printf("%20.10s\n", name);
printf("%.5s\n", name);
printf("%-20.10s\n", name);
printf("%5s\n", name);
}
```

Output

```
OUTPUT OF CHARACTERS
A
   A
      A
   A
A
OUTPUT OF STRINGS
```

Contd.

```
ANIL KUMAR GUPTA
      ANIL KUMAR GUPTA
         ANIL KUMAR
ANIL
ANIL KOLMER
ANIL KUMAR GUPTA
```

Fig. 5.11 *Printing of characters and strings*

5.5.5 Mixed Data Output

It is permitted to mix data types in one **printf** statement. For example, the statement of the type is valid.

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

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

Table 5.3 ***Commonly used printf Format Codes***

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

Table 5.4 ***Commonly used Output Format Flags***

Flag	Meaning
–	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	Causes leading zeros to appear.
# (with o or x)	Causes octal and hex items to be preceded by O and Ox, respectively.
# (with e, f or g)	Causes a decimal point to be present in all floating point numbers, even if it is whole number. Also prevents the truncation of trailing zeros in g-type conversion

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

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

5.5.6 Enhancing the Readability of Output

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

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

The system usually provides two blank spaces between the numbers. However, this can be increased by selecting a suitable field width for the numbers or by introducing a 'tab' character between the specifications. For example, the statement

```
printf("a = %d\t b = %d", a, b);
```

will provide four blank spaces between the two fields. We can also print them on two separate lines by using the statement

```
printf("a = %d\n b = %d", a, b);
```

Messages and headings can be printed by using the character strings directly in the **printf** statement. Examples:

```
printf("\n OUTPUT RESULTS \n");
printf("Code\t Name\t Age\n");
printf("Error in input data\n");
printf("Enter your name\n");
```

Just Remember

- While using **getchar** function, care should be exercised to clear any unwanted characters in the input stream.
- Do not forget to include <**stdio.h**> headerfiles when using functions from standard input/output library.
- Do not forget to include <**ctype.h**> header file when using functions from character handling library.
- Provide proper field specifications for every variable to be read or printed.
- Enclose format control strings in double quotes.
- Do not forget to use address operator & for basic type variables in the input list of **scanf**.
- Use double quotes for character string constants.

- Use single quotes for single character constants.
- Provide sufficient field to handle a value to be printed.
- Be aware of the situations where output may be imprecise due to formatting.
- Do not specify any precision in input field specifications.
- Do not provide any white-space at the end of format string of a **scanf** statement.
- Do not forget to close the format string in the **scanf** or **printf** statement with double quotes.
- Using an incorrect conversion code for data type being read or written will result in runtime error.
- Do not forget the comma after the format string in **scanf** and **printf** statements.
- Not separating read and write arguments is an error.
- Do not use commas in the format string of a **scanf** statement.
- Using an address operator & with a variable in the **printf** statement will result in runtime error.

Review Questions

5.1 State whether the following statements are *true* or *false*.

(a) The purpose of the header file <studio.h> is to store the programs created by the users.
(b) The C standard function that receives a single character from the keyboard is **getchar**.
(c) The **getchar** cannot be used to read a line of text from the keyboard.
(d) The input list in a **scanf** statement can contain one or more variables.
(e) When an input stream contains more data items than the number of specifications in a **scanf** statement, the unused items will be used by the next **scanf** call in the program.
(f) Format specifiers for output convert internal representations for data to readable characters.
(g) Variables form a legal element of the format control string of a **printf** statement.
(h) The **scanf** function cannot be used to read a single character from the keyboard.
(i) The format specification %+ –8d prints an integer left-justified in a field width of 8 with a plus sign, if the number is positive.
(j) If the field width of a format specifier is larger than the actual width of the value, the value is printed right-justified in the field.
(k) The print list in a **printf** statement can contain function calls.
(l) The format specification %5s will print only the first 5 characters of a given string to be printed.

5.2 Fill in the blanks in the following statements.

(a) The _____ specification is used to read or write a short integer.
(b) The conversion specifier _____ is used to print integers in hexadecimal form.
(c) For using character functions, we must include the header file _____ in the program.
(d) For reading a double type value, we must use the specification _____.
(e) The specification _____ is used to read a data from input list and discard it without assigning it to many variables.
(f) The specification _____ may be used in **scanf** to terminate reading at the encounter of a particular character.
(g) The specification %[ ] is used for reading strings that contain _____.

(h) By default, the real numbers are printed with a precision of _____ decimal places.
(i) To print the data left-justified, we must use _____ in the field specification.
(j) The specifier _____ prints floating-point values in the scientific notation.

5.3 Distinguish between the following pairs:
(a) ***getchar*** and ***scanf*** functions.
(b) %s and %c specifications for reading.
(c) %s and %[ ] specifications for reading.
(d) %g and %f specification for printing.
(e) %f and %e specifications for printing.

5.4 Write **scanf** statements to read the following data lists:
(a) 78 B 45 (b) 123 1.23 45A
(c) 15-10-2002 (d) 10 TRUE 20

5.5 State the outputs produced by the following **printf** statements.
(a) printf (“%d%c%f”, 10, ‘x’, 1.23);
(b) printf (“%2d %c %4.2f”, 1234,, ‘x’, 1.23);
(c) printf (“%d\t%4.2f”, 1234, 456);
(d) printf (“\”%08.2f\””, 123.4);
(e) printf (“%d%d %d”, 10, 20);

For questions 5.6 to 5.10 assume that the following declarations have been made in the program:

```
int year, count;
float amount, price;
char code, city[10];
double root;
```

5.6 State errors, if any, in the following input statements.
(a) scanf(“%c%f%d”, city, &price, &year);
(b) scanf(“%s%d”, city, amount);
(c) scanf(“%f, %d, &amount, &year);
(d) scanf(\n”%f”, root);
(e) scanf(“%c %d %ld”, *code, &count, Root);

5.7 What will be the values stored in the variables **year** and **code** when the data

1988, x

is keyed in as a response to the following statements:
(a) scanf(“%d %c”, &year, &code);
(b) scanf(“%c %d”, &year, &code);
(c) scanf(“%d %c”, &code, &year);
(d) scanf(“%s %c”, &year, &code);

5.8 The variables **count, price**, and **city** have the following values:

count <—— 1275
price <—— –235.74
city <—— Cambridge

Show the exact output that the following output statements will produce:
(a) printf(“%d %f”, count, price);

(b) printf("%2d\n%f", count, price);
(c) printf("%d %f", price, count);
(d) printf("%10dxxxx%5.2f",count, price);
(e) printf("%s", city);
(f) printf(%-10d %-15s", count, city);

5.9 State what (if anything) is wrong with each of the following output statements:
(a) printf(%d 7.2%f, year, amount);
(b) printf("%-s, %c"\n, city, code);
(c) printf("%f, %d, %s, price, count, city);
(d) printf("%c%d%f\n", amount, code, year);

5.10 In response to the input statement
scanf("%4d%*%d", &year, &code, &count);
the following data is keyed in:

19883745

What values does the computer assign to the variables **year, code**, and **count**?

5.11 How can we use the **getchar**( ) function to read multicharacter strings?
5.12 How can we use the **putchar** ( ) function to output multicharacter strings?
5.13 What is the purpose of **scanf**( ) function?
5.14 Describe the purpose of commonly used conversion characters in a **scanf**( ) function.
5.15 What happens when an input data item contains
(a) more characters than the specified field width and
(b) fewer characters than the specified field width?
5.16 What is the purpose of **print**( ) function?
5.17 Describe the purpose of commonly used conversion characters in a **printf**( ) function.
5.18 How does a control string in a **printf**( ) function differ from the control string in a **scanf**( ) function?
5.19 What happens if an output data item contains
(a) more characters than the specified field width and
(b) fewer characters than the specified field width?
5.20 How are the unrecognized characters within the control string are interpreted in
(a) **scanf** function; and
(b) **printf** function?

Programming Exercises

5.1 Given the string "WORDPROCESSING", write a program to read the string from the terminal and display the same in the following formats:
(a) WORD PROCESSING
(b) WORD
PROCESSING
(c) W.P.

5.2 Write a program to read the values of x and y and print the results of the following expressions in one line:

(a) $\frac{x + y}{x \quad y}$ (b) $\frac{x + y}{2}$ (c) (x+y)(x–y)

5.3 Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form:

35.7 50.21 – 23.73 – 46.45

5.4 Write a program that reads 4 floating point values in the range, 0.0 to 20.0, and prints a horizontal bar chart to represent these values using the character * as the fill character. For the purpose of the chart, the values may be rounded off to the nearest integer. For example, the value 4.36 should be represented as follows.

```
*   *   *   *
*   *   *   *   4.36
*   *   *   *
```

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

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

```
                 45
               × 37
             -----
7 × 45 is      315
3 × 45 is      135
             -----
Add them      1665
             -----
```

5.6 Write a program to read three integers from the keyboard using one **scanf** statement and output them on one line using:

(a) three **printf** statements,
(b) only one **printf** with conversion specifiers, and
(c) only one **printf** without conversion specifiers.

5.7 Write a program that prints the value 10.45678 in exponential format with the following specifications:

(a) correct to two decimal places;
(b) correct to four decimal places; and
(c) correct to eight decimal places.

5.8 Write a program to print the value 345.6789 in fixed-point format with the following specifications:

(a) correct to two decimal places;
(b) correct to five decimal places; and
(c) correct to zero decimal places.

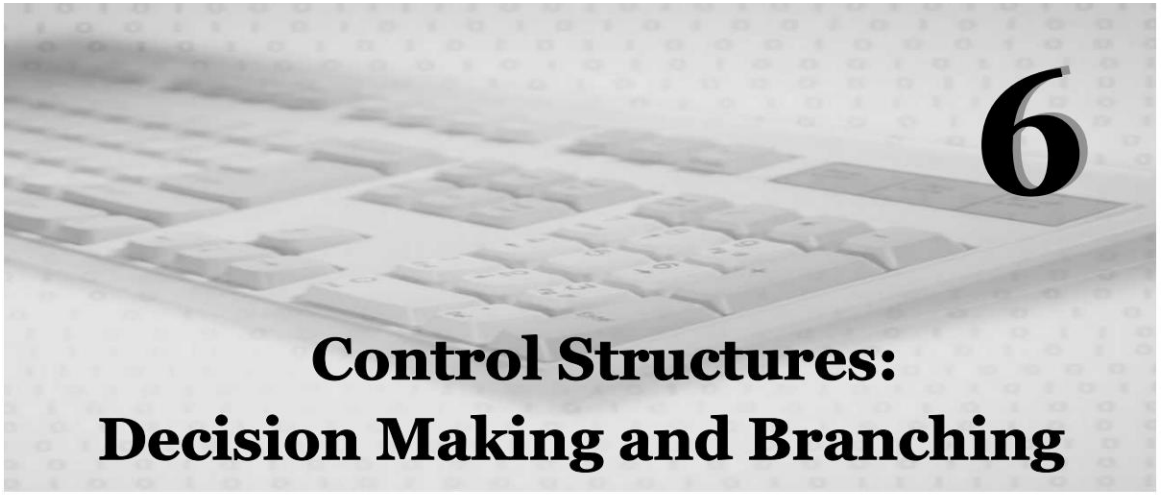
5.9 Write a program to read the name ANIL KUMAR GUPTA in three parts using the **scanf** statement and to display the same in the following format using the **printf** statement.

(a) ANIL K. GUPTA
(b) A.K. GUPTA
(c) GUPTA A.K.

5.10 Write a program to read and display the following table of data.

Name	Code	Price
Fan	67831	1234.50
Motor	450	5786.70

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



6 Control Structures: Decision Making and Branching

6.1 Introduction

We have seen that a C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary. However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

C language possesses such decision-making capabilities by supporting the following statements:

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

These statements are popularly known as *decision-making statements*. Since these statements 'control' the flow of execution, they are also known as *control statements*.

We have already used some of these statements in the earlier examples. Here, we shall discuss their features, capabilities and applications in more detail.

6.2 Decision Making with if Statement

The **if** statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

```
if (test expression)
```

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (or non-zero) or 'false' (zero), it transfers the control to a particular statement. This point of program has two *paths* to follow, one for the *true* condition and the other for the *false* condition as shown in Fig. 6.1.

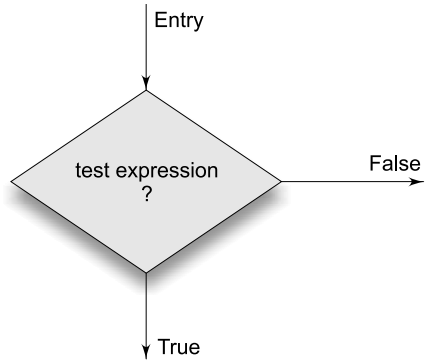


Fig. 6.1 *Two-way branching*

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

1. **if** (bank balance is zero)
   borrow money
2. **if** (room is dark)
   put on lights
3. **if** (code is 1)
   person is male
4. **if** (age is more than 55)
   person is retired

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:

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

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

6.3 Simple if Statement

The general form of a simple **if** statement is

```
if (test expression)
 {
       statement-block;
 }
statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x*. Remember, when the condition is true both the statement-block and the statement-x are executed in sequence. This is illustrated in Fig. 6.2.

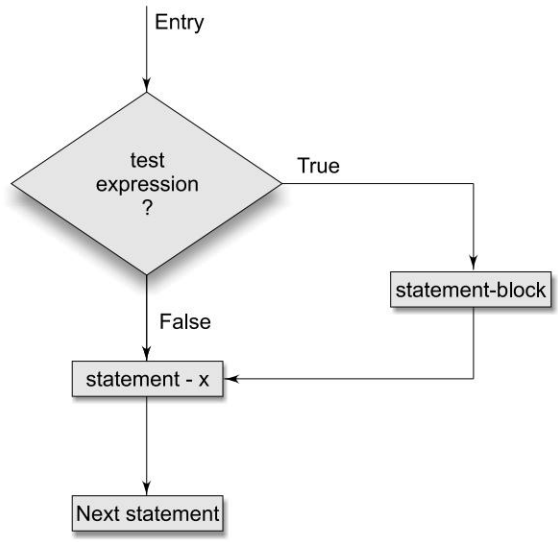


Fig. 6.2 *Flowchart of simple* ***if*** *control*

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

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

The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bonus_marks are added to his marks before they are printed. For others, bonus_marks are not added.

Example 6.1 The program in Fig. 6.3 reads four values a, b, c, and d from the terminal and evaluates the ratio of (a+b) to (c–d) and prints the result, if c–d is not equal to zero.

The program given in Fig. 6.3 has been run for two sets of data to see that the paths function properly. The result of the first run is printed as,

Ratio = –3.181818

```
Program
  main()
  {
        int a, b, c, d;
        float ratio;

        printf("Enter four integer values\n");
        scanf("%d %d %d %d", &a, &b, &c, &d);

        if (c-d != 0) /* Execute statement block */
        {
              ratio = (float)(a+b)/(float)(c-d);
              printf("Ratio = %f\n", ratio);
        }
  }
Output
  Enter four integer values
  12 23 34 45
  Ratio = -3.181818

  Enter four integer values
  12  23  34  34
```

Fig. 6.3 *Illustration of simple* ***if*** *statement*

The second run has neither produced any results nor any message. During the second run, the value of (c–d) is equal to zero and therefore, the statements contained in the statement-block are skipped. Since no other statement follows the statement-block, program stops without producing any output.

Note the use of **float** conversion in the statement evaluating the **ratio**. This is necessary to avoid truncation due to integer division. Remember, the output of the first run –3.181818 is printed correct to six decimal places. The answer contains a round off error. If we wish to have higher accuracy, we must use **double** or **long double** data type.

The simple **if** is often used for counting purposes. The Example 6.2 illustrates this.

Example 6.2 The program in Fig. 6.4 counts the number of boys whose weight is less than 50 kg and height is greater than 170 cm.

The program has to test two conditions, one for weight and another for height. This is done using the compound relation

```
if (weight < 50 && height > 170)
```

This would have been equivalently done using two **if** statements as follows:

```
if (weight < 50)
  if (height > 170)
    count = count +1;
```

If the value of **weight** is less than 50, then the following statement is executed, which in turn is another **if** statement. This **if** statement tests **height** and if the **height** is greater than 170, then the **count** is incremented by 1.

```
Program
   main()
   {
          int count, i;
          float weight, height;
          count = 0;
          printf("Enter weight and height for 10 boys\n");

          for (i =1; i <= 10; i++)
          {
                scanf("%f %f", &weight, &height);
                if (weight < 50 && height > 170)
                      count = count + 1;
          }
          printf("Number of boys with weight < 50 kg\n");
          printf("and height > 170 cm = %d\n", count);
   }
Output
   Enter weight and height for 10 boys
   45    176.5
   55    174.2
   47    168.0
   49    170.7
   54    169.0
   53    170.5
   49    1676.0
   48    175.0
   47    167
   51    170
   Number of boys with weight < 50 kg
   and height > 170 cm = 3
```

Fig. 6.4 *Use of* ***if*** *for counting*

Applying De Morgan's Rule

While designing decision statements, we often come across a situation where the logical NOT operator is applied to a compound logical expression, like !(x&&y||!z). However, a positive logic is always easy to read and comprehend than a negative logic. In such cases, we may apply what is known as **De Morgan's** rule to make the total expression positive. This rule is as follows:

"Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators."

That is,

x becomes !x
!x becomes x
&& becomes ||
|| becomes &&

Examples:

!(x && y || !z) becomes !x || !y && z
!(x <=0 || !condition) becomes x >0&& condition

6.4 The if.....Else Statement

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

```
If (test expression)
   {
      True-block statement(s)
   }
else
   {
      False-block statement(s)
   }
statement-x
```

If the *test expression* is true, then the *true-block statement(s),* immediately following the **if** statements are executed; otherwise, the *false-block statement(s)* are executed. In either case, either *true-block* or *false-block* will be executed, not both. This is illustrated in Fig. 6.5. In both the cases, the control is transferred subsequently to the statement-x.

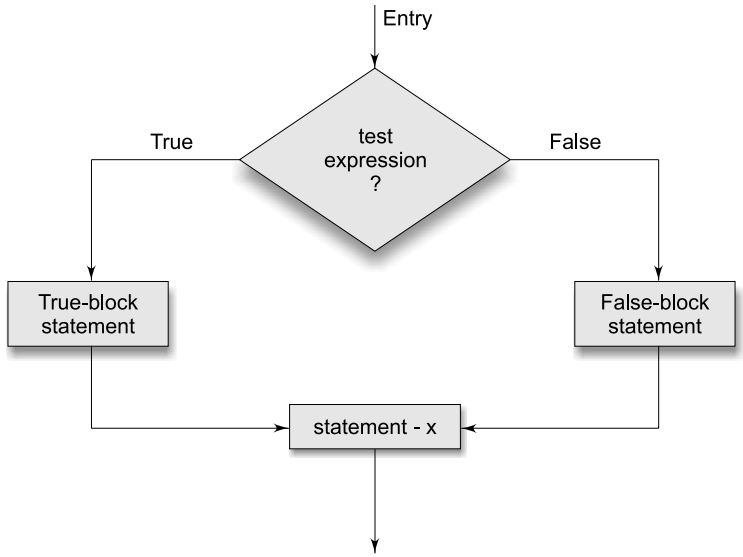


Fig. 6.5 *Flowchart of* ***if.....else*** *control*

Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The program statement to do this may be written as follows:

```
.........
.........
if (code == 1)
   boy = boy + 1;
if (code == 2)
   girl = girl+1;
.........
.........
```

The first test determines whether or not the student is a boy. If yes, the number of boys is increased by 1 and the program continues to the second test. The second test again determines whether the student is a girl. This is unnecessary. Once a student is identified as a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not both. The above program segment can be modified using the **else** clause as follows:

```
..........
..........
if (code == 1)
   boy = boy + 1;
else
   girl = girl + 1;
xxxxxxxxxx
..........
```

Here, if the code is equal to 1, the statement **boy = boy + 1**; is executed and the control is transferred to the statement **xxxxxx**, after skipping the else part. If the code is not equal to 1, the statement **boy = boy + 1**; is skipped and the statement in the **else** part **girl = girl + 1**; is executed before the control reaches the statement **xxxxxxxx.**

Consider the program given in Fig. 6.3. When the value (c–d) is zero, the ratio is not calculated and the program stops without any message. In such cases we may not know whether the program stopped due to a zero value or some other error. This program can be improved by adding the **else** clause as follows:

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

Example 6.3 A program to evaluate the power series:

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

is given in Fig. 6.6. It uses **if** ..... **else** to test the accuracy

The power series contains the recurrence relationship of the type

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

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

$$T_0 = 1$$

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

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

Program

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

Output

```
Enter value of x:0
Terms = 2 Sum = 1.000000
Enter value of x:0.1
Terms = 5 Sum = 1.105171
Enter value of x:0.5
Terms = 7 Sum = 1.648720
```

Contd.

```
Enter value of x:0.75
Terms = 8 Sum = 2.116997
Enter value of x:0.99
Terms = 9 Sum = 2.691232
Enter value of x:1
Terms = 9 Sum = 2.718279
```

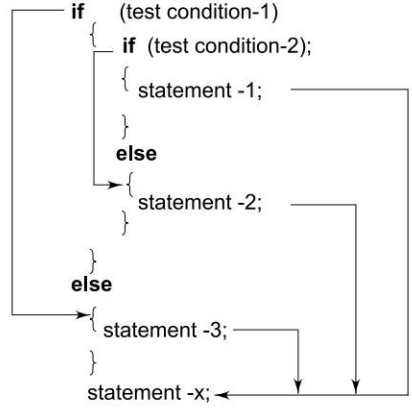
Fig. 6.6 *Illustration of* ***if...else*** *statement*

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

6.5 Nesting of if....Else Statements

When a series of decisions are involved, we may have to use more than one **if...else** statement in *nested* form as shown below:

The logic of execution is illustrated in Fig. 6.7. If the *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.



A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the balance held on 31st December is given to every one, irrespective of their balance, and 5 per cent is given to female account holders if their balance is more than Rs. 5000. This logic can be coded as follows:

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

```
        }
        else
        {
            bonus = 0.02 * balance;
        }
        balance = balance + bonus;
        .........
        .........
```

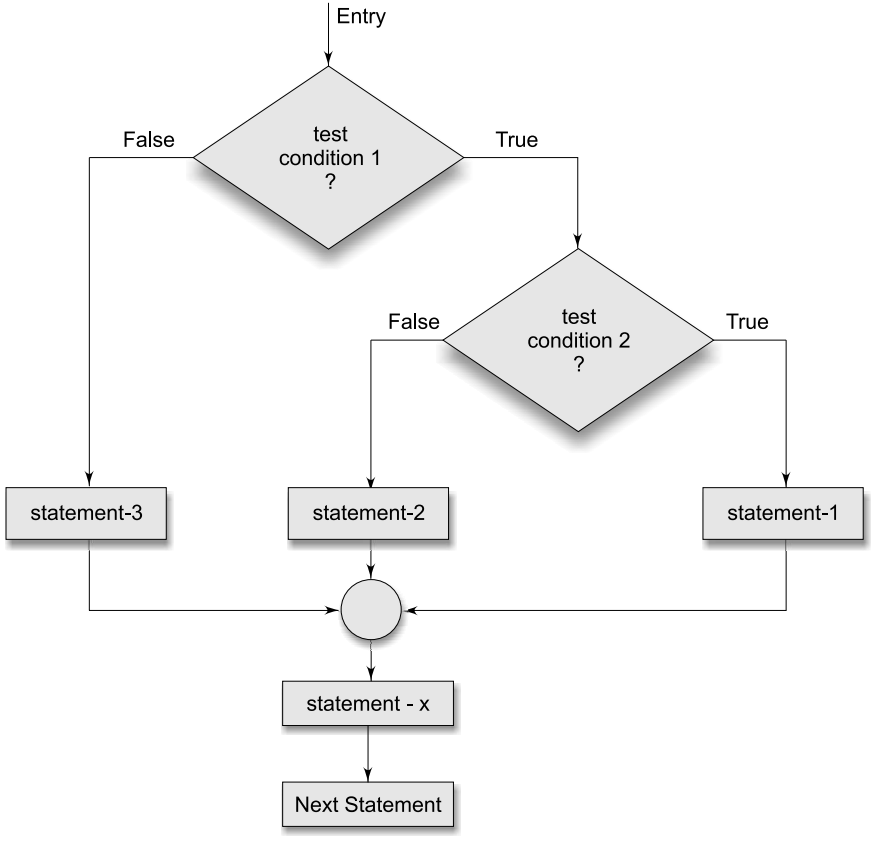


Fig. 6.7 *Flowchart of nested* ***if...else*** *statements*

When nesting, care should be exercised to match every **if** with an **else**. Consider the following alternative to the above program (which looks right at the first sight):

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

There is an ambiguity as to over which **if** the **else** belongs to. In C, an else is linked to the closest non-terminated **if.** Therefore, the **else** is associated with the inner **if** and there is no else option for the outer **if**. This means that the computer is trying to execute the statement

```
balance = balance + bonus;
```

without really calculating the bonus for the male account holders.

Consider another alternative, which also looks correct:

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

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

Example 6.4 The program in Fig. 6.8 selects and prints the largest of the three numbers using nested **if**....**else** statements.

```
Program
   main()
   {
   float A, B, C;
   printf("Enter three values\n");
   scanf("%f %f %f", &A, &B, &C);
   printf("\nLargest value is ");
   if (A>B)
   {
      if (A>C)
         printf("%f\n", A);
      else
         printf("%f\n", C);
   }
   else
   {
      if (C>B)
         printf("%f\n", C);
      else
         printf("%f\n", B);
   }
     }
```

Contd

```
Output
   Enter three values
   23445 67379 88843
   Largest value is 88843.000000
```

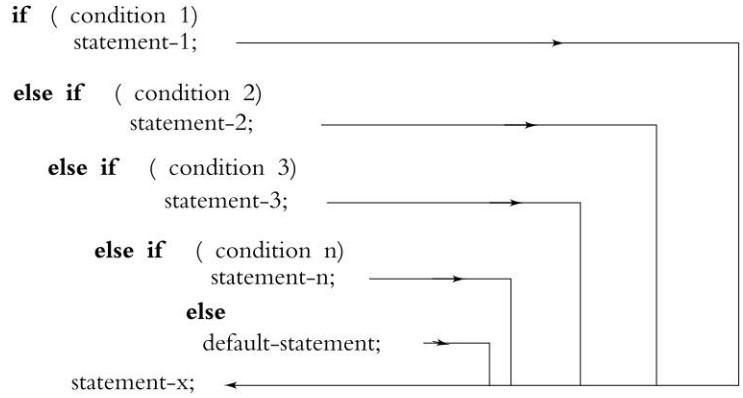
Fig. 6.8 *Selecting the largest of three numbers*

Dangling Else Problem

One of the classic problems encountered when we start using nested **if….else** statements is the dangling else. This occurs when a matching **else** is not available for an **if**. The answer to this problem is very simple. Always match an **else** to the most recent unmatched **if** in the current block. In some cases, it is possible that the false condition is not required. In such situations, **else** statement may be omitted "**else** is always paired with the most recent unpaired **if**"

6.6 The Else if Ladder

There is another way of putting **if**s together when multipath decisions are involved. A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if**. It takes the following general form:



This construct is known as the **else if** ladder. The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final **else** containing the *default-statement* will be executed. Figure 6.9 shows the logic of execution of **else if** ladder statements.

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

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

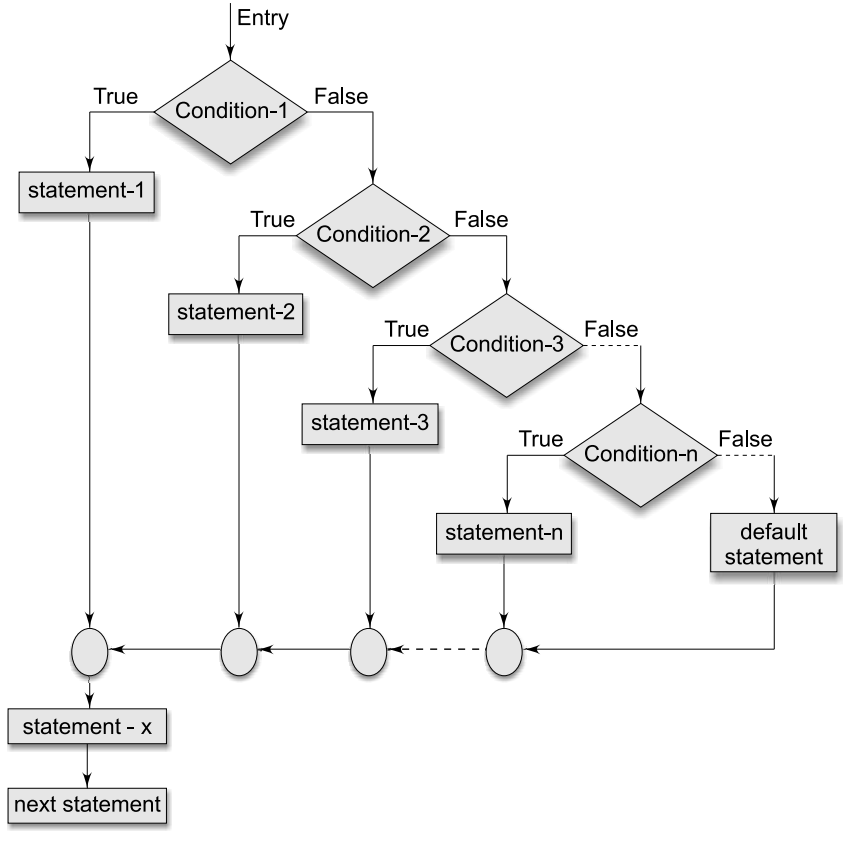


Fig. 6.9 *Flowchart of* ***else..if*** *ladder*

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

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

Consider another example given below:

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

Code numbers other than 1, 2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested **if...else** statements.

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

In such situations, the choice is left to the programmer. However, in order to choose an **if** structure that is both effective and efficient, it is important that the programmer is fully aware of the various forms of an **if** statement and the rules governing their nesting.

Example 6.5 An electric power distribution company charges its domestic consumers as follows:

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

The program in Fig. 6.10 reads the customer number and power consumed and prints the amount to be paid by the customer.

```
  Program
     main()
     {
        int units, custnum;
        float charges;
```

```
        printf("Enter CUSTOMER NO. and UNITS consumed\n");
        scanf("%d %d", &custnum, &units);
        if (units <= 200)
           charges = 0.5 * units;
        else if   (units <= 400)
                    charges = 100 + 0.65 * (units - 200);
                       else if (units <= 600)
                       charges = 230 + 0.8 * (units - 400);
                          else
                          charges = 390 + (units - 600);
        printf("\n\nCustomer No: %d: Charges = %.2f\n",
           custnum, charges);
    }
Output
    Enter CUSTOMER NO. and UNITS consumed 101 150
    Customer No:101 Charges = 75.00
    Enter CUSTOMER NO. and UNITS consumed 202 225
    Customer No:202 Charges = 116.25
    Enter CUSTOMER NO. and UNITS consumed 303 375
    Customer No:303 Charges = 213.75
    Enter CUSTOMER NO. and UNITS consumed 404 520
    Customer No:404 Charges = 326.00
    Enter CUSTOMER NO. and UNITS consumed 505 625
    Customer No:505 Charges = 415.00
```

Fig. 6.10 *Illustration of* ***else..if*** *ladder*

Rules for Indentation

When using control structures, a statement often controls many other statements that follow it. In such situations it is a good practice to use *indentation* to show that the indented statements are dependent on the preceding controlling statement. Some guidelines that could be followed while using indentation are listed below:

- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
- Align vertically else clause with their matching if clause.
- Use braces on separate lines to identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of the braces.
- Align the opening and closing braces.
- Use appropriate comments to signify the beginning and end of blocks.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line.

6.7 The Switch Statement

We have seen that when one of the many alternatives is to be selected, we can use an **if** statement to control the selection. However, the complexity of such a program increases dramatically when the

number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a **switch.** The **switch** statement tests the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statements associated with that **case** is executed. The general form of the **switch** statement is as shown below:

```
switch (expression)
{
   case value-1:
                block-1
                break;
   case value-2:
                block-2
                break;
   ......
   ......
   default:
                default-block
                break;
}
statement-x;
```

The *expression* is an integer expression or characters. *Value-1, value-2* ..... are constants or constant expressions (evaluable to an integral constant) and are known as *case labels*. Each of these values should be unique within a **switch** statement. **block-1, block-2** .... are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that **case** labels end with a colon (:).

When the **switch** is executed, the value of the expression is successfully compared against the values *value-1, value-2,....* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The **break** statement at the end of each block signals the end of a particular case and causes an exit from the **switch** statement, transferring the control to the **statement-x** following the **switch.**

The **default** is an optional case. When present, it will be executed if the value of the *expression* does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the **statement-x**. (ANSI C permits the use of as many as 257 case labels).

The selection process of **switch** statement is illustrated in the flow chart shown in Fig. 6.11.

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

```
---
---
index = marks/10
switch (index)
{
  case 10:
  case 9:
```

```
  case 8:
        grade = "Honours";
        break;
  case 7:
  case 6:
        grade = "First Division";
        break;
  case 5:
        grade = "Second Division";
        break;
  case 4:
        grade = "Third Division";
        break;
  default:
        grade = "Fail";
        break;
}
printf("%s\n", grade);
---
---
```

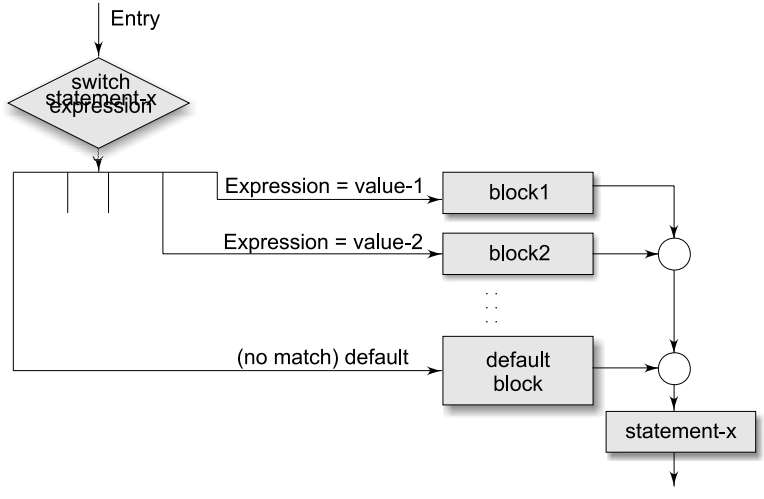


Fig. 6.11 *Selection process of the* ***switch*** *statement*

Note that we have used a conversion statement

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

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

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

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

```
grade = "Honours";
break;
```

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

The **switch** statement is often used for menu selection. For example:

```
____
____
printf(" TRAVEL GUIDE\n\n");
printf(" A Air Timings\n" );
printf(" T Train Timings\n");
printf(" B Bus Service\n" );
printf(" X To skip\n" );
printf("\n Enter your choice\n");
character = getchar();
switch (character)
{
  case 'A' :
              air-display();
              break;
  case 'B' :
              bus-display();
              break;
  case 'T' :
              train-display();
              break;
default :
              printf(" No choice\n");
}
  ____
  ____
```

It is possible to nest the **switch** statements. That is, a **switch** may be part of a **case** statement. ANSI C permits 15 levels of nesting.

Rules for Switch Statement

- The **switch** expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with semicolon.
- The **break** statement transfers the control out of the **switch** statement.
- The **break** statement is optional. That is, two or more case labels may belong to the same statements.
- The **default** label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one **default** label.
- The **default** may be placed anywhere but usually placed at the end.
- It is permitted to nest **switch** statements.

6.8 The ? : Operator

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

```
Conditional expression ? expression1 : expression2
```

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

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

can be written as

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

Consider the evaluation of the following function:

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

$y = 2x + 5$ for $x > 2$

This can be evaluated using the conditional operator as follows:

```
y = ( x > 2 ) ? (2 * x + 5) : (1.5 * x + 3);
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x < 40 \end{cases}$$

This complex equation can be written as

```
salary = (x != 40) ? ((x < 40) ? (4*x+100) : (4.5*x+150)) : 300;
```

The same can be evaluated using **if...else** statements as follows:

```
if (x <= 40)
      if (x < 40)
         salary = 4 * x+100;
      else
         salary = 300;
else
         salary = 4.5 * x+150;
```

When the conditional operator is used, the code becomes more concise and perhaps, more efficient. However, the readability is poor. It is better to use **if** statements when more than a single nesting of conditional operator is required.

Example 6.6 An employee can apply for a loan at the beginning of every six months, but he will be sanctioned the amount according to the following company rules:

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

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

A program to process loan applications and to sanction loans is given in Fig. 6.12.

Program

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

Output

```
Enter the values of previous two loans:
0 20000
Enter the value of new loan:
45000
Previous loans pending:
0 20000
```

Contd

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

Fig. 6.12 *Illustration of the conditional operator*

The program uses the following variables:

loan3	-	present loan amount requested
loan2	-	previous loan amount pending
loan1	-	previous to previous loan pending
sum23	-	sum of loan2 and loan3
sancloan	-	loan sanctioned

The rules for sanctioning new loan are:

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

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

Some Guidelines for Writing Multiway Selection Statements

Complex mulitiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are guidelines that would help inprove redability and facilitate maintenance.

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

6.9 The Goto Statement

So far we have discussed ways of controlling the flow of execution based on certain specified conditions. Like many other languages, C supports the **goto** statement to branch unconditionally from one point to

another in the program. Although it may not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of **goto** might be desirable.

The **goto** requires a *label* in order to identify the place where the branch is to be made. A *label* is any valid variable name, and must be followed by a colon. The *label* is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and *label* statements are shown below:



Forward jump Backward jump

The *label*: can be anywhere in the program either before or after the **goto** label; statement.
During running of a program when a statement like

goto begin;

is met, the flow of control will jump to the statement immediately following the label **begin**:. This happens unconditionally.

Note that a **goto** breaks the normal sequential execution of the program. If the *label:* is before the statement **goto** *label*; a *loop* will be formed and some statements will be executed repeatedly. Such a jump is known as a *backward jump*. On the other hand, if the *label:* is placed after the **goto** *label*; some statements will be skipped and the jump is known as a *forward jump*.

A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

```
main()
{
     double x, y;
     read:
     scanf("%f", &x);
     if (x < 0) goto read;
     y = sqrt(x);
     printf("%f %f\n", x, y);
     goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statement and the other to skip any further computation when the number is negative.

Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite loop*. The computer goes round and round until we take some special steps to terminate the loop. Such infinite loops should be avoided. Example 6.7 illustrates how such infinite loops can be eliminated.

Example 6.7 Program presented in Fig. 6.13 illustrates the use of the goto statement. The program evaluates the square root for five numbers. The variable count keeps the count of numbers read. When count is less than or equal to 5, goto read; directs the control to the label read; otherwise, the program prints a message and stops.

```
Program
  #include <math.h>
  main()
  {
        double x, y;
        int count;
        count = 1;
        printf("Enter FIVE real values in a LINE \n");
  read:
        scanf("%lf", &x);
        printf("\n");
        if (x < 0)
           printf("Value - %d is negative\n",count);
        else
        {
           y = sqrt(x);
           printf("%lf\t %lf\n", x, y);
        }
        count = count + 1;
        if (count <= 5)
  goto read;
        printf("\nEnd of computation");
  }
Output
  Enter FIVE real values in a LINE
  50.70 40 -36 75 11.25
  50.750000       7.123903
  40.000000       6.324555
  Value -3 is negative
  75.000000       8.660254
  11.250000       3.354102
  End of computation
```

Fig. 6.13 *Use of the **goto** statement*

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

```
————
————
while (————)
{
   for (————)
   {
   ————
   ————
   if (————)goto end_of_program;
   ————
   }                                          Jumping
————                                          out of
————                                          loops
}
end_of_program:
```

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

Just Remember

- Be aware of dangling **else** statements.
- Be aware of any side effects in the control expression such as if(x++).
- Use braces to encapsulate the statements in **if** and **else** clauses of an if…. else statement.
- Check the use of =operator in place of the equal operator = =.
- Do not give any spaces between the two symbols of relational operators = =, !=, >= and <=.
- Writing !=, >= and <= operators like =!, => and =< is an error.
- Remember to use two ampersands (&&) and two bars (||) for logical operators. Use of single operators will result in logical errors.
- Do not forget to place parentheses for the if expression.
- It is an error to place a semicolon after the if expression.
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal.
- Do not forget to use a break statement when the cases in a switch statement are exclusive.
- Although it is optional, it is a good programming practice to use the default clause in a switch statement.
- It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.)
- Do not use the same constant in two case labels in a switch statement.
- Avoid using operands that have side effects in a logical binary expression such as (x––&&++y). The second operand may not be evaluated at all.
- Try to use simple logical expressions.

Review Questions

6.1 State whether the following are *true* or *false*:

(a) When **if** statements are nested, the last **else** gets associated with the nearest **if** without an **else**.

(b) One **if** can have more than one **else** clause.

(c) A **switch** statement can always be replaced by a series of **if..else** statements.

(d) A **switch** expression can be of any type.

(e) A program stops its execution when a **break** statement is encountered.

(f) Each expression in the **else if** must test the same variable.

(g) Any expression can be used for the **if** expression.

(h) Each case label can have only one statement.

(i) The **default** case is required in the **switch** statement.

(j) The predicate !((x >= 10)|(y = = 5)) is equivalent to (x < 10) && (y !=5).

6.2 Fill in the blanks in the following statements.

(a) The _____ operator is true only when both the operands are true.

(b) Multiway selection can be accomplished using an **else if** statement or the _____ statement.

(c) The _____ statement when executed in a **switch** statement causes immediate exit from the structure.

(d) The ternary conditional expression using the operator ?: could be easily coded using _____ statement.

(e) The expression ! (x ! = y) can be replaced by the expression _____.

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

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

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

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

6.4 The following is a segment of a program:

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

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

6.5 Rewrite each of the following without using compound relations:

(a)
```
if (grade <= 59 && grade >= 50)
     second = second + 1;
```

(b)
```
if (number > 100 || number < 0)
     printf(" Out of range");
else
     sum = sum + number;
```

(c)
```
if ((M1 > 60 && M2 > 60) || T > 200)
     printf(" Admitted\n");
else
     printf(" Not admitted\n");
```

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

(a) `x = = 10 && x > 10 && !x`

(b) `x = = 10 || x > 10 && ! x`

(c) `x = = 10 && x > 10 || ! x`

(d) `x = = 10 || x > 10 || !x`

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

(a) `switch (y);`

(b) `case 10;`

(c) `switch (x + y)`

(d) `switch (x) {case 2: y = x + y; break};`

6.8 Simplify the following compound logical expressions

(a) `!(x <=10)`

(b) `!(x = = 10) ||! ( (y = = 5) || (z < 0) )`

(c) `! ( (x +y = = z) && !(z > 5)`

(d) `!( (x <=5) && (y = = 10) & & (z < 5) )`

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

(a)
```
if (x && y)
     x = 10;
else
     y = 10;
```

(b)
```
if (x || y || z)
     y = 10;
else
     z = 0;
```

(c)
```
if (x)
 if (y)
     z = 10;
else
     z = 0;
```

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

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

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

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

6.11 Find the error, if any, in the following statements:

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

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

(c)
```
if (x = 10)
printf ("Good" ) ;
```

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

6.12 What is the output of the following program?

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

6.13 What is the output of the following program?

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

6.14 What is the output of the following program?

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

6.15 What is the output of the following program?

```
main( )
{
       int m, n, p ;
       for ( m = 0; m < 3; m++ )
       for (n = 0; n<3; n++ )
       for ( p = 0; p < 3;; p++ )
       if ( m + n + p == 2 )
       goto print;
       print :
       printf("%d, %d, %d", m, n, p);
}
```

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

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

6.17 What will be the output when the following segment is executed?

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

6.18 What will be the output when the following segment is executed?

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

6.19 What will be the output of the following segment when executed?

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

6.20 What will be output of the following segment when executed?

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

Programming Exercises

6.1 Write a program to determine whether a given number is 'odd' or 'even' and print the message
NUMBER IS EVEN
or
NUMBER IS ODD
(a) without using **else** option, and
(b) with **else** option.

6.2 Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.

6.3 A set of two linear equations with two unknowns x1 and x2 is given below:

$$ax_1 + bx_2 = m$$
$$cx_1 + dx_2 = n$$

The set has a unique solution

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

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

provided the denominator ad – cb is not equal to zero.

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

6.4 Given a list of marks ranging from 0 to 100, write a program to compute and print the number of students:

(a) who have obtained more than 80 marks,
(b) who have obtained more than 60 marks,
(c) who have obtained more than 40 marks,
(d) who have obtained 40 or less marks,
(e) in the range 81 to 100,
(f) in the range 61 to 80,
(g) in the range 41 to 60, and
(h) in the range 0 to 40.

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

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

(a) Marks in Mathematics >= 60
(b) Marks in Physics >= 50
(c) Marks in Chemistry >= 40
(d) Total in all three subjects >= 200

or

Total in Mathematics and Physics >= 150

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

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

Square Root Table

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

6.7 Shown below is a Floyd's triangle.

1
2 3
4 5 6
7 8 9 10

11 15

.

.

79 91

(a) Write a program to print this triangle.

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

1

0 1

1 0 1

0 1 0 1

1 0 1 0 1

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

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

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

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

$$y = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x < 40 \end{cases}$$

using

(a) nested **if** statements,

(b) **else if** statements, and

(c) conditional operator ? :

6.10 Write a program to compute the real roots of a quadratic equation

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

The roots are given by the equations

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

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

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

(a) No solution, if both a and b are zero

(b) There is only one root, if a = 0 (x = –c/b)

(c) There are no real roots, if $b^2 - 4ac$ is negative
(d) Otherwise, there are two real roots

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

6.11 Write a program to read three integer values from the keyboard and displays the output stating that they are the sides of right-angled triangle.

6.12 An electricity board charges the following rates for the use of electricity:
For the first 200 units: 80 P per unit
For the next 100 units: 90 P per unit
Beyond 300 units: Rs 1.00 per unit
All users are charged a minimum of Rs. 100 as meter charge. If the total amount is more than Rs. 400, then an additional surcharge of 15% of total amount is charged.
Write a program to read the names of users and number of units consumed and print out the charges with names.

6.13 Write a program to compute and display the sum of all integers that are divisible by 6 but not divisible by 4 and lie between 0 and 100. The program should also count and display the number of such values.

6.14 Write an interactive program that could read a positive integer number and decide whether the number is a prime number and display the output accordingly.
Modify the program to count all the prime numbers that lie between 100 and 200.

Note *A prime number is a positive integer that is divisible only by 1 or by itself.*

6.15 Write a program to read a double-type value x that represents angle in radians and a character-type variable T that represents the type of trigonometric function and display the value of
(a) sin(x), if s or S is assigned to T,
(b) cos (x), if c or C is assigned to T, and
(c) tan (x), if t or T is assigned to T
using (i) **if.....else** statement and (ii) **switch** statement.

Debugging Exercises

6.1 Identify the error in the following program, if any:

```
#include <stdio.h>
#include <conio.h>
void main()
{
 int g=4;
 clrscr();
 if(g==5)

 getch();
}
```

6.2 Identify the error in the following program, if any:

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

void main()
{
 int year;
 clrscr();

 printf("\nEnter the year value: ");
 scanf("%d",&year);/*Reading the year value*/

 if(year%4==0);/*Determining whether year is a multiple of 4*/
  printf("\n\n%d is a leap year",year);
 else
  printf("\n%d is not a leap year",year);

 getch();
}
```

6.3 Identify the error in the following program, if any:

```
#include <stdio.h>

void main()
{
  long int num1,num2;
  int temp,d1=0,d2=0,d3=0,d4=0,d5=0,d6=0,d7=0,d8=0,d9=0,d0=0;
  printf("\nEnter the number:");
  scanf("%d",&num1);

  num2=num1;
  while(num1!=0)
  {
    temp=num1%10;
         switch(temp)  /*Using  the  switch-case  decision  making
construct*/
    {
        default:
           ; /*Do-nothing*/
        case 1:
        d1++; /*Counting number of Ones*/
        break;
        case 2:
        d2++; /*Counting number of Twos*/
        break;
```

```
            case 3:
            d3++; /*Counting number of Threes*/
            break;
            case 4:
            d4++; /*Counting number of Fours*/
            break;
            case 5:
            d5++; /*Counting number of Fives*/
            break;
            case 0:
            d0++; /*Counting number of Zeros*/
            break;
            case 6:
            d6++; /*Counting number of Sixes*/
            break;
            case 7:
            d7++; /*Counting number of Sevens*/
            break;
            case 8:
            d8++; /*Counting number of Eights*/
            break;
            case 9:
            d9++; /*Counting number of Nines*/
            break;

          }
          num1=num1/10;
        }
        /*Displaying the frequency of individual digits in a number*/
        printf("\nThe no of 0s in %ld are %d",num2,d0);
        printf("\nThe no of 1s in %ld are %d",num2,d1);
        printf("\nThe no of 2s in %ld are %d",num2,d2);
        printf("\nThe no of 3s in %ld are %d",num2,d3);
        printf("\nThe no of 4s in %ld are %d",num2,d4);
        printf("\nThe no of 5s in %ld are %d",num2,d5);
        printf("\nThe no of 6s in %ld are %d",num2,d6);
        printf("\nThe no of 7s in %ld are %d",num2,d7);
        printf("\nThe no of 8s in %ld are %d",num2,d8);
        printf("\nThe no of 9s in %ld are %d",num2,d9);

    }
```

6.4 Identify the error in the following program, if any:

```
#include <stdio.h>

void main()
```

```
{
 int g=4;
 if(g==5)
  goto label;

 getch();
}

label:
/*do nothing*/
```

6.5 Identify the error in the following program, if any:

```
include <stdio.h>

void main()
{
 int g=4;
 int h=7;
 int k;

 k
 =
 g
 >
 h
 ?
 1
 :
 2
 ;
printf("%d",k);

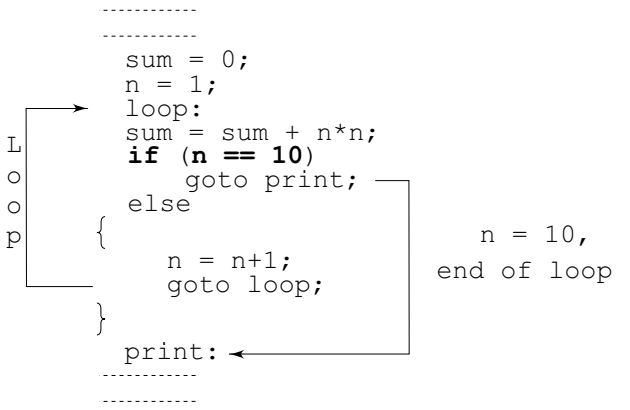
}
```



7 Control Structures: Decision Making and Looping

7.1 Introduction

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



This program does the following things:

1. Initializes the variable **n**.
2. Computes the square of **n** and adds it to **sum**.

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

The program evaluates the statement

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

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

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

Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled loop* or as the *exit-controlled loop.* The flow charts in Fig. 7.1 illustrate these structures. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. The entry-controlled and exit-controlled loops are also known as *pre-test* and *post-test* loops respectively.

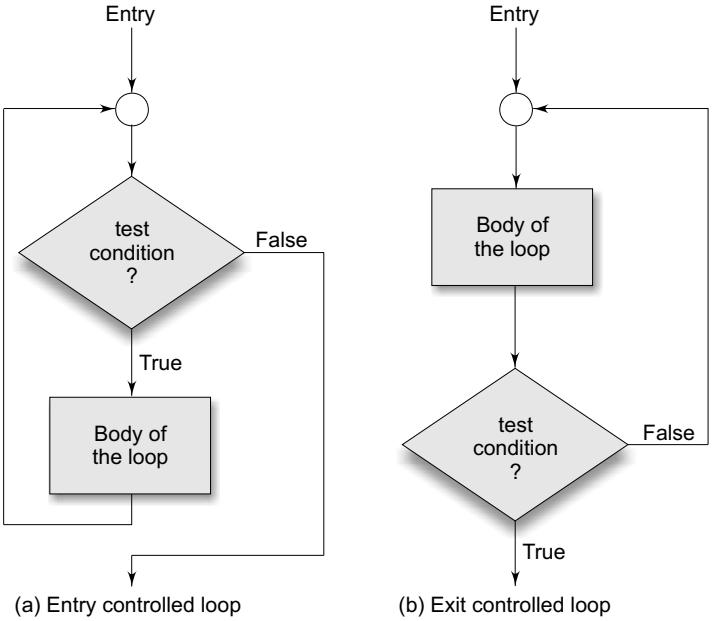


Fig. 7.1 *Loop control structures*

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an *infinite loop* and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides for three *constructs* for performing *loop* operations. They are:

1. The **while** statement.
2. The **do** statement.
3. The **for** statement.

We shall discuss the features and applications of each of these statements in this chapter.

Sentinel Loops

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

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

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

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

7.2 The While Statement

The simplest of all the looping structures in C is the **while** statement. We have used **while** in many of our earlier programs. The basic format of the **while** statement is

```
while (test condition)
{
    body of the loop
}
```

The **while** is an *entry-controlled* loop statement. The *test-condition* is evaluated and if the condition is *true*, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes *false* and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

We can rewrite the program loop discussed in Section 7.1 as follows:

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

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

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

```
        =========
        character = ' ' ;
        while (character != 'Y')
              character = getchar();
        xxxxxxx;
        =========
```

First the **character** is initialized to ' '. The **while** statement then begins by testing whether **character** is not equal to Y. Since the **character** was initialized to ' ', the test is true and the loop statement

```
character = getchar();
```

is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed. When Y is pressed, the condition becomes false because **character** equals Y, and the loop terminates, thus transferring the control to the statement xxxxxxx;. This is a typical example of sentinel-controlled loops. The character constant 'y' is called *sentinel* value and the variable **character** is the condition variable, which often referred to as the *sentinel variable.*

Example 7.1 A program to evaluate the equation

$y = x^n$

when n is a non-negative integer, is given in Fig. 7.2.

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

```
Program
  main()
  {
     int count, n;
     float x, y;
     printf("Enter the values of x and n : ");
     scanf("%f %d", &x, &n);
     y = 1.0;
     count = 1;                /* Initialisation */
     /* LOOP BEGINs */
     while ( count <= n)       /* Testing */
     {
        y = y*x;
        count++;               /* Incrementing */
     }
     /* END OF LOOP */
     printf("\nx = %f; n = %d; x to power n = %f\n",x,n,y);
  }
Output
  Enter the values of x and n : 2.5 4
  x = 2.500000; n = 4; x to power n = 39.062500
  Enter the values of x and n : 0.5 4
  x = 0.500000; n = 4; x to power n = 0.062500
```

Fig. 7.2 *Program to compute **x** to the power n using **while** loop*

7.3 The Do Statement

The **while** loop construct that we have discussed in the previous section, makes a test of condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement. This takes the form:

```
do
{
   body of the loop
```

Contd.

```
        }
        while (test-condition);
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test-condition* in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the *test-condition* is evaluated at the bottom of the loop, the **do...while** construct provides an *exit-controlled* loop and therefore the body of the loop is *always executed at least once*.

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

```
                    do
                    {
                         printf ("Input a number\n");
loop                     number = getnum ( );
                    }
                    while (number > 0);
```

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

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

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

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

Consider another example:

```
    {
        ----------
        I = 1;                              /* Initializing */
        sum = 0;

        do
        {
        sum = sum + I;
loop    I = I+2;                            /* Incrementing */
        }
        while(sum < 40 || I < 10);          /* Testing */
        printf("%d %d\n", I, sum);
        ----------
```

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

Example 7.2 A program to print the multiplication table from 1 × 1 to 12 × 10 as shown below is given in Fig. 7.3.

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

This program contains two **do.... while** loops in nested form. The outer loop is controlled by the variable row and executed 12 times. The inner loop is controlled by the variable column and is executed 10 times, each time the outer loop is executed. That is, the inner loop is executed a total of 120 times, each time printing a value in the table.

Program:

```
#define COLMAX 10
#define ROWMAX 12
main()
{
      int row,column, y;
      row = 1;
      printf(" MULTIPLICATION TABLE \n");
      printf("-----------------------\n");
      do /*......OUTER LOOP BEGINS........*/
      {
            column = 1;
            do /*.......INNER LOOP BEGINS.......*/
            {
               y = row * column;
               printf("%4d", y);
               column = column + 1;
            }
      while (column <= COLMAX); /*...INNER LOOP ENDS...*/
      printf("\n");
      row = row + 1;
      }
   while (row <= ROWMAX);/*..... OUTER LOOP ENDS .....*/
   printf("---------------------------------\n");
}
```

Output

```
              MULTIPLICATION TABLE
_     _     _     _     _     _     _     _     _     _     _     _
1     2     3     4     5     6     7     8     9     10
2     4     6     8     10    12    14    16    18    20
3     6     9     12    15    18    21    24    27    30
4     8     12    16    20    24    28    32    36    40
```

Contd.

```
5     10    15    20    25    30    35    40    45    50
6     12    18    24    30    36    42    48    54    60
7     14    21    28    35    42    49    56    63    70
8     16    24    32    40    48    56    64    72    80
9     18    27    36    45    54    63    72    81    90
10    20    30    40    50    60    70    80    90    100
11    22    33    44    55    66    77    88    99    110
12    24    36    48    60    72    84    96    108   120
```

Fig. 7.3 *Printing of a multiplication table using* ***do...while loop***

Notice that the **printf** of the inner loop does not contain any new line character (\n). This allows the printing of all row values in one line. The empty **printf** in the outer loop initiates a new line to print the next row.

7.4 The for Statement

7.4.1 Simple 'for' Loops

The **for** loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the **for** loop is

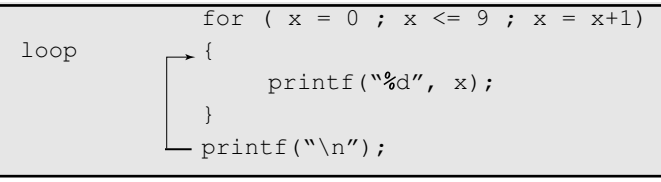
```
for ( initialization ; test-condition ; increment)
{
      body of the loop
}
```

The execution of the **for** statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as **i** = 1 and count = 0. The variables **i** and **count** are known as loop-control variables.
2. The value of the control variable is tested using the test-condition. The *test-condition* is a relational expression, such as i < 10 that determines when the loop will exit. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Note *C99 enhances the* ***for*** *loop by allowing declaration of variables in the initialization portion.*

Consider the following segment of a program:



This **for** loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the *increment* section, x = x+1.

The **for** statement allows for *negative increments*. For example, the loop discussed above can be written as follows:

```
for ( x = 9 ; x >= 0 ; x = x-1 )
     printf("%d", x);
printf("\n");
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

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

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

Let us again consider the problem of sum of squares of integers discussed in Section 7.1. This problem can be coded using the **for** statement as follows:

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

The body of the loop

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

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

One of the important points about the **for** loop is that all the three actions, namely *initialization, testing,* and *incrementing*, are placed in the **for** statement itself, thus making them visible to the programmers and users, in one place. The **for** statement and its equivalent of **while** and **do** statements are shown in Table 7.1.

Table 7.1 ***Comparison of the Three Loops***

for	*while*	*do*

for (n=1; n<=10; ++n)
{
_____
_____
}
	n = 1;
	while (n<=10)
	{
	_____
	_____
	n = n+1;
	}
		n = 1;
		do
		{
		_____
		_____
		n = n+1;
		}
		while(n<=10);

Example 7.3 The program in Fig. 7.4 uses a for loop to print the "Powers of 2" table for the power 0 to 20, both positive and negative.

The program evaluates the value

$$p = 2^n$$

successively by multiplying 2 by itself n times.

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

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

7.4.2 Additional Features of for Loop

The **for** loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the **for** statement. The statements

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

can be rewritten as

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

Program

```
main()
{
   long int p;
   int n;
   double q;
   printf("---------------------------------\n");
   printf(" 2 to power n       n       2 to power -n\n");
   printf("---------------------------------\n");
   p = 1;
   for (n = 0; n < 21 ; ++n) /* LOOP BEGINS */
   {
      if (n == 0)
         p = 1;
      else
         p = p * 2;
```

Contd.

```
        q = 1.0/(double)p ;
        printf("%10ld %10d %20.12lf\n", p, n, q);
    }                                   /* LOOP ENDS */
    printf("-----------------------------------\n");
  }
```

Output

```
        ----------------------------------------
           2 to power n    n         2 to power -n
        ----------------------------------------
                 1             0       1.000000000000
                 2             1       0.500000000000
                 4             2       0.250000000000
                 8             3       0.125000000000
                16             4       0.062500000000
                32             5       0.031250000000
                64             6       0.015625000000
               128             7       0.007812500000
               256             8       0.003906250000
               512             9       0.001953125000
              1024            10       0.000976562500
              2048            11       0.000488281250
              4096            12       0.000244140625
              8192            13       0.000122070313
             16384            14       0.000061035156
             32768            15       0.000030517578
             65536            16       0.000015258789
            131072            17       0.000007629395
            262144            18       0.000003814697
            524288            19       0.000001907349
           1048576            20       0.000000953674
        ----------------------------------------
```

Fig. 7.4 *Program to print '**Power of 2**' table using for loop*

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

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

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

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

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

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

The loop uses a compound test condition with the counter variable **i** and sentinel variable **sum**. The loop is executed as long as both the conditions i < 20 and **sum** < 100 are true. The **sum** is evaluated inside the loop.

It is also permissible to use expressions in the assignment statements of initialization and increment sections. For example, a statement of the type

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

is perfectly valid.

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

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

Both the initialization and increment sections are omitted in the **for** statement. The initialization has been done before the **for** statement and the control variable is incremented inside the loop. In such cases, the sections are left 'blank'. However, the semicolons separating the sections must remain. If the test-condition is not present, the **for** statement sets up an *'infinite'* loop. Such loops can be broken using **break** or **goto** statements in the loop.

We can set up *time delay loops* using the null statement as follows:

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

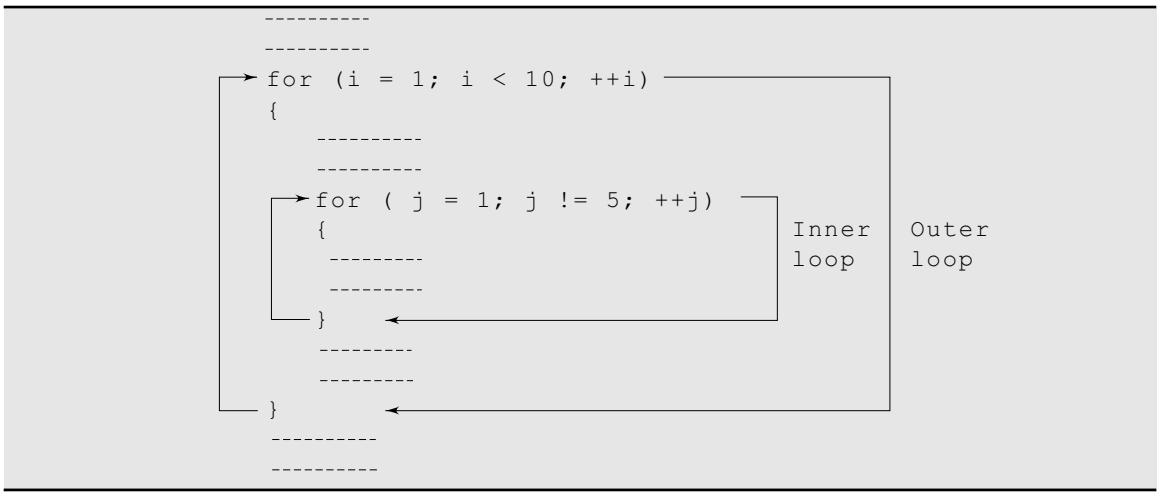
This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *null* statement. This can also be written as

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

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

7.4.3 Nesting of for Loops

Nesting of loops, that is, one **for** statement within another **for** statement, is allowed in C. For example, two loops can be nested as follows:



The nesting may continue up to any desired level. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each **for** statement. (ANSI C allows up to 15 levels of nesting. However, some compilers permit more.)

The program to print the multiplication table discussed in Example 7.2 can be written more concisely using nested for statements as follows:

```
---------------
      for (row = 1; row <= ROWMAX ; ++row)
      }
         for (column = 1; column <= COLMAX ; ++column)
         }
               y = row * column;
               printf("%4d", y);
         }
         printf("( *** F A I L *** )\n\n");
   }
   -----------------
```

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

Example 7.4 A class of **n** students take an annual examination in **m** subjects. A program to read the marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them is given in Fig. 7.5.

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

The outer loop includes three parts:

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

Program

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

Output

```
Enter number of students and subjects
3  6
Enter roll_number : 8701
Enter marks of 6 subjects for ROLL NO 8701
81 75 83 45 61 59
TOTAL MARKS = 404 ( First Division )
Enter roll_number : 8702
Enter marks of 6 subjects for ROLL NO 8702
51 49 55 47 65 41
TOTAL MARKS = 308 ( Second Division )
Enter roll_number : 8704
Enter marks of 6 subjects for ROLL NO 8704
40 19 31 47 39 25
TOTAL MARKS = 201 ( *** F A I L *** )
```

Fig. 7.5 *Illustration of* ***nested for loops***

Selecting a Loop

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

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

7.4.4 Programs on Pattern Generation

Example 7.5 Write a program that reads a value n and generates the following pattern:

```
1
12
123
1234
12345
1234
123
12
1
```

Here n = 5.

Program

```
#include <stdio.h>
#include <conio.h>
void main ()
{
   int n,i,j,k;
   clrscr();
   printf("Enter the value of n: ");
   scanf("%d",&n);

   for(i=1;i<=(2*n-1);i++)
   {
      if(i<n)
       j=i;
      else
       j=n-(i%n);

      for(k=1;k<=j;k++)
       printf("%d",k);
```

Contd.

```
      printf("\n");
    }
    getch();
  }
Output
Enter the value of n: 10
1
12
123
1234
12345
123456
1234567
12345678
123456789
12345678910
123456789
12345678
1234567
123456
12345
1234
123
12
1
```

Fig. 7.6 *Program to read a value and generate a pattern*

Example 7.6 Write a program that reads a value n and generates the following pattern:

55555
4444
333
22
1
22
333
4444
55555

Here n = 5.

Program

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

void main ()
{
```

Contd.

```
   int n,i,j,k;
   clrscr();

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

   printf("\n");
   for(i=1;i<=(2*n-1);i++)
 {
  if(i<n)
   j=n-i+1;
  else
   j=(i%n)+1;

  for(k=1;k<=j;k++)
   printf("%d",j);

  printf("\n");
 }
 getch();
}
```

Output

```
Enter the value of n: 8

88888888
7777777
666666
55555
4444
333
22
1
22
333
4444
55555
666666
7777777
88888888
```

Fig. 7.7 *Program to read a value and generate a pattern*

Example 7.7 Write a program that reads a value n and generates the following pattern:

```
      1
    1 2 3
  1 2 3 4 5
```

1 2 3 4 5 6 7
1 2 3 4 5 6 7 8 9
Here n = 5.

Program

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

void main ()
{
 int n,i,j;
 clrscr();

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

 printf(“\n”);
 for(i=1;i<=n;i++)
 {
  for(j=0;j<=(n-i);j++)
   printf(“  “);

  for(j=1;j<=(2*i-1);j++)
   printf( “%d “,j);

  printf(“\n”);
 }
 getch();
}
```

Output

```
Enter the value of n: 5
          1
        1 2 3
      1 2 3 4 5
    1 2 3 4 5 6 7
  1 2 3 4 5 6 7 8 9
```

Fig. 7.8 *Program to read a value and generate a pattern*

Example 7.8 Write a program that reads a value n and generates the following pattern:

1
121
12321

```
   121
    1
```

Here n = 5.

Program

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

void main ()
{
 int n,i,j,k;
 clrscr();

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

 printf(“\n”);

 for(i=1;i<=n;i++)
 {
  if(i<(n/2)+1)
   j=i;
  else
   j=n-i+1;


  for(k=0;k<=n-j;k++)
   printf(“ “);

  for(k=1;k<j;k++)
   printf(“%d”,k);

  for(k=j;k>=1;k--)
   printf(“%d”,k);

  printf(“\n”);
 }

 getch();
}
```

Output

```
Enter the value of n: 9

         1
        121
       12321
      1234321
     123454321
      1234321
```

Contd.

```
        12321
         121
          1

Enter the value of n: 8

          1
         121
        12321
       1234321
       1234321
        12321
         121
          1
```

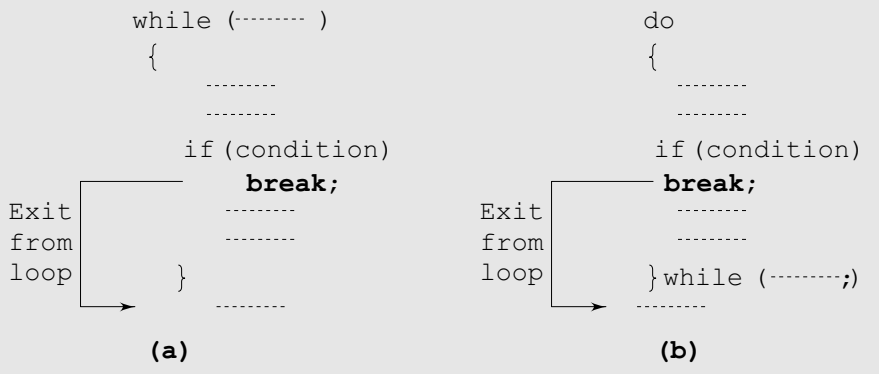
Fig. 7.9 *Program to read a value and generate a pattern*

7.5 Jumps in Loops

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing, say, 100 names. A program loop written for reading and testing the names 100 times must be terminated as soon as the desired name is found. C permits a *jump* from one statement to another within a loop as well as a *jump* out of a loop.

7.5.1 Jumping Out of a Loop

An early exit from a loop can be accomplished by using the break statement or the goto statement. We have already seen the use of the **break** in the **switch** statement and the **goto** in the **if...else** construct. These statements can also be used within **while, do**, or **for** loops. They are illustrated in Fig. 7.10 and Fig. 7.11.



Contd.

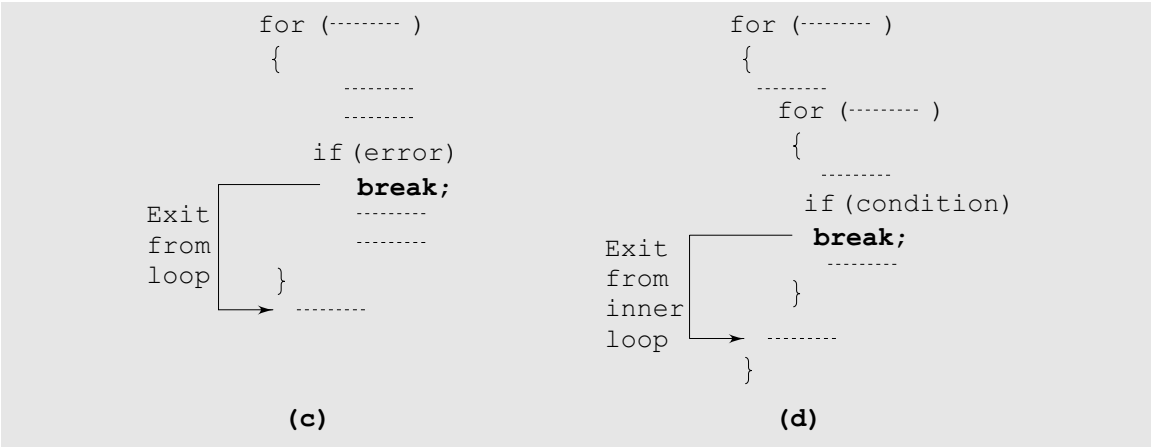


Fig. 7.10 *Exiting a loop with **break** statement*

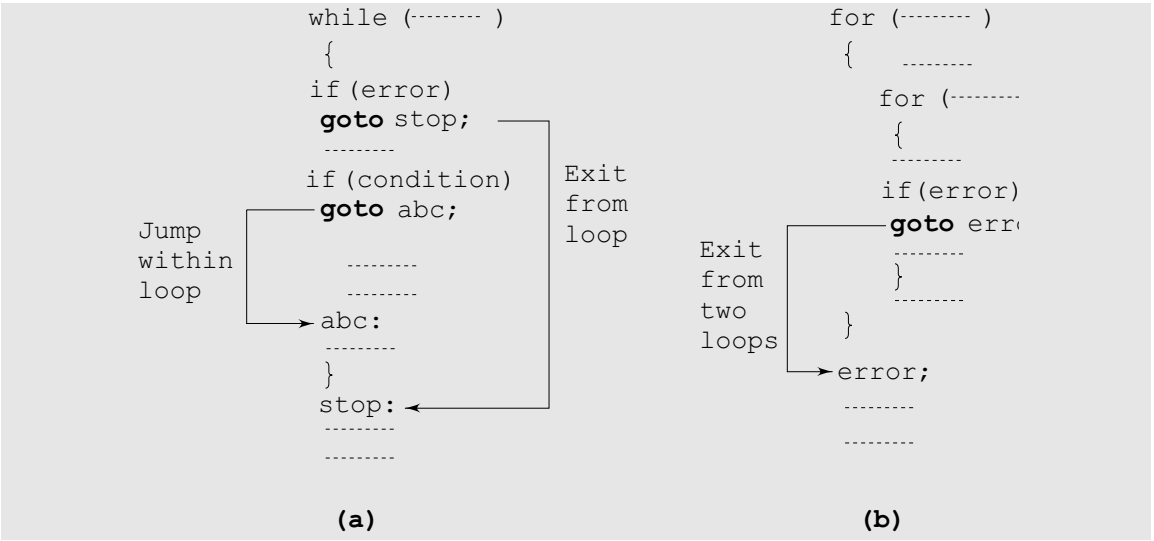


Fig. 7.11 *Jumping within and exiting from the loops with **goto** statement*

When a break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** will exit only a single loop.

Since a **goto** statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of **goto** is to exit from deeply nested loops when an error occurs. A simple **break** statement would not work here.

Example 7.9 The program in Fig. 7.12 illustrates the use of the break statement in a C program.

The program reads a list of positive values and calculates their average. The **for** loop is written to read 1000 values. However, if we want the program to calculate the average of any set of values less than 1000, then we must enter a 'negative' number after the last value in the list, to mark the end of input.

```
Program
  main()
  {
     int m;
     float x, sum, average;
     printf("This program computes the average of a
                        set of numbers\n");
     printf("Enter values one after another\n");
     printf("Enter a NEGATIVE number at the end.\n\n");
     sum = 0;
     for (m = 1 ; m < = 1000 ; ++m)
     {
           scanf("%f", &x);
           if (x < 0)
              break;
           sum += x ;
     }
     average = sum/(float)(m-1);
     printf("\n");
     printf("Number of values = %d\n", m-1);
     printf("Sum              = %f\n", sum);
     printf("Average          = %f\n", average);
  }
Output
  This program computes the average of a set of numbers
  Enter values one after another
  Enter a NEGATIVE number at the end.
  21 23 24 22 26 22 -1
  Number of values   = 6
  Sum                = 138.000000
  Average            = 23.000000
```

Fig. 7.12 *Use **of break** in a program*

Each value, when it is read, is tested to see whether it is a positive number or not. If it is positive, the value is added to the **sum**; otherwise, the loop terminates. On exit, the average of the values read is calculated and the results are printed out.

Example 7.10 A program to evaluate the series

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots\cdots + x^n$$

for $-1 < x < 1$ with 0.01 per cent accuracy is given in Fig. 7.13. The **goto** statement is used to exit the loop on achieving the desired accuracy.

We have used the **for** statement to perform the repeated addition of each of the terms in the series. Since it is an infinite series, the evaluation of the function is terminated when the term x^n reaches the desired accuracy. The value of n that decides the number of loop operations is not known and therefore we have decided arbitrarily a value of 100, which may or may not result in the desired level of accuracy.

Program

```
#define    LOOP            100
#define    ACCURACY        0.0001
main()
{
      int n;
      float x, term, sum;
      printf("Input value of x : ");
      scanf("%f", &x);
      sum = 0 ;
      for (term = 1, n = 1 ; n <= LOOP ; ++n)
      {
            sum += term ;
            if (term <= ACCURACY)
            goto output; /* EXIT FROM THE LOOP */
         term *= x ;
      }
      printf("\nFINAL VALUE OF N IS NOT SUFFICIENT\n");
      printf("TO ACHIEVE DESIRED ACCURACY\n");
      goto end;
      output:
      printf("\nEXIT FROM LOOP\n");
      printf("Sum = %f; No.of terms = %d\n", sum, n);
      end:
      ;         /* Null Statement */
}
```

Output

```
Input value of x : .21
EXIT FROM LOOP
Sum = 1.265800; No.of terms = 7
Input value of x : .75
EXIT FROM LOOP
Sum = 3.999774; No.of terms = 34
Input value of x : .99
FINAL VALUE OF N IS NOT SUFFICIENT
TO ACHIEVE DESIRED ACCURACY
```

Fig. 7.13 *Use of **goto** to exit from a loop*

The test of accuracy is made using an **if** statement and the **goto** statement exits the loop as soon as the accuracy condition is satisfied. If the number of loop repetitions is not large enough to produce the desired accuracy, the program prints an appropriate message.

Note that the **break** statement is not very convenient to use here. Both the normal exit and the **break** exit will transfer the control to the same statement that appears next to the loop. But, in the present problem, the normal exit prints the message

"FINAL VALUE OF N IS NOT SUFFICIENT
TO ACHIEVE DESIRED ACCURACY"

and the *forced exit* prints the results of evaluation. Notice the use of a *null* statement at the end. This is necessary because a program should not end with a label.

Structured Programming

Structured programming is an approach to the design and development of programs. It is a discipline of making a program's logic easy to understand by using only the basic three control structures:

- Sequence (straight line) structure
- Selection (branching) structure
- Repetition (looping) structure

While sequence and loop structures are sufficient to meet all the requirements of programming, the selection structure proves to be more convenient in some situations.

The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured.

Structured programming discourages the implementation of unconditional branching using jump statements such as **goto**, **break** and **continue**. In its purest form, structured programming is synonymous with "*goto less programming*".

Do not go to **goto** statement!

7.5.2 Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the **break** statement, C supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The **continue** statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the **continue** statement is simply

```
continue;
```

The use of the **continue** statement in loops is illustrated in Fig. 7.14. In **while** and **do** loops, **continue** causes the control to go directly to the test-condition and then to continue the iteration process. In the case of **for** loop, the increment section of the loop is executed before the test-condition is evaluated.

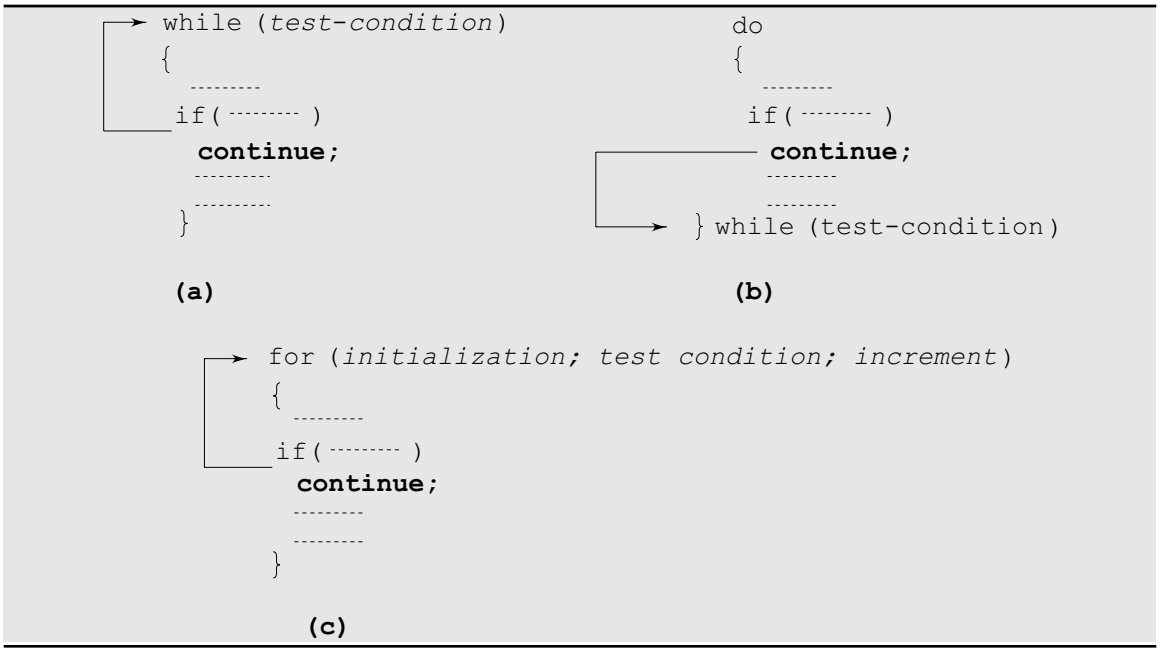


Fig. 7.14 *Bypassing and continuing in loops*

Example 7.11 The program in Fig. 7.15 illustrates the use of **continue** statement.

The program evaluates the square root of a series of numbers and prints the results. The process stops when the number 9999 is typed in.

In case, the series contains any negative numbers, the process of evaluation of square root should be bypassed for such numbers because the square root of a negative number is not defined. The **continue** statement is used to achieve this. The program also prints a message saying that the number is negative and keeps an account of negative numbers.

The final output includes the number of positive values evaluated and the number of negative items encountered.

Program:

```
#include <math.h>
main()
{
   int count, negative;
   double number, sqroot;
   printf("Enter 9999 to STOP\n");
   count = 0 ;
   negative = 0 ;
   while (count < = 100)
   {
      printf("Enter a number : ");
```

Contd.

```
            scanf("%lf", &number);
            if (number == 9999)
               break;          /* EXIT FROM THE LOOP */
            if (number < 0)
            {
               printf("Number is negative\n\n");
               negative++ ;
               continue; /* SKIP REST OF THE LOOP */
            }
            sqroot = sqrt(number);
            printf("Number     = %lf\n Square root = %lf\n\n",
                                 number, sqroot);
            count++ ;
         }
         printf("Number of items done = %d\n", count);
         printf("\n\nNegative items = %d\n", negative);
         printf("END OF DATA\n");
      }
Output
      Enter 9999 to STOP
      Enter a number : 25.0
      Number          = 25.000000
      Square root     = 5.000000
      Enter a number : 40.5
      Number          = 40.500000
      Square root     = 6.363961
      Enter a number : -9
      Number is negative
      Enter a number : 16
      Number          = 16.000000
      Square root     = 4.000000
      Enter a number : -14.75
      Number is negative
      Enter a number : 80
      Number          = 80.000000
      Square root     = 8.944272
      Enter a number : 9999
      Number of items done = 4
      Negative items         = 2
      END OF DATA
```

Fig. 7.15 *Use of* ***continue*** *statement*

7.5.3 Avoiding goto

As mentioned earlier, it is a good practice to avoid using **goto**. There are many reasons for this. When **goto** is used, many compilers generate a less efficient code. In addition, using many of them makes a

program logic complicated and renders the program unreadable. It is possible to avoid using **goto** by careful program design. In case any goto is absolutely necessary, it should be documented. The **goto** jumps shown in Fig. 7.16 would cause problems and therefore must be avoided.

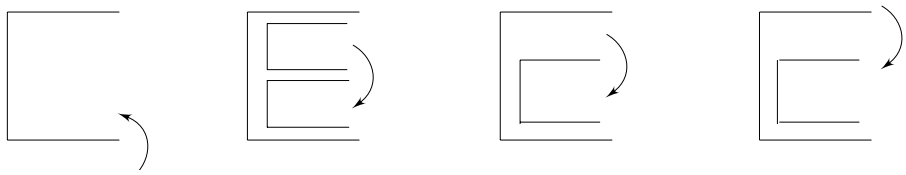


Fig. 7.16 *goto jumps to be avoided*

7.5.4 Jumping out of the Program

We have just seen that we can jump out of a loop using either the **break** statement or **goto** statement. In a similar way, we can jump out of a program by using the library function **exit**( ). In case, due to some reason, we wish to break out of a program and return to the operating system, we can use the **exit**( ) function, as shown below:

```
........
........
if (test-condition) exit(0) ;
........
........
```

The **exit( )** function takes an integer value as its argument. Normally *zero* is used to indicate normal termination and a *nonzero* value to indicate termination due to some error or abnormal condition. The use of **exit( )** function requires the inclusion of the header file **<stdlib.h>**.

7.6 Concise Test Expressions

We often use test expressions in the **if, for, while** and **do** statements that are evaluated and compared with zero for making branching decisions. Since every integer expression has a true/false value, we need not make exlicit comparisons with zero. For instance, the expression x is true whenever x is not zero, and false when x is zero. Applying! operator, we can write concise test expressions without using any relational operators.

if (expression == 0)

is equivalent to

if (!expression)

Similarly,

if (expression! = 0)

is equivalent to

if (expression)

For example,

if (m%5 == 0 && n%5 == 0) is same as **if (!(m%5)&&!(n%5))**

Just Remember

- Do not forget to place the semicolon at the end of **do ….while** statement.
- Placing a semicolon after the control expression in a **while** or **for** statement is not a syntax error but it is most likely a logic error.
- Using commas rather than semicolon in the header of a **for** statement is an error.
- Do not forget to place the **increment** statement in the body of a **while** or **do…while** loop.
- It is a common error to use wrong relational operator in test expressions. Ensure that the loop is evaluated exactly the required number of times.
- Avoid a common error using = in place of = = operator.
- Do not change the control variable in both the **for** statement and the body of the loop. It is a logic error.
- Do not compare floating-point values for equality.
- Avoid using **while** and **for** statements for implementing exit-controlled (post-test) loops. Use **do…while** statement. Similarly, do not use **do…while** for pre-test loops.
- When performing an operation on a variable repeatedly in the body of a loop, make sure that the variable is initialized properly before entering the loop.
- Although it is legally allowed to place the initialization, testing and increment sections outside the header of a **for** statement, avoid them as far as possible.
- Although it is permissible to use arithmetic expressions in initialization and increment section, be aware of round off and truncation errors during their evaluation.
- Although statements preceding a for and statements in the body can be placed in the **for** header, avoid doing so as it makes the program more difficult to read.
- The use of **break** and **continue** statements in any of the loops is considered unstructured programming. Try to eliminate the use of these jump statements, as far as possible.
- Avoid the use of **goto** anywhere in the program.
- Indent the statements in the body of loops properly to enhance readability and understandability.
- Use of blank spaces before and after the loops and terminating remarks are highly recommended.
- Use the function **exit()** only when breaking out of a program is necessary.

Review Questions

7.1 State whether the following statements are *true* or *false*.

(a) The **do…while** statement first executes the loop body and then evaluate the loop control expression.

(b) In a pretest loop, if the body is executed **n** times, the test expression is executed **n** + 1 times.

(c) The number of times a control variable is updated always equals the number of loop iterations.

(d) Both the pretest loops include initialization within the statement.

(e) In a **for** loop expression, the starting value of the control variable must be less than its ending value.
(f) The initialization, test condition and increment parts may be missing in a **for** statement.
(g) **While** loops can be used to replace **for** loops without any change in the body of the loop.
(h) An exit-controlled loop is executed a minimum of one time.
(i) The use of **continue** statement is considered as unstructured programming.
(j) The three loop expressions used in a **for** loop header must be separated by commas.

7.2 Fill in the blanks in the following statements.
(a) In an exit-controlled loop, if the body is executed n times, the test condition is evaluated _____ times.
(b) The _____ statement is used to skip a part of the statements in a loop.
(c) A **for** loop with the no test condition is known as _____ loop.
(d) The sentinel-controlled loop is also known as _____ loop.
(e) In a counter-controlled loop, variable known as _____ is used to count the loop operations.

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

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

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

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

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

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

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

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

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

(d)
```
for ( n = 1; ; n = n + 1)
sum = sum + n;
```

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

7.9 What would be the output of each of the following code segments?

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

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

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

(d)
```
for (m = 10; m > 7, m -=2)
    printf(m);
```

7.10 Compare, in terms of their functions, the following pairs of statements:

(a) while and do...while
(b) while and for
(c) break and goto
(d) break and continue
(e) continue and goto

7.11 Analyse each of the program segments that follow and determine how many times the body of each loop will be executed.

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

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

(c)
```
int i;
for (i = 0; i <= 5; i = i+2/3)
{
     ---
     ---
     ---
}
```

(d)
```
int m = 10;
int n = 7;
while ( m % n >= 0)
{
     ---
     m = m + 1;
     n = n + 2;
     ---
}
```

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

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

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

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

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

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

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

7.13 Write a **for** statement to print each of the following sequences of integers:

(a) 1, 2, 4, 8, 16, 32

(b) 1, 3, 9, 27, 81, 243

(c) – 4, –2, 0, 2, 4

(d) –10, –12, –14, –18, –26, – 42

7.14 Change the following **for** loops to **while** loops:

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

(b)
```
for ( ; scanf("%d", & m) != -1;)
printf(m);
```

7.15 Change the **for** loops in Exercise 7.14 to **do** loops.

7.16 What is the output of following code?

```
int m = 100, n = 0;
while ( n == 0 )
{
        if ( m < 10 )
                  break;
        m = m-10;
```

7.17 What is the output of the following code?

```
int m = 0 ;
do
{
       if (m > 10 )
             continue ;
       m = m + 10 ;
} while ( m < 50 ) ;
printf("%d", m);
```

7.18 What is the output of the following code?

```
int n = 0, m = 1 ;
do
{
       printf(m) ;
       m++ ;
}
while (m <= n) ;
```

7.19 What is the output of the following code?

```
int n = 0, m ;
for (m = 1; m <= n + 1 ; m++ )
   printf(m);
```

7.20 When do we use the following statement?

```
for (; ; )
```

Programming Exercises

7.1 Given a number, write a program using **while** loop to reverse the digits of the number. For example, the number

12345

should be written as

54321

(**Hint:** Use modulus operator to extract the last digit and the integer division by 10 to get the n–1 digit number from the n digit number.)

7.2 The factorial of an integer m is the product of consecutive integers from 1 to m. That is,

factorial m = m! = m x (m–1) x ..... x 1.

Write a program that computes and prints a table of factorials for any given m.

7.3 Write a program to compute the sum of the digits of a given integer number.

7.4 The numbers in the sequence

1 1 2 3 5 8 13 21 .....

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

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

7.5 Rewrite the program of the Example 7.1 using the **for** statement.

7.6 Write a program to evaluate the following investment equation

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

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

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

r : 0.10, 0.11, 0.12, ..., 0.20

n : 1, 2, 3, ..., 10

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

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

$$P = V$$

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

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

(a)
```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

(b)
```
* * * * *
 * * * *
  * * *
   * *
    *
```

7.8 Write a program to read the age of 100 persons and count the number of persons in the age group 50 to 60. Use **for** and **continue** statements.

7.9 Rewrite the program of case study 7.4 (plotting of two curves) using **else...if** constructs instead of **continue** statements.

7.10 Write a program to print a table of values of the function

$$y = \exp(-x)$$

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

Table for Y = EXP(−X)

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

Contd.

x	0.1	0.2	0.3	.....	0.9
.					
.					
9.0					

7.11 Write a program that will read a positive integer and determine and print its binary equivalent. (**Hint**: The bits of the binary representation of an integer can be generated by repeatedly dividing the number and the successive quotients by 2 and saving the remainder, which is either 0 or 1, after each division.)

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

```
* * * * * * * * * * * * * * * * * *
* * - - - - - - - - - - - - - - - - - - * *
* * * * * * * * * - - - - - - - - - - * *
* * * *
* * * *
* * * *
* * * * * - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - * * * *
                                * * * *
                                * * * *
                                * * * *
* * * * - - - - - - - - — - - - * * * *
* * * - - - - - - - - - - - - - - * * * *
* * - - - - - - - - - - - - - - - * * * *
```

7.13 Write a program to compute the value of Euler's number e, that is used as the base of natural logarithms. Use the following formula.

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

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

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

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

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

(c) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \cdots\cdots$

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

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

where c = original cost
d = rate of depreciation (per year)
n = number of years
p = present value after y years.

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

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

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

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

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

7.17 Write a program to graph the function

$$y = \sin(x)$$

in the interval 0 to 180 degrees in steps of 15 degrees. Use the concepts discussed in the Case Study 4 in Chapter 7.

7.18 Write a program to print all integers that are **not divisible** by either 2 or 3 and lie between 1 and 100. Program should also account the number of such integers and print the result.

7.19 Modify the program of Exercise 7.16 to print the character O instead of S at the center of the square as shown below.

```
S S S S S
S S S S S
S S O S S
S S S S S
S S S S S
```

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

Debugging Exercises

7.1 Identify the error in the following program, if any:

```
#include <stdio.h>

void main()
{
 int i,n;
 long sum=0;

 printf("Enter the value of n = ");
 scanf("%d",&n); /*Reading number of terms in the series, n*/

 i=1;
 for(;i<=n;) /*Calculating the sum of the series*/
 {
```

```
  sum=sum+i*i;i=i+1;
 }
 printf("Sum of series = %ld",sum);

}
```

7.2 Identify the error in the following code for reading matrix elements from the console:

```
for(i=0;i<3;i++)
{
  for(j=0;j<3;j++)
   printf("a[%d][%d] = ",i,j);
   scanf("%d",&a[i][j]);
}
```

7.3 Identify the error in the following program for determining whether a given number is Armstrong or not:

```
#include <stdio.h>
void main()
{
 int num, temp, sum=0, i;
 printf("\nEnter a number: ");
 scanf("%d", &num);
 temp=num;
 while(temp>0)
 {
  i=temp%10;
  sum=sum+i*i*i;
  temp=temp/10;
 };
 if(sum==num)
  printf("\n%d is an Armstrong number",num);
 else
 printf("\n%d is not an Armstrong number",num);

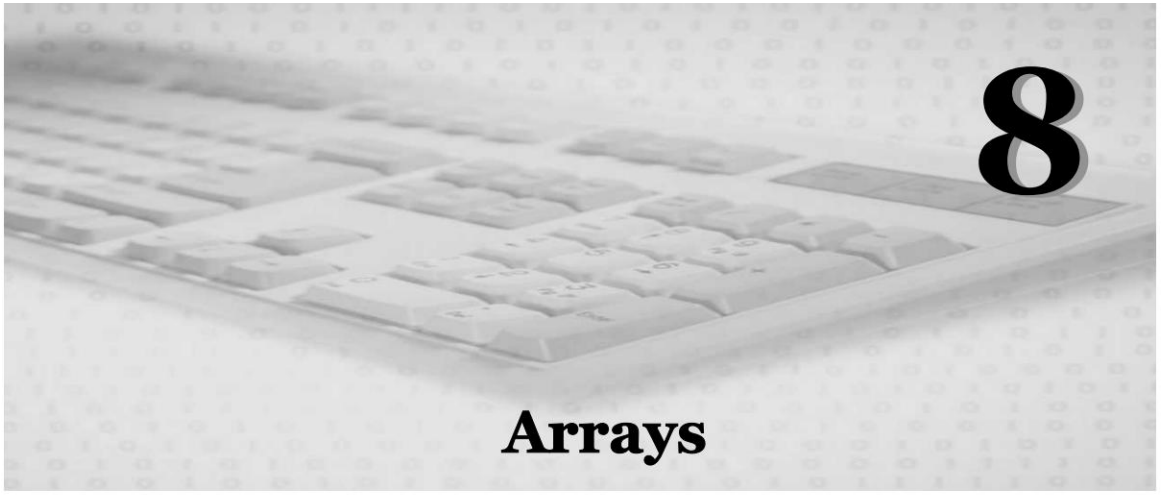
}
```

7.4 Identify the error in the following program, if any:

```
#include <stdio.h>
#include <conio.h>
void main()
{
 int i;
 do
 {
```

```
  printf("\nEnter the value of i = ");
  scanf("%d",&i);
  printf("\nYou entered i = %d ",i);
 }while(i>0)
 printf("\n\nYou just entered a negative value for i..the program
will quit.");

 getch();
}
```



8 Arrays

8.1 Introduction

So far we have used only the fundamental data types, namely **char, int, float, double** and variations of **int** and **double**. Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as *array* that can be used for such applications.

An array is a *fixed-size* sequenced collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept of an array can be used:

- List of temperatures recorded every hour in a day, or a month, or a year.
- List of employees in an organization.
- List of products and their cost sold by a store.
- Test scores of a class of students.
- List of customers and their telephone numbers.
- Table of daily rainfall data.

and so on.

Since an array provides a convenient structure for representing data, it is classified as one of the *data structures* in C. Other data structures include structures, lists, queues and trees. A complete discussion of all data structures is beyond the scope of this text. However, we shall consider structures in Chapter 12.

As we mentioned earlier, an array is a sequenced collection of related data items that share a common name. For instance, we can use an array name *salary* to represent a *set of salaries* of a group of employees in an organization. We can refer to the individual salaries by writing a number called *index* or *subscript* in brackets after the array name. For example,

```
salary [10]
```

represents the salary of 10^{th} employee. While the complete set of values is referred to as an array, individual values are called *elements*.

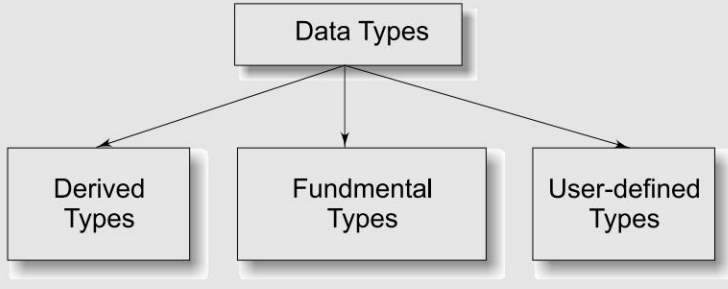
The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array,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

We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions. In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.

- One-dimensional arrays
- Two-dimensional arrays
- Multidimensional arrays

Data Structures

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



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

Arrays and structures are referred to as *structured data types* because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations. In programming parlance, such data types are known as *data structures.*

In addition to arrays and structures, C supports creation and manipulation of the following data structures:

- Linked Lists
- Stacks
- Queues
- Trees

8.2 One-Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted variable* or a *one-dimensional array*. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation

$$A = \frac{\sum_{i=1}^{n} x_i}{n}$$

to calculate the average of n values of x. The subscripted variable x_i refers to the ith element of x. In C, single-subscripted variable x_i can be expressed as

x[1], x[2], x[3],.....x[n]

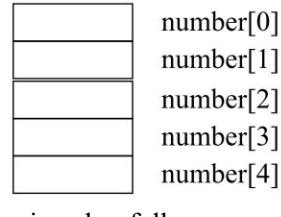
The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35,40,20,57,19), by an array variable **number**, then we may declare the variable **number** as follows

```
int number[5];
```

and the computer reserves five storage locations as shown below:



The values to the array elements can be assigned as follows:

```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

This would cause the array **number** to store the values as shown below:

number[0]	35
number[1]	40
number[2]	20
number[3]	57
number[4]	19

These elements may be used in programs just like any other C variable. For example, the following are valid statements:

```
a = number[0] + 10;
number[4] = number[0] + number [2];
number[2] = x[5] + y[10];
value[6] = number[i] * 3;
```

The subscripts of an array can be integer constants, integer variables like i, or expressions that yield integers. *C performs no bounds checking and, therefore, care should be exercised to ensure that the array indices are within the declared limits.*

8.3 Declaration of One-Dimensional Arrays

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

```
type variable-name[ size ];
```

The *type* specifies the type of element that will be contained in the array, such as **int, float,** or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember:

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a numeric constant or a symbolic constant.

The C language treats character strings simply as arrays of characters. The *size* in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

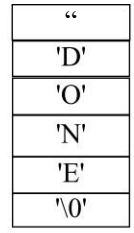
declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name.**

"WELL DONE"

Each character of the string is treated as an element of the array **name** and is stored in the memory as follows:

'W'
'E'
'L'
'L'

Contd.



When the compiler sees a character string, it terminates it with an additional null character. Thus, the element **name[10]** holds the null character ‘\0’. *When declaring character arrays, we must allow one extra element space for the null terminator*.

Example 8.1 Write a program using a single-subscripted variable to evaluate the following expressions:

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

The values of x1,x2,....are read from the terminal.

Program in Fig. 8.1 uses a one-dimensional array **x** to read the values and compute the sum of their squares.

Program

```
main()
   {
      int i ;
      float x[10], value, total ;

/* . . . . . .READING VALUES INTO ARRAY . . . . . . */

      printf(“ENTER 10 REAL NUMBERS\n”) ;

      for( i = 0 ; i < 10 ; i++ )
      {
         scanf(“%f”, &value) ;
         x[i] = value ;
      }
/* . . . . . .COMPUTATION OF TOTAL . . . . . .*/

      total = 0.0 ;
      for( i = 0 ; i < 10 ; i++ )
          total = total + x[i] * x[i] ;

/*. . . . PRINTING OF x[i] VALUES AND TOTAL . . . */

      printf(“\n”);
      for( i = 0 ; i < 10 ; i++ )
         printf(“x[%2d] = %5.2f\n”, i+1, x[i]) ;
```

Contd.

```
            printf("\ntotal = %.2f\n", total) ;
        }
Output
     ENTER 10 REAL NUMBERS

     1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.10

            x[ 1] = 1.10
            x[ 2] = 2.20
            x[ 3] = 3.30
            x[ 4] = 4.40
            x[ 5] = 5.50
            x[ 6] = 6.60
            x[ 7] = 7.70
            x[ 8] = 8.80
            x[ 9] = 9.90
            x[10] = 10.10
            Total = 446.86
```

Fig. 8.1 *Program to illustrate* ***one-dimensional array***

Note *C99 permits arrays whose size can be specified at run time.*

8.4 Initialization of One-dimensional Arrays

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". An array can be initialized at either of the following stages:

- At compile time
- At run time

8.4.1 Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

```
type array-name[size] = { list of values };
```

The values in the list are separated by commas. For example, the statement

```
int number[3] = { 0,0,0 };
```

will declare the variable **number** as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

```
float total[5] = {0.0,15.75,-10};
```

will initialize the first three elements to 0.0, 15.75, and –10.0 and the remaining two elements to zero.

The *size* may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

```
int counter[ ] = {1,1,1,1};
```

will declare the **counter** array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus, the statement

```
char name[ ] = {'J','o', 'h', 'n', '\0'};
```

declares the **name** to be an array of five characters, initialized with the string "John" ending with the null character. Alternatively, we can assign the string literal directly as under:

```
char name [ ] = "John";
```

(Character arrays and strings are discussed in detail in Chapter 9)

Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases, the remaining elements are inilialized to *zero*, if the array type is numeric and *NULL* if the type is char. For example,

```
int number [5] = {10, 20};
```

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0. Similarly, the declaration.

```
char city [5] = {'B'};
```

will initialize the first element to 'B' and the remaining four to NULL. It is a good idea, however, to declare the size explicitly, as it allows the compiler to do some error checking.

Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement

```
int number [3] = {10, 20, 30, 40};
```

will not work. It is illegal in C.

8.4.2 Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
--------
--------
 for (i = 0; i < 100; i = i+1)
 {
    if  i < 50
        sum[i] = 0.0;     /* assignment statement */
    else
        sum[i] = 1.0;
 }
--------
--------
```

The first 50 elements of the array **sum** are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.

We can also use a read function such as **scanf** to initialize an array. For example, the statements

```
int x [3];
scanf("%d%d%d", &x[0], &[1], &x[2]);
```

will initialize array elements with the values entered through the keyboard.

Example 8.2 Given below is the list of marks obtained by a class of 50 students in an annual examination.

43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74 81 49 37
40 49 16 75 87 91 33 24 58 78 65 56 76 67 45 54 36 63 12 21
73 49 51 19 39 49 68 93 85 59

Write a program to count the number of students belonging to each of following groups of marks: 0–9, 10–19, 20–29,.....,100.

The program coded in Fig. 8.2 uses the array **group** containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

Program

```
#define MAXVAL 50
#define COUNTER  11
main()
{
      float         value[MAXVAL];
      int           i, low, high;
      intgroup[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
      /* . . . . . .READING AND COUNTING . . . . . .*/
      for( i = 0 ; i < MAXVAL ; i++ )
      {
      /*. . . . . .READING OF VALUES . . . . . */
         scanf("%f", &value[i]) ;
      /*. . . . . .COUNTING FREQUENCY OF GROUPS. . . . . */
         ++ group[ (int) ( value[i]) / 10] ;
      }
      /* . . . .PRINTING OF FREQUENCY TABLE . . . . . .*/
      printf("\n");
      printf(" GROUP     RANGE     FREQUENCY\n\n") ;
      for( i = 0 ; i < COUNTER ; i++ )
      {
            low = i * 10 ;
            if(i == 10)
               high = 100 ;
```

Contd.

```
                else
                high = low + 9 ;
                printf(" %2d %3d to %3d %d\n",
                          i+1, low, high, group[i] ) ;
            }
        }
```

Output

```
    43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74
    81 49 37 40 49 16 75 87 91 33 24 58 78 65 56 76 67 (Input data)
    45 54 36 63 12 21 73 49 51 19 39 49 68 93 85 59
                GROUP          RANGE          FREQUENCY
                  1          0     to     9         2
                  2         10     to    19         4
                  3         20     to    29         4
                  4         30     to    39         5
                  5         40     to    49         8
                  6         50     to    59         8
                  7         60     to    69         7
                  8         70     to    79         6
                  9         80     to    89         4
                 10         90     to    99         2
                 11        100     to   100         0
```

Fig. 8.2 *Program for* ***frequency counting***

Note that we have used an initialization statement.

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

which can be replaced by

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

This will initialize all the elements to zero.

Searching and Sorting

Searching and sorting are the two most frequent operations performed on arrays. Computer Scientists have devised several data structures and searching and sorting techniques that facilitate rapid access to data stored in lists.

Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an *ordered list.* Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available. The three simple and most important among them are:

- Bubble sort
- Selection sort
- Insertion sort

Contd

Other sorting techniques include Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the *search key.* If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are:

- Sequential search
- Binary search

A detailed discussion on these techniques is beyond the scope of this text. Consult any good book on data structures and algorithms.

8.5 Two-Dimensional Arrays

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	Item1	Item2	Item3
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four *rows* and three *columns*. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

In mathematics, we represent a particular value in a matrix by using two subscripts such as $\mathbf{v_{ij}}$. Here **v** denotes the entire matrix and $\mathbf{v_{ij}}$ refers to the value in the i^{th} row and j^{th} column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as

v[4][3]

Two-dimensional arrays are declared as follows:

```
type array_name [row_size][column_size];
```

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

Two-dimensional arrays are stored in memory, as shown in Fig. 8.3. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

Example 8.3 Write a program to read and display a simple 3 X 3 matrix.

```
Program
     /*Program for realizing a 3 X 3 matrix using 2-D arrays*/
     #include <stdio.h>
```

```
#include <conio.h>
void main()
{
 int i,j,a[3][3];
 clrscr();
 /*Reading matrix elements*/
 printf("Enter the elements of the 3 X 3 matrix:\n");
 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
  {
   printf("a[%d][%d] = ",i,j);
   scanf("%d",&a[i][j]);
  }
 /*Printing matrix elements*/
 printf("The various elements contained in the 3 X 3 matrix are:\n");
 for(i=0;i<3;i++)
 {
  printf("\n\t\t    ");
  for(j=0;j<3;j++)
   printf("%d\t",a[i][j]);
 }
 getch();
}
```

Output

```
Enter the elements of the 3 X 3 matrix:
a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[1][0] = 4
a[1][1] = 5
a[1][2] = 6
a[2][0] = 7
a[2][1] = 8
a[2][2] = 9
The various elements contained in the 3 X 3 matrix are:
                    1   2       3
                    4   5       6
                    7   8       9
```

Fig. 8.3 *Program to read and display a simple 3 × 3 matrix*

Example 8.4 Write a program to add two 3 × 3 matrices.

Program

```
/*Program for adding two 3 X 3 matrices using 2-D arrays*/
#include <stdio.h>
#include <conio.h>
```

Contd

```
void main()
{
 int i,j,a[3][3],b[3][3],c[3][3];
 clrscr();

 printf("Enter the first 3 X 3 matrix:\n");
 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
   printf("a[%d][%d] = ",i,j);
   scanf("%d",&a[i][j]);/*Reading the elements of 1st matrix*/
  }
 }
 printf("Enter the second 3 X 3 matrix:\n");
 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
   printf("b[%d][%d] = ",i,j);
   scanf("%d",&b[i][j]);/*Reading the elements of 2nd matrix*/
  }
 }
 printf("\nThe entered matrices are: \n");
 for(i=0;i<3;i++)
 {
  printf("\n");
  for(j=0;j<3;j++)
   printf("%d\t",a[i][j]);/*Displaying the elements of 1st matrix*/
  printf("\t\t");
  for(j=0;j<3;j++)
   printf("%d\t",b[i][j]);/*Displaying the elements of 2nd matrix*/
 }
 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
   c[i][j] =a[i][j]+b[i][j];/*Computing the sum of two matrices*/
 printf("\n\nThe sum of the two matrices is shown below: \n");
 for(i=0;i<3;i++)
 {
  printf("\n\t\t    ");
  for(j=0;j<3;j++)
   printf("%d\t",c[i][j]);/*Displaying the result*/
 }
 getch();
}
```

Output

```
Enter the first 3 X 3 matrix:
a[0][0] = 1
```

```
a[0][1] = 2
a[0][2] = 3
a[1][0] = 4
a[1][1] = 5
a[1][2] = 6
a[2][0] = 7
a[2][1] = 8
a[2][2] = 9
Enter the second 3 X 3 matrix:
b[0][0] = 9
b[0][1] = 8
b[0][2] = 7
b[1][0] = 6
b[1][1] = 5
b[1][2] = 4
b[2][0] = 3
b[2][1] = 2
b[2][2] = 1
The entered matrices are:
1       2       3                       9       8       7
4       5       6                       6       5       4
7       8       9                       3       2       1

The sum of the two matrices is shown below:

                    10  10      10
                    10  10      10
                    10  10       10
```

Fig. 8.4 *Program to add 3 × 3 matrices*

Example 8.5 Write a program to multiply two 3 × 3 matrices.

Program

```
/*Program for multiplying two 3 X 3 matrices using 2-D arrays*/
#include <stdio.h>
#include <conio.h>

void main()
{
 int i,j,k,a[3][3],b[3][3],c[3][3];
 clrscr();
 printf("Enter the first 3 X 3 matrix:\n");
 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
```

Contd

```
    printf("a[%d][%d] = ",i,j);
    scanf("%d",&a[i][j]);/*Reading the elements of the 1st matrix*/
   }
  }

  printf("Enter the second 3 X 3 matrix:\n");
  for(i=0;i<3;i++)
  {
   for(j=0;j<3;j++)
   {
    printf("b[%d][%d] = ",i,j);
    scanf("%d",&b[i][j]);/*Reading the elements of the 2nd matrix*/
   }
  }
  printf("\nThe entered matrices are: \n");

  for(i=0;i<3;i++)
  {
   printf("\n");
   for(j=0;j<3;j++)
   {
    printf("%d\t",a[i][j]);/*Displaying the elements of the 1st matrix*/
   }
   printf("\t\t");
   for(j=0;j<3;j++)
   {
    printf("%d\t",b[i][j]);/*Displaying the elements of the 2nd matrix*/
   }
  }
  /*Multiplying the two matrices*/
  for(i=0;i<3;i++)
   for(j=0;j<3;j++)
   {
    c[i][j]=0;
    for(k=0;k<3;k++)
     c[i][j]=c[i][j]+a[i][k]*b[k][j];
   }
  printf("\n\nThe product of the two matrices is shown below: \n");

  for(i=0;i<3;i++)
  {
   printf("\n\t\t    ");
   for(j=0;j<3;j++)
   {
    printf("%d\t",c[i][j]); /*Displaying the result*/
   }
  }
```

Contd

```
    getch();
    }
Output
    Enter the first 3 X 3 matrix:
    a[0][0] = 1
    a[0][1] = 2
    a[0][2] = 3
    a[1][0] = 4
    a[1][1] = 5
    a[1][2] = 6
    a[2][0] = 7
    a[2][1] = 8
    a[2][2] = 9
    Enter the second 3 X 3 matrix:
    b[0][0] = 9
    b[0][1] = 8
    b[0][2] = 7
    b[1][0] = 6
    b[1][1] = 5
    b[1][2] = 4
    b[2][0] = 3
    b[2][1] = 2
    b[2][2] = 1

    The entered matrices are:
    1       2       3                       9       8       7
    4       5       6                       6       5       4
    7       8       9                       3       2       1

    The product of the two matrices is shown below:
                        30  24      18
                        84  69      54
                        138 114     90
```

Fig. 8.5 *Program to multiply two 3 × 3 matrix*

Example 8.6 Write a program to generate the transpose of a 3 × 3 matrix.

```
Program
    /*Program for generating the transpose of a 3X3 matrix using 2-D arrays*/
    #include <stdio.h>
    #include <conio.h>

    void main()
    {
```

Contd.

```
       int i,j,a[3][3],b[3][3];
       clrscr();
       printf("Enter a 3 X 3 matrix:\n");
       for(i=0;i<3;i++)
       {
        for(j=0;j<3;j++)
        {
         printf("a[%d][%d] = ",i,j);
         scanf("%d",&a[i][j]); /*Reading the elements of the 3X3 matrix*/
        }
       }

       printf("\nThe entered matrix is: \n");
       for(i=0;i<3;i++)
       {
        printf("\n");
        for(j=0;j<3;j++)
        {
         printf("%d\t",a[i][j]); /*Displaying the matrix*/
        }
       }

       for(i=0;i<3;i++)
       {
        for(j=0;j<3;j++)
         b[i][j]=a[j][i]; /*Computing matrix transpose*/
       }
       printf("\n\nThe transpose of the matrix is: \n");

       for(i=0;i<3;i++)
       {
        printf("\n");
        for(j=0;j<3;j++)
        {
         printf("%d\t",b[i][j]); /*Displaying the resultant transposed matrix*/
        }
       }
       getch();
      }
```

Output

```
     Enter a 3 X 3 matrix:
     a[0][0] = 1
     a[0][1] = 2
     a[0][2] = 3
     a[1][0] = 4
     a[1][1] = 5
     a[1][2] = 6
```

Contd.

```
a[2][0] = 7
a[2][1] = 8
a[2][2] = 9
The entered matrix is:
1       2       3
4       5       6
7       8       9

The transpose of the matrix is:
1       4       7
2       5       8
3       6       9
```

Fig. 8.6 *Program to generate the transpose of a 3 X 3 matrix]*

Example 8.7 Write a program using a two-dimensional array to compute and print the following information from the table of data discussed above:

(a) Total value of sales by each girl.
(b) Total value of each item sold.
(c) Grand total of sales of all items by all girls.

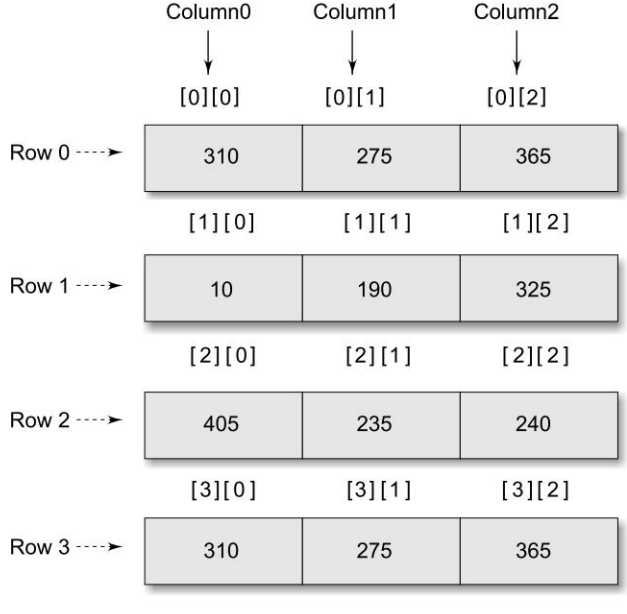


Fig. 8.7 *Representation of a **two-dimensional** array in memory*

The program and its output are shown in Fig. 8.8. The program uses the variable **value** in two-dimensions with the index i representing girls and j representing items. The following equations are used in computing the results:

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

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

(c) Grand total = $\sum_{i=0}^{3} \sum_{j=0}^{2}$ value[i][j]

= $\sum_{i=0}^{3}$ girl_total[i]

= $\sum_{j=0}^{2}$ item_total[j]

Program

```
#define MAXGIRLS  4
#define MAXITEMS  3
main()
{
   int value[MAXGIRLS][MAXITEMS];
   int girl_total[MAXGIRLS] , item_total[MAXITEMS];
   int i, j, grand_total;
/*.......READING OF VALUES AND COMPUTING girl_total ...*/

   printf("Input data\n");
   printf("Enter values, one at a time, row-wise\n\n");

   for( i = 0 ; i < MAXGIRLS ; i++ )
   {
         girl_total[i] = 0;
         for( j = 0 ; j < MAXITEMS ; j++ )
         {
               scanf("%d", &value[i][j]);
               girl_total[i] = girl_total[i] + value[i][j];
         }
   }
/*.......COMPUTING item_total.........................*/

   for( j = 0 ; j < MAXITEMS ; j++ )
   {
         item_total[j] = 0;
         for( i =0 ; i < MAXGIRLS ; i++ )
               item_total[j] = item_total[j] + value[i][j];
   }
/*.......COMPUTING grand_total.........................*/
```

Contd.

```
        grand_total = 0;
        for( i =0 ; i < MAXGIRLS ; i++ )
            grand_total = grand_total + girl_total[i];
    /* .......PRINTING OF RESULTS.........................*/

    printf("\n GIRLS TOTALS\n\n");
        for( i = 0 ; i < MAXGIRLS ; i++ )
              printf("Salesgirl[%d] = %d\n", i+1, girl_total[i] );
        printf("\n ITEM TOTALS\n\n");
        for( j = 0 ; j < MAXITEMS ; j++ )
              printf("Item[%d] = %d\n", j+1 , item_total[j] );
            printf("\nGrand Total = %d\n", grand_total);
        }
Output
    Input data
    Enter values, one at a time, row_wise
    310 257 365
    210 190 325
    405 235 240
    260 300 380
    GIRLS TOTALS
    Salesgirl[1] = 950
    Salesgirl[2] = 725
    Salesgirl[3] = 880
    Salesgirl[4] = 940
    ITEM TOTALS
    Item[1] = 1185
    Item[2] = 1000
    Item[3] = 1310
    Grand Total = 3495
```

Fig. 8.8 *Illustration of two-dimensional arrays*

Example 8.8 Write a program to compute and print a multiplication table for numbers 1 to 5 as shown below:

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	.	.	.
4	4	8	.	.	.
5	5	10	.	.	25

The program shown in Fig. 8.9 uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested for loops as follows:

```
product[i][j] = row * column
```

where i denotes rows and j denotes columns of the product table. Since the indices i and j range from 0 to 4, we have introduced the following transformation:

$$row = i+1$$
$$column = j+1$$

Program

```
#define ROWS      5
#define COLUMNS   5
main()
{
   int row, column, product[ROWS][COLUMNS] ;
   int i, j ;
   printf(" MULTIPLICATION TABLE\n\n") ;
   printf(" ") ;
   for( j = 1 ; j <= COLUMNS ; j++ )
      printf("%4d" , j ) ;
   printf("\n") ;
   printf("————————————————————————————————\n");
   for( i = 0 ; i < ROWS ; i++ )
   {
         row = i + 1 ;
         printf("%2d |", row) ;
         for( j = 1 ; j <= COLUMNS ; j++ )
         {
            column = j ;
            product[i][j] = row * column ;
            printf("%4d", product[i][j] ) ;
         }
         printf("\n") ;
   }
}
```

Output

```
MULTIPLICATION TABLE
      1   2   3   4   5
 1 |  1   2   3   4   5
 2 |  2   4   6   8  10
 3 |  3   6   9  12  15
 4 |  4   8  12  16  20
 5 |  5  10  15  20  25
```

Fig. 8.9 *Program to print multiplication table using two-dimensional array*

8.6 Initializing Two-Dimensional Arrays

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
int table[2][3] = { 0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

```
int table[2][3] = {{0,0,0}, {1,1,1}};
```

by surrounding the elements of the each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int table[2][3] = {
                      {0,0,0},
                      {1,1,1}
                  };
```

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension. That is, the statement is permitted.

```
int table [ ] [3] =  {
                        { 0, 0, 0},
                        { 1, 1, 1}
                     };
```

If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

```
int table[2][3] =  {
                      {1,1},
                      {2}
                   };
```

will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int m[3][5] = { {0}, {0}, {0}};
```

The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero. The following statement will also achieve the same result:

```
int m [3] [5] = { 0, 0};
```

Example 8.9 A survey to know the popularity of four cars (Ambassador, Fiat, Dolphin and Maruti) was conducted in four cities (Bombay, Calcutta, Delhi and Madras). Each person surveyed was asked to give his city and the type of car he was using. The results, in coded form, are tabulated as follows:

M	1	C	2	B	1	D	3	M	2	B	4
C	1	D	3	M	4	B	2	D	1	C	3
D	4	D	4	M	1	M	1	B	3	B	3
C	1	C	1	C	2	M	4	M	4	C	2
D	1	C	2	B	3	M	1	B	1	C	2
D	3	M	4	C	1	D	2	M	3	B	4

Codes represent the following information:

M – Madras	1 – Ambassador
D – Delhi	2 – Fiat
C – Calcutta	3 – Dolphin
B – Bombay	4 – Maruti

Write a program to produce a table showing popularity of various cars in four cities.

A two-dimensional array **frequency** is used as an accumulator to store the number of cars used, under various categories in each city. For example, the element **frequency** [i][j] denotes the number of cars of type j used in city i. The **frequency** is declared as an array of size 5 × 5 and all the elements are initialized to zero.

The program shown in Fig. 8.10 reads the city code and the car code, one set after another, from the terminal. Tabulation ends when the letter X is read in place of a city code.

Program

```
main()
{
   int i, j, car;
   int frequency[5][5] = { {0},{0},{0},{0},{0} };
   char city;
   printf("For each person, enter the city code \n");
   printf("followed by the car code.\n");
   printf("Enter the letter X to indicate end.\n");
/*. . . . . TABULATION BEGINS . . . . . */
   for( i = 1 ; i < 100 ; i++ )
   {
      scanf("%c", &city );
      if( city == 'X' )
         break;
      scanf("%d", &car );
      switch(city)
      {
               case 'B' :   frequency[1][car]++;
                            break;
               case 'C' :   frequency[2][car]++;
                            break;
               case 'D' :   frequency[3][car]++;
                            break;
               case 'M' :   frequency[4][car]++;
                            break;
         }
```

Contd

```
        }
    /*. . . . .TABULATION COMPLETED AND PRINTING BEGINS. . . .*/
        printf("\n\n");
        printf(" POPULARITY TABLE\n\n");
        printf("-------------------------------------\n");
        printf("City Ambassador Fiat Dolphin Maruti \n");
        printf("-------------------------------------\n");
        for( i = 1 ; i <= 4 ; i++ )
        {
            switch(i)
            {
                case 1 : printf("Bombay ") ;
                    break ;
            case 2 : printf("Calcutta ") ;
                    break ;
            case 3 : printf("Delhi ") ;
                    break ;
            case 4 : printf("Madras ") ;
                    break ;
        }
        for( j = 1 ; j <= 4 ; j++ )
           printf("%7d", frequency[i][j] ) ;
        printf("\n") ;
      }
      printf("-------------------------------------------------\n");
    /*. . . . . PRINTING ENDS. . . . . .*/
    }
```

Output

```
For each person, enter the city code
followed by the car code.
Enter the letter X to indicate end.
M  1  C  2  B  1  D  3  M  2  B  4
C  1  D  3  M  4  B  2  D  1  C  3
D  4  D  4  M  1  M  1  B  3  B  3
C  1  C  1  C  2  M  4  M  4  C  2
D  1  C  2  B  3  M  1  B  1  C  2
D  3  M  4  C  1  D  2  M  3  B  4     X
```

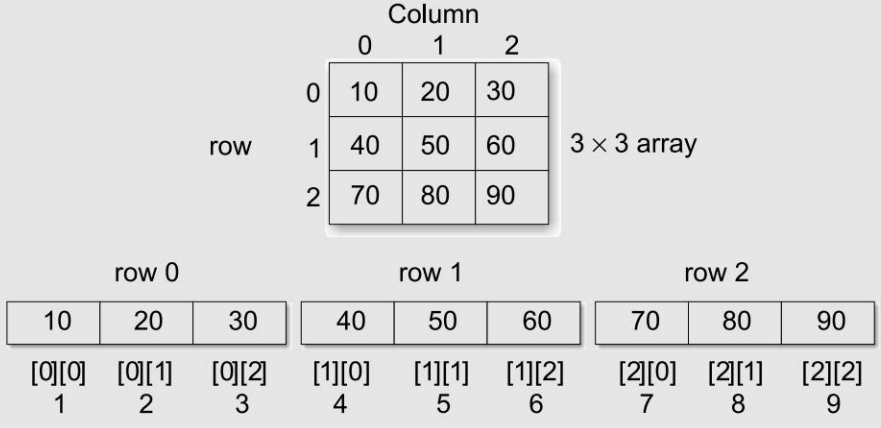
POPULARITY TABLE

City	Ambassador	Fiat	Dolphin	Maruti
Bombay	2	1	3	2
Calcutta	4	5	1	0
Delhi	2	1	3	2
Madras	4	1	1	4

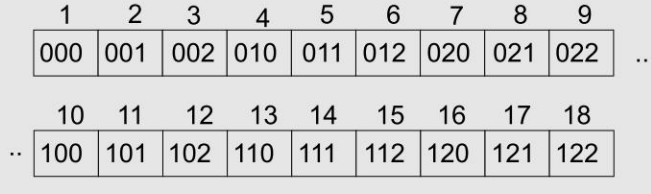
Fig. 8.10 *Program to tabulate a survey data*

Memory Layout

The subscripts in the definition of a two-dimensional array represent rows and columns. This format maps the way that data elements are laid out in the memory. The elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we consider the memory as a row of bytes, with the lowest address on the left and the highest address on the right, a simple array will be stored in memory with the first element at the left end and the last element at the right end. Similarly, a two-dimensional array is stored "row-wise, starting from the first row and ending with the last row, treating each row like a simple array. This is illustrated below.



For a multi-dimensional array, the order of storage is that the first element stored has 0 in all its subscripts, the second has all of its subscripts 0 except the far right which has a value of 1 and so on. The elements of a 2 x 3 x 3 array will be stored as under



The far right subscript increments first and the other subscripts increment in order from right to left. The sequence numbers 1, 2,……, 18 represents the location of that element in the list

8.7 Multi-Dimensional Arrays

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

```
type array_name[s1][s2][s3]....[sm];
```

where s_i is the size of the ith dimension. Some examples are:

```
int survey[3][5][12];
```

```
float table[5][4][5][3];
```

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly **table** is a four-dimensional array containing 300 elements of floating-point type.

The array **survey** may represent a survey data of rainfall during the last three years from January to December in five cities.

If the first index denotes year, the second city and the third month, then the element **survey[2][3][10]** denotes the rainfall in the month of October during the second year in city-3.

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

Year 1

month city	1	2	.....	12
1 5				

Year 2

month *city*	1	2	.....	12
1 5				

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

8.8 Dynamic Arrays

So far, we created arrays at compile time. An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run time. The process of allocating memory at compile time is known as *static memory allocation* and the arrays that receive static memory allocation are called *static arrays*. This approach works fine as long as we know exactly what our data requirements are.

Consider a situation where we want to use an array that can vary greatly in size. We must guess what will be the largest size ever needed and create the array accordingly. A difficult task in fact! Modern languages like C do not have this limitation. In C it is possible to allocate memory to arrays at run time. This feature is known as *dynamic memory allocation* and the arrays created at run time are called *dynamic* arrays. This effectively postpones the array definition to run time.

Dynamic arrays are created using what are known as *pointer variables* and *memory management functions* **malloc**, **calloc** and **realloc**. These functions are included in the header file **<stdlib.h>.** The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues. We discuss in detail pointers and pointer variables in Chapter 11.

Just Remember

- We need to specify three things, namely, name, type and size, when we declare an array.
- Always remember that subscripts begin at 0 (not 1) and end at size –1.
- Defining the size of an array as a symbolic constant makes a program more scalable.
- Be aware of the difference between the "kth element" and the "element k". The kth element has a subscript k-1, whereas the element k has a subscript of k itself.
- Do not forget to initialize the elements; otherwise they will contain "garbage".
- Supplying more initializers in the initializer list is a compile time error.
- Use of invalid subscript is one of the common errors. An incorrect or invalid index may cause unexpected results.
- When using expressions for subscripts, make sure that their results do not go outside the permissible range of 0 to size –1. Referring to an element outside the array bounds is an error.
- When using control structures for looping through an array, use proper relational expressions to eliminate "off-by-one" errors. For example, for an array of size 5, the following **for** statements are wrong:

```
for (i = 1; i < =5; i+ +)
for (i = 0; i < =5; i+ +)
for (i = 0; i = =5; i+ +)
for (i = 0; i < 4; i+ +)
```

- Referring a two-dimensional array element like x[i, j] instead of x[i] [j] is a compile time error.
- When initializing character arrays, provide enough space for the terminating null character.
- Make sure that the subscript variables have been properly initialized before they are used.
- Leaving out the subscript reference operator [ ] in an assignment operation is compile time error.
- During initialization of multi–dimensional arrays, it is an error to omit the array size for any dimension other than the first.

Review Questions

8.1 State whether the following statements are *true* or *false*.

(a) The type of all elements in an array must be the same.

(b) When an array is declared, C automatically initializes its elements to zero.

(c) An expression that evaluates to an integral value may be used as a subscript.

(d) Accessing an array outside its range is a compile time error.

(e) A **char** type variable cannot be used as a subscript in an array.

(f) An unsigned long int type can be used as a subscript in an array.

(g) In C, by default, the first subscript is zero.
(h) When initializing a multidimensional array, not specifying all its dimensions is an error.
(i) When we use expressions as a subscript, its result should be always greater than zero.
(j) In C, we can use a maximum of 4 dimensions for an array.
(k) In declaring an array, the array size can be a constant or variable or an expression.
(l) The declaration int x[2] = {1,2,3}; is illegal.

8.2 Fill in the blanks in the following statements.
(a) The variable used as a subscript in an array is popularly known as _____ variable.
(b) An array can be initialized either at compile time or at _____.
(c) An array created using **malloc** function at run time is referred to as _____ array.
(d) An array that uses more than two subscript is referred to as _____ array.
(e) _____ is the process of arranging the elements of an array in order.

8.3 Identify errors, if any, in each of the following array declaration statements, assuming that ROW and COLUMN are declared as symbolic constants:

(a)
```
int score (100);
```
(b)
```
float values [10,15];
```
(c)
```
float average[ROW],[COLUMN];
```
(d)
```
char name[15];
```
(e)
```
int sum[ ];
```
(f)
```
double salary [i + ROW]
```
(g)
```
long int number [ROW]
```
(h)
```
int array x[COLUMN];
```

8.4 Identify errors, if any, in each of the following initialization statements.

(a)
```
int number[ ] = {0,0,0,0,0};
```
(b)
```
float item[3][2] = {0,1,2,3,4,5};
```
(c)
```
char word[ ] = {'A','R', 'R', 'A', 'Y'};
```
(d)
```
int m[2,4] = {(0,0,0,0)(1,1,1,1)};
```
(e)
```
float result[10] = 0;
```

8.5 Assume that the arrays A and B are declared as follows:

```
int A[5][4];
float B[4];
```

Find the errors (if any) in the following program segments.

(a)
```
for (i=1; i<=5; i++)
   for(j=1; j<=4; j++)
   A[i][j] = 0;
```
(b)
```
for (i=1; i<4; i++)
   scanf("%f", B[i]);
```
(c)
```
for (i=0; i<=4; i++)
   B[i] = B[i]+i;
```
(d)
```
for (i=4; i>=0; i--)
   for (j=0; j<4; j++)
   A[i][j] = B[j] + 1.0;
```

8.6 Write a **for** loop statement that initializes all the diagonal elements of an array to one and others to zero as shown below. Assume 5 rows and 5 columns.

1	0	0	0	0	.....	0
0	1	0	0	0	.....	0
0	0	1	0	0	.....	0
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
0	0	0	0	0	.....	1

8.7 We want to declare a two-dimensional integer type array called **matrix** for 3 rows and 5 columns. Which of the following declarations are correct?

(a) `int maxtrix [3],[5];`
(b) `int matrix [5] [3];`
(c) `int matrix [1+2] [2+3];`
(d) `int matrix [3,5];`
(e) `int matrix [3] [5];`

8.8 Which of the following initialization statements are correct?

(a) `char str1[4] = "GOOD";`
(b) `char str2[ ] = "C";`
(c) `char str3[5] = "Moon";`
(d) `char str4[ ] = {'S', 'U', 'N'};`
(e) `char str5[10] = "Sun";`

8.9 What is a data structure? Why is an array called a data structure?

8.10 What is a dynamic array? How is it created? Give a typical example of use of a dynamic array.

8.11 What is the error in the following program?

```
main ( )
{
        int x ;
        float y [ ] ;
        ......
}
```

8.12 What happens when an array with a specified size is assigned

(a) with values fewer than the specified size; and
(b) with values more than the specified size.

8.13 Discuss how initial values can be assigned to a multidimensional array.

8.14 What is the output of the following program?

```
main ( )
{
        int m [ ] = { 1,2,3,4,5 }
        int x, y = 0;
```

```
        for (x = 0; x < 5; x++ )
                y = y + m [ x ];
        printf("%d", y) ;
}
```

8.15 What is the output of the following program?

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

Programming Exercises

8.1 Write a program for fitting a straight line through a set of points (x_i,y_i), i = 1,....,n.
The straight line equation is

$$y = mx + c$$

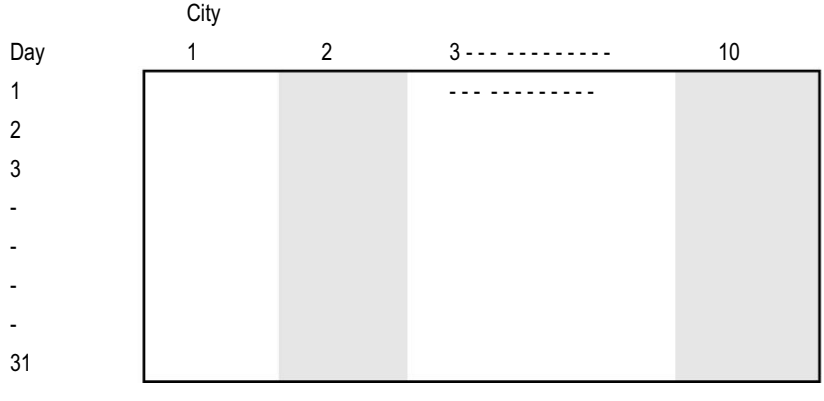
and the values of m an c are gven by

$$m = \frac{n\Sigma(x_1 y_i) - (\Sigma x_1)(\Sigma y_i)}{n\left(\Sigma x_i^2\right) - (\Sigma x_i)^2}$$

$$c = \frac{1}{n}(\Sigma\, y_i - m\,\Sigma\, x_i)$$

All summations are from 1 to n.

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



Write a program to read the table elements into a two-dimensional array **temperature**, and to find the city and day corresponding to

(a) the highest temperature and

(b) the lowest temperature.

8.3 An election is contested by 5 candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the votes cast for each candidate using an array variable **count**. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot' and the program should also count the number of spoilt ballots.

8.4 The following set of numbers is popularly known as Pascal's triangle.

```
1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
-       -       -       -       -       -       -
-       -       -       -       -       -       -       -
```

If we denote rows by i and columns by j, then any element (except the boundary elements) in the triangle is given by

$$p_{ij} = p_{i-1,\, j-1} + p_{i-1,j}$$

Write a program to calculate the elements of the Pascal triangle for 10 rows and print the results.

8.5 The annual examination results of 100 students are tabulated as follows:

Roll No.	Subject 1	Subject 2	Subject 3
.			
.			
.			

Write a program to read the data and determine the following:

(a) Total marks obtained by each student.

(b) The highest marks in each subject and the Roll No. of the student who secured it.

(c) The student who obtained the highest total marks.

8.6 Given are two one-dimensional arrays A and B which are sorted in ascending order.Write a program to **merge** them into a single sorted array C that contains every item from arrays A and B, in ascending order.

8.7 Two matrices that have the same number of rows and columns can be multiplied to produce a third matrix. Consider the following two matrices.

$$A = \begin{bmatrix} a_{11}\ a_{12} \ldots\ldots a_{1n} \\ a_{12}\ a_{22} \ldots\ldots a_{2n} \\ . \qquad\qquad . \\ . \qquad\qquad . \\ . \qquad\qquad . \\ a_{n1} \ldots\ldots\ldots\ldots a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11}\ b_{12} \ldots\ldots b_{1n} \\ b_{12}\ b_{22} \ldots\ldots b_{2n} \\ . \qquad\qquad . \\ . \qquad\qquad . \\ . \qquad\qquad . \\ b_{n1} \ldots\ldots\ldots\ldots b_{nn} \end{bmatrix}$$

The product of **A** and **B** is a third matrix C of size $n \times n$ where each element of C is given by the following equation.

$$\mathbf{C}_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Write a program that will read the values of elements of A and B and produce the product matrix **C.**

8.8 Write a program that fills a five-by-five matrix as follows:

- Upper left triangle with +1s
- Lower right triangle with –1s
- Right to left diagonal with zeros

Display the contents of the matrix using not more than two **printf** statements

8.9 Selection sort is based on the following idea:

Selecting the largest array element and swapping it with the last array element leaves an unsorted list whose size is 1 less than the size of the original list. If we repeat this step again on the unsorted list we will have an ordered list of size 2 and an unordered list size n–2 . When we repeat this until the size of the unsorted list becomes one, the result will be a sorted list.

Write a program to implement this algorithm.

8.10 Develop a program to implement the binary search algorithm. This technique compares the search key value with the value of the element that is midway in a "sorted" list. Then;

(a) If they match, the search is over.

(b) If the search key value is less than the middle value, then the first half of the list contains the key value.

(c) If the search key value is greater than the middle value, then the second half contains the key value.

Repeat this "divide-and-conquer" strategy until we have a match. If the list is reduced to one non-matching element, then the list does not contain the key value.

Use the sorted list created in Exercise 8.9 or use any other sorted list.

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

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

8.13 Write a program to read a matrix of size m × n and print its transpose.

8.14 Every book published by international publishers should carry an International Standard Book Number (ISBN). It is a 10 character 4 part number as shown below.

0-07-041183-2

The first part denotes the region, the second represents publisher, the third identifies the book and the fourth is the check digit. The check digit is computed as follows:

Sum = (1 × first digit) + (2 × second digit) + (3 × third digit) + - - - - + (9 × ninth digit).

Check digit is the remainder when sum is divided by 11. Write a program that reads a given ISBN number and checks whether it represents a valid ISBN.

8.15 Write a program to read two matrices A and B and print the following:

(a) A + B; and (b) A – B.

Debugging Exercises

8.1 Identify the error in the following program, if any:

```
#include <stdio.h>

void main( )
{
 int i;
 int arr[5];

 printf("Enter the array values:\n");
 for(i=0;i<=5;i++)
  scanf("%d", &arr[i]);

}
```

8.2 Identify the error in the following program that computes the sum of array elements:

```
#include <stdio.h>

void main( )
{
 int i,sum;
 int arr[5];

 printf("Enter the array values:\n");
 for(i=0;i<5;i++)
    scanf("%d", arr[i]); /*Reading values to be stored in the
array*/
```

```
 printf("\nArray arr[5] contains the following elements:\n");
 for(i=0;i<5;i++)
  printf("arr[%d] = %d\n", i,arr[i]);/*Printing array values*/

 sum=0;
 for(i=0;i<5;i++)
  sum=sum+arr[i];/*Computing sum of elements stored in the array*/

  printf("\nSum of array elements = %d",sum); /*Displaying the
result*/

}
```

8.3 Will the following program compile successfully? If yes, what will be output?

```
#include <stdio.h>

void main()
{
 int i,j;
 int arr[2][2]={1,2,3};

 for(i=0;i<2;i++)
  for(j=0;j<2;j++)
   printf("arr[%d][%d] = %d\n",i,j,arr[i][j]);

}
```

8.4 Identify the error in the following program, if any:

```
#include <stdio.h>
void main()
{
 int a[]={1,2,3,4,5};
 int b[5];
 int i;

 b[]=a[];
 for(i=0;i<5;i++)
  printf("b[%d] = %d\n",i,b[i]);

}
```



9 Strings

9.1 Introduction

A string is a sequence of characters that is treated as a single data item. We have used strings in a number of examples in the past. Any group of characters (except double quote sign) defined between double quotation marks is a string constant. Example:

"Man is obviously made to think."

If we want to include a double quote in the string to be printed, then we may use it with a back slash as shown below.

"\" Man is obviously made to think,\" said Pascal."

For example,

```
printf ("\" Well Done !"\");
```

will output the string

" Well Done !"

while the statement

```
printf(" Well Done !");
```

will output the string

Well Done !

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

- Reading and writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

In this chapter we shall discuss these operations in detail and examine library functions that implement them.

9.2 Declaring and Initializing String Variables

C does not support strings as a data type. However, it allows us to represent strings as character arrays. In C, therefore, a string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is:

```
char string_name[ size ];
```

The *size* determines the number of characters in the string_name. Some examples are:

```
char city[10];
char name[30];
```

When the compiler assigns a character string to a character array, it automatically supplies a *null* character (‘\0 ‘) at the end of the string. Therefore, the *size* should be equal to the maximum number of characters in the string *plus* one.

Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
char city [9] = " NEW YORK ";
char city [9]={'N','E','W',' ','Y','O','R','K','\0'};
```

The reason that **city** had to be 9 elements long is that the string NEW YORK contains 8 characters and one element space is provided for the null terminator. Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator.

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example, the statement

```
char string [ ] = {'G','O','O','D','\0'};
```

defines the array **string** as a five element array.

We can also declare the size much larger than the string size in the initializer. That is, the statement.

```
char str[10] = "GOOD";
```

is permitted. In this case, the computer creates a character array of size 10, places the value “GOOD” in it, terminates with the null character, and initializes all other elements to NULL. The storage will look like:

G	O	O	D	\0	\0	\0	\0	\0	\0

However, the following declaration is illegal.

```
char str2[3] = "GOOD";
```

This will result in a compile time error. Also note that we cannot separate the initialization from declaration. That is,

```
char str3[5];
str3 = "GOOD";
```

is not allowed. Similarly,

```
char s1[4] = "abc";
char s2[4];
s2 = s1; /* Error */
```

is not allowed. An array name cannot be used as the left operand of an assignment operator.

Terminating Null Character

You must be wondering, "why do we need a terminating null character?" As we know, a string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the "end-of-string" marker.

9.3 Reading Strings from Terminal

9.3.1 Using *scanf* Function

The familiar input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[10]
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal,

NEW YORK

then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the reading of string.

The **scanf** function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character. Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

The **address** array is created in the memory as shown below:

N	E	W	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Note that the unused locations are filled with garbage.

If we want to read the entire line "NEW YORK", then we may use two character arrays of appropriate sizes. That is,

```
char adr1[5], adr2[5];
scanf("%s %s", adr1, adr2);
```

with the line of text

NEW YORK

will assign the string "NEW" to **adr1** and "YORK" to **adr2**.

Example 9.1 Write a program to read a series of words from a terminal using **scanf** function.

The program shown in Fig. 9.1 reads four words and displays them on the screen. Note that the string 'Oxford Road' is treated as *two words* while the string 'Oxford-Road' as *one word.*

Program

```
main( )
{
      char word1[40], word2[40], word3[40], word4[40];
      printf("Enter text : \n");
      scanf("%s %s", word1, word2);
      scanf("%s", word3);
      scanf("%s", word4);
      printf("\n");
      printf("word1 = %s\nword2 = %s\n", word1, word2);
      printf("word3 = %s\nword4 = %s\n", word3, word4);
}
```

Output

```
Enter text :
Oxford Road, London M17ED
word1 = Oxford
word2 = Road,
word3 = London
word4 = M17ED
Enter text :
Oxford-Road, London-M17ED United Kingdom
word1 = Oxford-Road
word2 = London-M17ED
word3 = United
word4 = Kingdom
```

Fig. 9.1 *Reading a series of words using* ***scanf*** *function*

We can also specify the field width using the form %ws in the **scanf** statement for reading a specified number of characters from the input string . Example:

```
scanf("%ws", name);
```

Here, two things may happen.

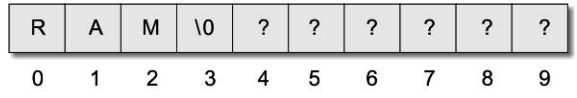
1. The width **w** is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.

2. The width **w** is less than the number of characters in the string. The excess characters will be truncated and left unread.

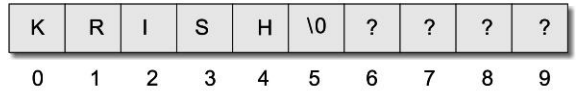
Consider the following statements:

```
char name[10];
scanf("%5s", name);
```

The input string RAM will be stored as:



The input string KRISHNA will be stored as:



9.3.2 Reading a Line of Text

We have seen just now that **scanf** with %**s** or %**ws** can read only strings without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the *edit set conversion code* %[. .] that can be used to read a line containing a variety of characters, including whitespaces. Recall that we have used this conversion code in Chapter 5. For example, the program segment

```
char line [80];
scanf("%[^\n]", line);
printf("%s", line);
```

will read a line of input from the keyboard and display the same on the screen. We would very rarely use this method, as C supports an intrinsic string function to do this job. This is discussed in the next section.

9.3.3 Using *getchar* and *gets* Functions

We have discussed in Chapter 5 as to how to read a single character from the terminal, using the function **getchar.** We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array. The reading is terminated when the newline character ('\n') is entered and the null character is then inserted at the end of the string. The **getchar** function call takes the form:

```
char ch;
ch = getchar( );
```

Note *that the getchar function has no parameters.*

Example 9.2 Write a program to read a line of text containing a series of words from the terminal.

The program shown in Fig. 9.2 can read a line of text (up to a maximum of 80 characters) into the string **line** using **getchar** function. Every time a character is read, it is assigned to its location in the string **line** and then tested for *newline* character. When the *newline* character is read (signalling the end of line), the reading loop is terminated and the *newline* character is replaced by the null character to indicate the end of character string.

When the loop is exited, the value of the index **c** is one number higher than the last character position in the string (since it has been incremented after assigning the new character to the string). Therefore the index value **c-1** gives the position where the *null* character is to be stored.

Program

```
#include <stdio.h>
main( )
{
   char line[81], character;
   int c;
   c = 0;
   printf("Enter text. Press <Return> at end\n");
   do
   {
   character = getchar();
   line[c] = character;
   c++;
   }
   while(character != '\n');
   c = c - 1;
   line[c] = '\0';
   printf("\n%s\n", line);
   }
```

Output

```
Enter text. Press <Return> at end
Programming in C is interesting.
Programming in C is interesting.
Enter text. Press <Return> at end
National Centre for Expert Systems, Hyderabad.
National Centre for Expert Systems, Hyderabad.
```

Fig. 9.2 *Program to **read a line of text** from terminal*

Another and more convenient method of reading a string of text containing whitespaces is to use the library function **gets** available in the <*stdio.h*> header file. This is a simple function with one string parameter and called as under:

gets (str);

str is a string variable declared properly. It reads characters into **str** from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike **scanf,** it does not skip whitespaces. For example the code segment

```
char line [80];
gets (line);
printf ("%s", line);
```

reads a line of text from the keyboard and displays it on the screen. The last two statements may be combined as follows:

```
printf("%s", gets(line));
```

(Be careful not to input more character that can be stored in the string variable used. Since C does not check array-bounds, it may cause problems.)

C does not provide operators that work on strings directly. For instance we cannot assign one string to another directly. For example, the assignment statements.

```
string = "ABC";
string1 = string2;
```

are not valid. If we really want to copy the characters in **string2** into **string1**, we may do so on a character-by-character basis.

Example 9.3 Write a program to copy one string into another and count the number of characters copied.

The program is shown in Fig. 9.3. We use a **for** loop to copy the characters contained inside **string2** into the **string1**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **string1.**

Program

```
main( )
{
   char string1[80], string2[80];
      int i;
      printf("Enter a string \n");
      printf("?");
      scanf("%s", string2);
      for( i=0 ; string2[i] != '\0'; i++)
         string1[i] = string2[i];
      string1[i] = '\0';
      printf("\n");
      printf("%s\n", string1);
      printf("Number of characters = %d\n", i );
   }
```

Output

```
Enter a string
?Manchester
Manchester
Number of characters = 10
Enter a string
?Westminster
Westminster
Number of characters = 11
```

Fig. 9.3 *Copying one string into another*

9.4 Writing Strings to Screen

9.4.1 Using *printf* Function

We have used extensively the **printf** function with %s format to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character. For example, the statement

```
printf("%s", name);
```

can be used to display the entire contents of the array **name**.
We can also specify the precision with which the array is displayed. For instance, the specification

```
%10.4
```

indicates that the *first four* characters are to be printed in a field width of 10 columns.

However, if we include the minus sign in the specification (e.g., %-10.4s), the string will be printed left-justified. The Example 9.4 illustrates the effect of various %s specifications.

Example 9.4 Write a program to store the string "United Kingdom" in the array **country** and display the string under various format specifications.

The program and its output are shown in Fig. 9.4. The output illustrates the following features of the **%s** specifications.

1. When the field width is less than the length of the string, the entire string is printed.
2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed left-justified.
5. The specification % **.**ns prints the first n characters of the string.

Program

```
main()
{
   char country[15] = "United Kingdom";
   printf("\n\n");
   printf("*123456789012345*\n");
   printf(" ------ \n");
   printf("%15s\n", country);
   printf("%5s\n", country);
   printf("%15.6s\n", country);
   printf("%-15.6s\n", country);
   printf("%15.0s\n", country);
   printf("%.3s\n", country);
   printf("%s\n", country);
   printf("------ \n");
}
```

Contd.

```
Output
   *123456789012345*
   _____
   United Kingdom
   United Kingdom
          United
   United
   Uni
   United Kingdom
   _____
```

Fig. 9.4 *Writing strings using %s format*

The **printf** on UNIX supports another nice feature that allows for variable field width or precision. For instance

```
printf("%*.*s\n", w, d, string);
```

prints the first **d** characters of the string in the field width of **w**.
This feature comes in handy for printing a sequence of characters. Example 9.5 illustrates this.

Example 9.5 Write a program using **for loop** to print the following output:

```
C
CP
CPr
CPro
.....
.....
CProgramming
CProgramming
.....
.....
CPro
CPr
CP
C
```

The outputs of the program in Fig. 9.5, for variable specifications **%12.*s, %.*s,** and **%*.1s** are shown in Fig. 9.6, which further illustrates the variable field width and the precision specifications.

```
Program
   main()
   {
      int c, d;
      char string[] = "CProgramming";
      printf("\n\n");
      printf("------------\n");
```

Contd.

```
    for( c = 0 ; c <= 11 ; c++ )
    {
       d = c + 1;
       printf("|%-12.*s|\n", d, string);
    }
    printf("|------------|\n");
    for( c = 11 ; c >= 0 ; c-- )
    {
       d = c + 1;
       printf("|%-12.*s|\n", d, string);
       }
       printf("--------------\n");
    }
```

Output

```
|C           |
|CP          |
|CPr         |
|CPro        |
|CProg       |
|CProgr      |
|CProgra     |
|CProgram    |
|CProgramm   |
|CProgrammi  |
|CProgrammin |
|CProgramming|
|CProgramming|
|CProgrammin |
|CProgrammi  |
|CProgramm   |
|CProgram    |
|CProgra     |
|CProgr      |
|CProg       |
|CPro        |
|CPr         |
|CP          |
|C           |
```

Fig. 9.5 *Illustration of variable field specifications by printing sequences of characters*

```
          C|
         CP|
        CPr|
       CPro|
      CProg|
     CProgr|
    CProgra|
   CProgram|
  CProgramm|
 CProgrammi|
CProgrammin|
```

```
C|
 C|
  C|
   C|
    C|
     C|
      C|
       C|
        C|
         C|
          C|
```

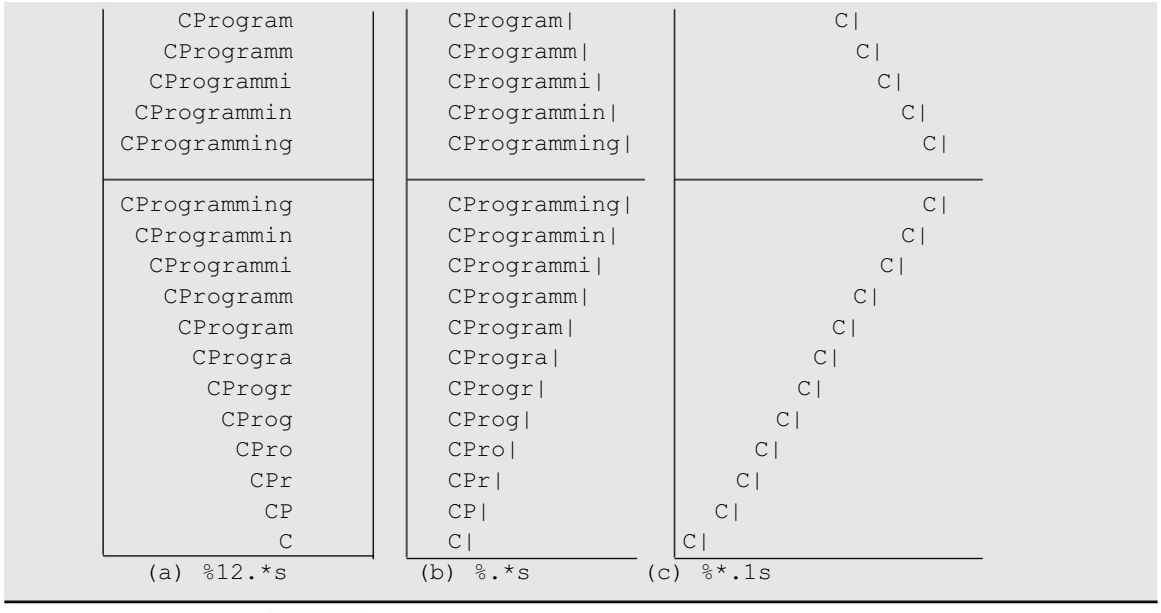


Fig. 9.6 *Further illustrations of variable specifications*

9.4.2 Using *putchar* and *puts* Functions

Like **getchar**, C supports another character handling function **putchar** to output the values of character variables. It takes the following form:

```
char ch = 'A';
putchar (ch);
```

The function **putchar** requires one parameter. This statement is equivalent to:

```
printf("%c", ch);
```

We have used **putchar** function in Chapter 4 to write characters to the screen. We can use this function repeatedly to output a string of characters stored in an array using a loop. Example:

```
char name[6] = "PARIS"
for (i=0, i<5; i++)
     putchar(name[i];
putchar('\n');
```

Another and more convenient way of printing string values is to use the function **puts** declared in the header file <*stdio.h*>. This is a one parameter function and invoked as under:

```
puts ( str );
```

where **str** is a string variable containing a string value. This prints the value of the string variable **str** and then moves the cursor to the beginning of the next line on the screen. For example, the program segment

```
char line [80];
gets (line);
puts (line);
```

reads a line of text from the keyboard and displays it on the screen. Note that the syntax is very simple compared to using the **scanf** and **printf** statements.

9.5 Arithmetic Operations on Characters

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.

To write a character in its integer representation, we may write it as an integer. For example, if the machine uses the ASCII representation, then,

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

will display the number 97 on the screen.

It is also possible to perform arithmetic operations on the character constants and variables. For example,

```
x = 'z'-1;
```

is a valid statement. In ASCII, the value of **'z'** is 122 and therefore, the statement will assign the value 121 to the variable **x.**

We may also use character constants in relational expressions. For example, the expression

```
ch >= 'A' && ch <= 'Z'
```

would test whether the character contained in the variable **ch** is an upper-case letter.

We can convert a character digit to its equivalent integer value using the following relationship:

```
x = character - '0';
```

where **x** is defined as an integer variable and **character** contains the character digit. For example, let us assume that the **character** contains the digit '7',

Then,

x = ASCII value of '7' – ASCII value of '0'
= 55 – 48
= 7

The C library supports a function that converts a string of digits into their integer values. The function takes the form

```
x = atoi(string);
```

x is an integer variable and **string** is a character array containing a string of digits. Consider the following segment of a program:

```
number = "1988";
year = atoi(number);
```

number is a string variable which is assigned the string constant "1988". The function **atoi** converts the string "1988" (contained in **number)** to its numeric equivalent 1988 and assigns it to the integer variable **year.** String conversion functions are stored in the header file <std.lib.h>.

Example 9.6 Write a program which would print the alphabet set a to z and A to Z in decimal and character form.

The program is shown in Fig. 9.7. In ASCII character set, the decimal numbers 65 to 90 represent upper case alphabets and 97 to 122 represent lower case alphabets. The values from 91 to 96 are excluded using an **if** statement in the **for** loop.

```
Program
    main()
   {
   char c;
   printf("\n\n");
   for( c = 65 ; c <= 122 ; c = c + 1 )
   {
   if( c > 90 && c < 97 )
      continue;
   printf("|%4d - %c ", c, c);
   }
   printf("|\n");
   }
Output
   | 65 -   A | 66 - B | 67 - C | 68 - D | 69 - E | 70 - F
   | 71 -   G | 72 - H | 73 - I | 74 - J | 75 - K | 76 - L
   | 77  -  M|  78 - N|  79 - O|  80 - P|  81 - Q|  82 - R
   | 83  -  S|  84 - T|  85 - U|  86 - V|  87 - W|  88 - X
   | 89  -  Y|  90 - Z|  97 - a|  98 - b|  99 - c| 100 - d
   |101  -  e| 102 - f| 103 - g| 104 - h| 105 - i| 106 - j
   |107  -  k| 108 - l| 109 - m| 110 - n| 111 - o| 112 - p
   |113  -  q| 114 - r| 115 - s| 116 - t| 117 - u| 118 - v
   |119  -  w| 120 - x| 121 - y| 122 - z|
```

Fig. 9.7 *Printing of the alphabet set in decimal and character form*

9.6 Putting Strings Together

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;
string2 = string1 + "hello";
```

are not valid. The characters from **string1** and **string2** should be copied into the **string3** one after the other. The size of the array **string3** should be large enough to hold the total characters.

The process of combining two strings together is called *concatenation*. Example 9.7 illustrates the concatenation of three strings.

Example 9.7 The names of employees of an organization are stored in three arrays, namely **first_name**, **second_name**, and **last_name**. Write a program to concatenate the three parts into one string to be called **name**.

The program is given in Fig. 9.8. Three **for** loops are used to copy the three strings. In the first loop, the characters contained in the **first_name** are copied into the variable **name** until the *null* character is reached. The *null* character is not copied; instead it is replaced by a *space* by the assignment statement

```
name[i] = ' ' ;
```

Similarly, the **second_name** is copied into **name**, starting from the column just after the space created by the above statement. This is achieved by the assignment statement

```
name[i+j+1] = second_name[j];
```

If **first_name** contains 4 characters, then the value of **i** at this point will be 4 and therefore the first character from **second_name** will be placed in the *fifth cell* of **name.** Note that we have stored a space in the *fourth cell*.

In the same way, the statement

```
name[i+j+k+2] = last_name[k];
```

is used to copy the characters from **last_name** into the proper locations of **name.**

At the end, we place a null character to terminate the concatenated string **name**. In this example, it is important to note the use of the expressions **i+j+1** and **i+j+k+2.**

Program

```
main()
{
   int i, j, k ;
   char first_name[10] = {"VISWANATH"} ;
   char second_name[10] = {"PRATAP"} ;
   char last_name[10] = {"SINGH"} ;
   char name[30] ;
/* Copy first_name into name */
   for( i = 0 ; first_name[i] != '\0' ; i++ )
      name[i] = first_name[i] ;
/* End first_name with a space */
   name[i] = ' ' ;
/* Copy second_name into name */
   for( j = 0 ; second_name[j] != '\0' ; j++ )
      name[i+j+1] = second_name[j] ;
/* End second_name with a space */
      name[i+j+1] = ' ' ;
/* Copy last_name into name */
   for( k = 0 ; last_name[k] != '\0'; k++ )
      name[i+j+k+2] = last_name[k] ;
/* End name with a null character */
```

Contd

```
        name[i+j+k+2] = '\0' ;
        printf("\n\n") ;
        printf("%s\n", name) ;
    }
Output
    VISWANATH PRATAP SINGH
```

Fig. 9.8 *Concatenation of strings*

9.7 Comparison of Two Strings

Once again, C does not permit the comparison of two strings directly. That is, the statements such as

```
if(name1 == name2)
if(name == "ABC")
```

are not permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character, whichever occurs first. The following segment of a program illustrates this.

```
i=0;
while(str1[i] == str2[i] && str1[i] != '\0'
          && str2[i] != '\0')
   i = i+1;
if (str1[i] == '\0' && str2[i] == '\0')
      printf("strings are equal\n");
   else
      printf("strings are not equal\n");
```

9.8 String-Handling Functions

Fortunately, the C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations discussed so far. Following are the most commonly used string-handling functions.

Function	Action
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

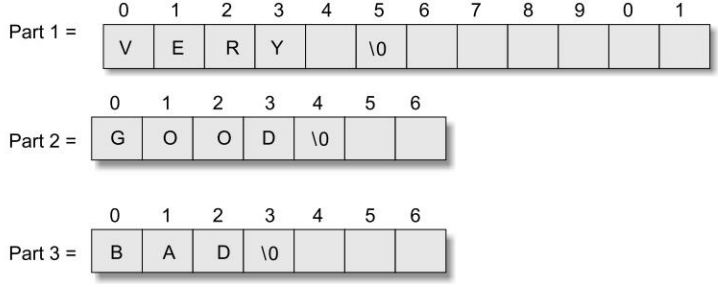
We shall discuss briefly how each of these functions can be used in the processing of strings.

9.8.1 *strcat ()* Function

The **strcat** function joins two strings together. It takes the following form:

```
strcat(string1, string2);
```

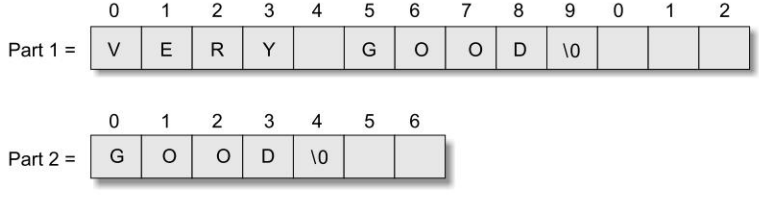
string1 and **string2** are character arrays. When the function **strcat** is executed, **string2** is appended to **string1**. It does so by removing the null character at the end of **string1** and placing **string2** from there. The string at **string2** remains unchanged. For example, consider the following three strings:



Execution of the statement

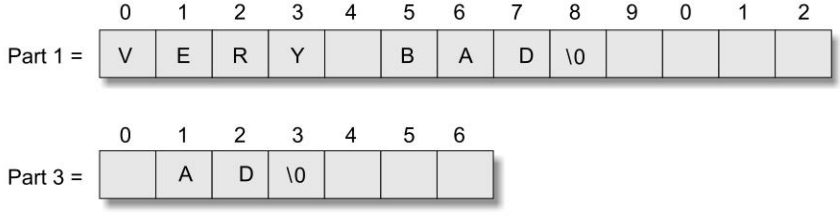
```
strcat(part1, part2);
```

will result in:



while the statement

will result in:



We must make sure that the size of **string1** (to which **string2** is appended) is large enough to accommodate the final string.

strcat function may also append a string constant to a string variable. The following is valid:

```
strcat(part1,"GOOD");
```

C permits nesting of **strcat** functions. For example, the statement

```
strcat(strcat(string1,string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in **string1.**

9.8.2 *strcmp ()* Function

The **strcmp** function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

```
strcmp(string1, string2);
```

string1 and **string2** may be string variables or string constants. Examples are:

```
strcmp(name1, name2);
strcmp(name1, "John");
strcmp("Rom", "Ram");
```

Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

```
strcmp("their", "there");
```

will return a value of –9 which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" in ASCII code is –9. If the value is negative, **string1** is alphabetically above **string2**.

9.8.3 *strcpy ()* Function

The **strcpy** function works almost like a string-assignment operator. It takes the form:

```
strcpy(string1, string2);
```

and assigns the contents of **string2** to **string1**. **string2** may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

will assign the string "DELHI" to the string variable **city.** Similarly, the statement

```
strcpy(city1, city2);
```

will assign the contents of the string variable **city2** to the string variable **city1**. The size of the array **city1** should be large enough to receive the contents of **city2.**

9.8.4 *strlen ()* Function

This function counts and returns the number of characters in a string. It takes the form

```
n = strlen(string);
```

Where **n** is an integer variable, which receives the value of the length of the **string.** The argument may be a string constant. The counting ends at the first null character.

Example 9.8 **s1**, **s2**, and **s3** are three string variables. Write a program to read two string constants into **s1** and **s2** and compare whether they are equal or not. If they are not, join them together. Then copy the contents of **s1** to the variable **s3**. At the end, the program should print the contents of all the three variables and their lengths.

The program is shown in Fig. 9.9. During the first run, the input strings are "New" and "York". These strings are compared by the statement

```
x = strcmp(s1, s2);
```

Since they are not equal, they are joined together and copied into **s3** using the statement

```
strcpy(s3, s1);
```

The program outputs all the three strings with their lengths.

During the second run, the two strings **s1** and **s2** are equal, and therefore, they are not joined together. In this case all the three strings contain the same string constant "London".

Program

```
#include <string.h>
main()
{  char s1[20], s2[20], s3[20];
   int x, l1, l2, l3;
   printf("\n\nEnter two string constants \n");
   printf("?");
   scanf("%s %s", s1, s2);
/* comparing s1 and s2 */
   x = strcmp(s1, s2);
   if(x != 0)
   {     printf("\n\nStrings are not equal \n");
         strcat(s1, s2); /* joining s1 and s2 */
   }
   else
      printf("\n\nStrings are equal \n");
/* copying s1 to s3
   strcpy(s3, s1);
/* Finding length of strings */
   l1 = strlen(s1);
   l2 = strlen(s2);
   l3 = strlen(s3);
/* output */
   printf("\ns1 = %s\t length = %d characters\n", s1, l1);
   printf("s2 = %s\t length = %d characters\n", s2, l2);
   printf("s3 = %s\t length = %d characters\n", s3, l3);
}
```

Output

```
Enter two string constants
? New York
Strings are not equal
s1 = NewYork    length = 7 characters
s2 = York       length = 4 characters
s3 = NewYork    length = 7 characters
```

Contd.

```
Enter two string constants
? London London
Strings are equal
s1 = London length = 6 characters
s2 = London length = 6 characters
s3 = London length = 6 characters
```

Fig. 9.9 *Illustration of string handling functions*

9.8.5 Other String Functions

The header file **<string.h>** contains many more string manipulation functions. They might be useful in certain situations.

strncpy()

In addition to the function **strcpy** that copies one string to another, we have another function **strncpy** that copies only the left-most n characters of the source string to the target string variable. This is a three-parameter function and is invoked as follows:

```
strncpy(s1, s2, 5);
```

This statement copies the first 5 characters of the source string **s2** into the target string **s1.** Since the first 5 characters may not include the terminating null character, we have to place it explicitly in the 6th position of **s2** as shown below:

```
s1[6] ='\0';
```

Now, the string **s1** contains a proper string.

strncmp()

A variation of the function **strcmp** is the function **strncmp**. This function has three parameters as illustrated in the function call below:

```
strncmp (s1, s2, n);
```

this compares the left-most n characters of **s1** to **s2** and returns.

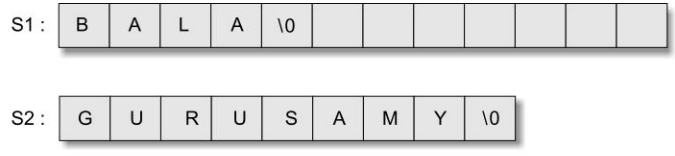
(a) 0 if they are equal;
(b) negative number, if s1 sub-string is less than s2; and
(c) positive number, otherwise.

strncat()

This is another concatenation function that takes three parameters as shown below:

```
strncat (s1, s2, n);
```

This call will concatenate the left-most n characters of **s2** to the end of **s1.** Example:



After **strncat** (s1, s2, s4); execution:



strstr()

It is a two-parameter function that can be used to locate a sub-string in a string. This takes the forms:

```
strstr (s1, s2);
strstr (s1, "ABC");
```

The function **strstr** searches the string **s1** to see whether the string **s2** is contained in **s1**. If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a NULL pointer. Example.

```
if (strstr (s1, s2) == NULL)
   printf("substring is not found");
else
   printf("s2 is a substring of s1");
```

We also have functions to determine the existence of a character in a string. The function call

```
strchr(s1, 'm');
```

will locate the first occurrence of the character 'm' and the call

```
strrchr(s1, 'm');
```

will locate the last occurrence of the character 'm' in the string **s1**.

Warnings

- When allocating space for a string during declaration, remember to count the terminating null character.
- When creating an array to hold a copy of a string variable of unknown size, we can compute the size required using the expression
  strlen (stringname) +1.
- When copying or concatenating one string to another, we must ensure that the target (destination) string has enough space to hold the incoming characters. Remember that no error message will be available even if this condition is not satisfied. The copying may overwrite the memory and the program may fail in an unpredictable way.
- When we use **strncpy** to copy a specific number of characters from a source string, we must ensure to append the null character to the target string, in case the number of characters is less than or equal to the source string.

9.9 Table of Strings

We often use lists of character strings, such as a list of the names of students in a class, list of the names of employees in an organization, list of places, etc. A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list. For example, a character array **student[30][15]** may be used to store a list of 30 names, each of length not more than 15 characters. Shown below is a table of five cities:

C	h	a	n	d	i	g	a	r	h
M	a	d	r	a	s				
A	h	m	e	d	a	b	a	d	
H	y	d	e	r	a	b	a	d	
B	o	m	b	a	y				

This table can be conveniently stored in a character array city by using the following declaration:

```
char city[ ] [ ]
        {
          "Chandigarh",
          "Madras",
          "Ahmedabad",
          "Hyderabad",
          "Bombay"
        } ;
```

To access the name of the ith city in the list, we write

```
city[i-1]
```

and therefore **city[0]** denotes "Chandigarh", **city[1]** denotes "Madras" and so on. This shows that once an array is declared as two-dimensional, it can be used like a one-dimensional array in further manipulations. That is, the table can be treated as a column of strings.

Example 9.9 Write a program that would sort a list of names in alphabetical order.

A program to sort the list of strings in alphabetical order is given in Fig. 9.10. It employs the method of bubble sorting described in Case Study 1 in the previous chapter.

Program

```
#define ITEMS 5
#define MAXCHAR 20
main( )
{
   char string[ITEMS][MAXCHAR], dummy[MAXCHAR];
   int i = 0, j = 0;
   /* Reading the list */
   printf ("Enter names of %d items \n ",ITEMS);
   while (i < ITEMS)
```

Contd.

```
            scanf ("%s", string[i++]);
         /* Sorting begins */
         for (i=1; i < ITEMS; i++) /* Outer loop begins */
         {
            for(j=1; j <= ITEMS-i ; j++) /*Inner loop begins*/
            {
               if (strcmp (string[j-1], string[j]) > 0)
               {  /* Exchange of contents */
                  strcpy (dummy, string[j-1]);
                  strcpy (string[j-1], string[j]);
                  strcpy (string[j], dummy );
               }
            } /* Inner loop ends */
         } /* Outer loop ends */
      /* Sorting completed */
      printf ("\nAlphabetical list \n\n");
      for (i=0; i < ITEMS ; i++)
         printf ("%s", string[i]);
   }
Output
   Enter names of 5 items
   London Manchester Delhi Paris Moscow
   Alphabetical list
   Delhi
   London
   Manchester
   Moscow
   Paris
```

Fig. 9.10 *Sorting of strings in alphabetical order*

Note that a two-dimensional array is used to store the list of strings. Each string is read using a **scanf** function with **%s** format. Remember, if any string contains a white space, then the part of the string after the white space will be treated as another item in the list by the **scanf.** In such cases, we should read the entire line as a string using a suitable algorithm. For example, we can use **gets** function to read a line of text containing a series of words. We may also use **puts** function in place of **scanf** for output.

9.10 Other Features of Strings

Other aspects of strings we have not discussed in this chapter include:

- Manipulating strings using pointers.
- Using string as function parameters.
- Declaring and defining strings as members of structures.

These topics will be dealt with later when we discuss functions, structures and pointers.

Just Remember

- Character constants are enclosed in single quotes and string constants are enclosed in double quotes.
- Allocate sufficient space in a character array to hold the null character at the end.
- Avoid processing single characters as strings.
- Using the address operator **&** with a **string** variable in the **scanf** function call is an error.
- It is a compile time error to assign a string to a character variable.
- Using a string variable name on the left of the assignment operator is illegal.
- When accessing individual characters in a string variable, it is logical error to access outside the array bounds.
- Strings cannot be manipulated with operators. Use string functions.
- Do not use string functions on an array **char** type that is not terminated with the null character.
- Do not forget to append the null character to the target string when the number of characters copied is less than or equal to the source string.
- Be aware the return values when using the functions **strcmp** and **strncmp** for comparing strings.
- When using string functions for copying and concatenating strings, make sure that the target string has enough space to store the resulting string. Otherwise memory overwriting may occur.
- The header file <stdio.h> is required when using standard I/O functions.
- The header file <ctype.h> is required when using character handling functions.
- The header file <stdlib.h> is required when using general utility functions.
- The header file <string.h> is required when using string manipulation functions.

Review Questions

9.1 State whether the following statements are *true* or *false.*

(a) When initializing a string variable during its declaration, we must include the null character as part of the string constant, like "GOOD\0".

(b) The **gets** function automatically appends the null character at the end of the string read from the keyboard.

(c) When reading a string with **scanf**, it automatically inserts the terminating null character.

(d) String variables cannot be used with the assignment operator.

(e) We cannot perform arithmetic operations on character variables.

(f) We can assign a character constant or a character variable to an **int** type variable.

(g) The function **scanf** cannot be used in any way to read a line of text with the white-spaces.

(h) The ASCII character set consists of 128 distinct characters.

(i) In the ASCII collating sequence, the uppercase letters precede lowercase letters.

(j) In C, it is illegal to mix character data with numeric data in arithmetic operations.

(k) The function **getchar** skips white-space during input.

(l) In C, strings cannot be initialized at run time.

(m) The input function **gets** has one string parameter.
(n) The function call **strcpy(s2, s1);** copies string s2 into string s1.
(o) The function call **strcmp("abc", "ABC");** returns a positive number.

9.2 Fill in the blanks in the following statements.
(a) We can use the conversion specification _____ in **scanf** to read a line of text.
(b) We can initialize a string using the string manipulation function _____.
(c) The function **strncat** has _____ parameters.
(d) To use the function **atoi** in a program, we must include the header file _____.
(e) The function _____ does not require any conversion specification to read a string from the keyboard.
(f) The function _____ is used to determine the length of a string.
(g) The _____ string manipulation function determines if a character is contained in a string.
(h) The function _____ is used to sort the strings in alphabetical order.
(i) The function call **strcat (s2, s1);** appends _____ to _____.
(j) The **printf** may be replaced by _____ function for printing strings.

9.3 Describe the limitations of using **getchar** and **scanf** functions for reading strings.

9.4 Character strings in C are automatically terminated by the *null* character. Explain how this feature helps in string manipulations.

9.5 Strings can be assigned values as follows:

(a) During type declaration	char string[ ] = {"....."};
(b) Using **strcpy** function	strcpy(string, ".....");
(c) Reading using **scanf** function	scanf("%s", string);
(d) Reading using **gets** function	gets(string);

Compare them critically and describe situations where one is superior to the others.

9.6 Assuming the variable **string** contains the value "The sky is the limit.", determine what output of the following program segments will be.

(a)
```
printf("%s", string);
```
(b)
```
printf("%25.10s", string);
```
(c)
```
printf("%s", string[0]);
```
(d)
```
for (i=0; string[i] != "."; i++)
        printf("%c", string[i]);
```
(e)
```
for (i=0; string[i] != '\0'; i++;)
printf("%d\n", string[i]);
```
(f)
```
for (i=0; i <= strlen[string]; ;)
{
        string[i++] = i;
        printf("%s\n", string[i]);
}
```
(g)
```
printf("%c\n", string[10] + 5);
```
(h)
```
printf("%c\n", string[10] + 5')
```

9.7 Which of the following statements will correctly store the concatenation of strings **s1** and **s2** in string **s3**?

(a) s3 = strcat (s1, s2);
(b) strcat (s1, s2, s3);
(c) strcat (s3, s2, s1);
(d) strcpy (s3, strcat (s1, s2));
(e) strcmp (s3, strcat (s1, s2));
(f) strcpy (strcat (s1, s2), s3);

9.8 What will be the output of the following statement?

```
printf ("%d", strcmp ("push", "pull"));
```

9.9 Assume that s1, s2 and s3 are declared as follows:

```
char s1[10] = "he", s2[20] = "she", s3[30], s4[30];
```

What will be the output of the following statements executed in sequence?

```
printf("%s", strcpy(s3, s1));
printf("%s", strcat(strcat(strcpy(s4, s1), "or"), s2));
printf("%d %d", strlen(s2)+strlen(s3), strlen(s4));
```

9.10 Find errors, if any, in the following code segments;

(a) char str[10]
strncpy(str, "GOD", 3);
printf("%s", str);

(b) char str[10];
strcpy(str, "Balagurusamy");

(c) if strstr("Balagurusamy", "guru") == 0);
printf("Substring is found");

(d) char s1[5], s2[10],
gets(s1, s2);

9.11 What will be the output of the following segment?

```
char s1[ ] = "Kolkotta" ;
char s2[ ] = "Pune" ;
strcpy (s1, s2) ;
printf("%s", s1) ;
```

9.12 What will be the output of the following segment?

```
char s1[ ] = "NEW DELHI" ;
char s2[ ] = "BANGALORE" ;
strncpy (s1, s2, 3) ;
printf("%s", s1) ;
```

9.13 What will be the output of the following code?

```
char s1[ ] = "Jabalpur" ;
char s2[ ] = "Jaipur" ;
printf(strncmp(s1, s2, 2) );
```

9.14 What will be the output of the following code?

```
char s1[ ] = "ANIL KUMAR GUPTA";
char s2[ ] = "KUMAR";
printf (strstr (s1, s2) );
```

9.15 Compare the working of the following functions:

(a) strcpy and strncpy;
(b) strcat and strncat; and
(c) strcmp and strncmp.

Programming Exercises

9.1 Write a program, which reads your name from the keyboard and outputs a list of ASCII codes, which represent your name.

9.2 Write a program to do the following:

(a) To output the question "Who is the inventor of C ?"
(b) To accept an answer.
(c) To print out "Good" and then stop, if the answer is correct.
(d) To output the message 'try again', if the answer is wrong.
(e) To display the correct answer when the answer is wrong even at the third attempt and stop.

9.3 Write a program to extract a portion of a character string and print the extracted string. Assume that m characters are extracted, starting with the nth character.

9.4 Write a program which will read a text and count all occurrences of a particular word.

9.5 Write a program which will read a string and rewrite it in the alphabetical order. For example, the word STRING should be written as GINRST.

9.6 Write a program to replace a particular word by another word in a given string. For example, the word "PASCAL" should be replaced by "C" in the text "It is good to program in PASCAL language."

9.7 A Maruti car dealer maintains a record of sales of various vehicles in the following form:

Vehicle type	Month of sales	Price
MARUTI-800	02/01	210000
MARUTI-DX	07/01	265000
GYPSY	04/02	315750
MARUTI-VAN	08/02	240000

Write a program to read this data into a table of strings and output the details of a particular vehicle sold during a specified period. The program should request the user to input the vehicle type and the period (starting month, ending month).

9.8 Write a program that reads a string from the keyboard and determines whether the string is a *palindrome* or not. (A string is a palindrome if it can be read from left and right with the same meaning. For example, Madam and Anna are palindrome strings. Ignore capitalization).

9.9 Write program that reads the cost of an item in the form RRRR.PP (Where RRRR denotes Rupees and PP denotes Paise) and converts the value to a string of words that expresses the numeric value in words. For example, if we input 125.75, the output should be "ONE HUNDRED TWENTY FIVE AND PAISE SEVENTY FIVE".

9.10 Develop a program that will read and store the details of a list of students in the format

Roll No.	**Name**	**Marks obtained**
. . . . .	. . . . .	. . . . .
. . . .	. . . . .	. . . .
. . . . .	. .. .	. . ..

and produce the following output lits:

(a) Alphabetical list of names, roll numbers and marks obtained.

(b) List sorted on roll numbers.

(c) List sorted on marks (rank-wise list)

9.11 Write a program to read two strings and compare them using the function **strncmp( )** and print a message that the first string is equal, less, or greater than the second one.

9.12 Write a program to read a line of text from the keyboard and print out the number of occurrences of a given substring using the function **strstr ( ).**

9.13 Write a program that will copy m consecutive characters from a string s1 beginning at position n into another string s2.

9.14 Write a program to create a directory of students with roll numbers. The program should display the roll number for a specified name and vice-versa.

9.15 Given a string

char str [ ] = "123456789" ;

Write a program that displays the following:

```
    1
  2 3 2
 3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

Debugging Exercises

9.1 Identify the error in the following program, if any:

```
#include <stdio.h>
void main( )
{
 char str[50];
 clrscr();
 printf("Enter a string:\n");
 gets(&str);
 puts("You just entered the string: ");
 puts(str);
}
```

9.2 Identify the error in the following program that determines whether a string is a palindrome or not:

```
/*Program to check whether a string is palindrome or not*/
#include <stdio.h>
#include <string.h>
void main()
{
 char chk='t', str[30];
 int len,left,right;
 printf("\nEnter a string: ");
 scanf("%s", &str); /*Reading the input string*/
 len=strlen(str);
 left=0;
 right=len;
  while(left < right && chk=='t')
 {
  if(str[left]==str[right]) /*Comparing the respective characters on
the left and right side of the string*/
    ;
  else
   chk='f';
  left++;
  right--;
 }
 /*Displaying the result*/
 if(chk=='t')
  printf("\nThe sting %s is a palindrome",str);
 else
  printf("\nThe sting %s is not a palindrome",str);

 }
```

9.3 Identify the error in the following program, if any:

```
#include <stdio.h>

void main( )
{
```

```
 char str[50];

 printf("Enter a string:\n");
 gets(str);

printf("\nThe given string is: %s",str);
 printf("\n\nAfter converting all characters into uppercase, the
string becomes: \n\n%s",toupr(str));
}
```

9.4 Identify the error in the following program that reverses the input string:

```
#include <stdio.h>
#include <string.h>
void main()
{
 char str[50],revstr[40];
 int i,len;
 printf("\nEnter a string: ");
 gets(str);
 /*Finding the reverse of the string*/
 len=strlen(str);
 for(i=0;i<len;i++)
  revstr[len-i-1]=str[i];
 printf("\n\nThe reverse of string %s is %s",str,revstr); /*Displaying
the result*/
  getch();
  }
```

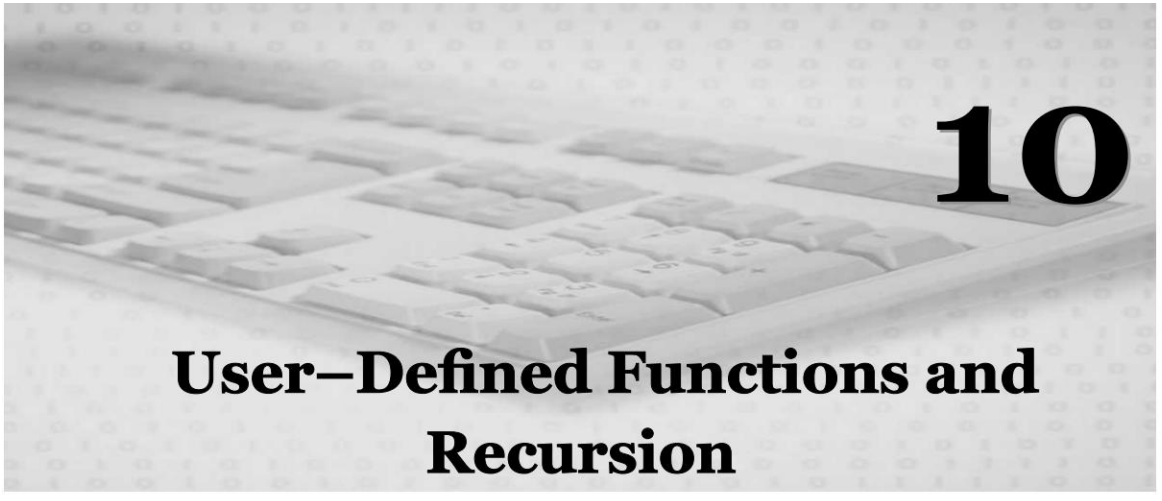
9.5 Identify the error in the following program that concatenates two strings:

```
#include <stdio.h>
#include <string.h>
void main( )
{
 char str1[50], str2[30];
 printf("Enter string 1:\n");
 gets(str1);
 printf("Enter string2:\n");
 gets(str2);
  printf("The result of concatenating strings %s and %s is:
",str1,str2);
```

```
/*Using the strcat function to concatenate the two strings*/
 printf("%s",strcat(str1,str2));
 getch();
}
```

9.6 Identify the error in the following program that counts the number of vowels and consonants in a string:

```
#include <stdio.h>
#include <string.h>
void main()
{
 char str[30];
 int vow=0,cons=0,i=0;
 printf("Enter a string: ");
 gets(str);
 while(str[i] != NULL)
 {
  if(str[i] == 'a' || str[i] == 'A' || str[i] == 'e' || str[i] == 'E'
|| str[i] == 'i' || str[i] == 'I' || str[i] == 'o' || str[i] == 'O'
|| str[i] == 'u' || str[i] == 'U') /*Checking for vowels*/
   vow++;
  else
   cons++;
  i++;
 }
 /*Displaying the result*/
 printf("\nNumber of Vowels = %d",vow);
 printf("\nNumber of Consonents = %d",cons);
}
```



10

User–Defined Functions and Recursion

10.1 Introduction

We have mentioned earlier that one of the strengths of C language is C functions. They are easy to define and use. We have used functions in every program that we have discussed so far. However, they have been primarily limited to the three functions, namely, **main, printf**, and **scanf**. In this chapter, we shall consider in detail the following:

- How a function is designed?
- How a function is integrated into a program?
- How two or more functions are put together? and
- How they communicate with one another?

C functions can be classified into two categories, namely, *library* functions and *user-defined* functions. **main** is an example of user-defined functions. **printf** and **scanf** belong to the category of library functions. Other library functions are **sqrt, cos, strcat,** etc. The main distinction between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library. In fact, this is one of the strengths of C language.

10.2 Need for User-defined Functions

As pointed out earlier, **main** is a specially recognized function in C. Every program must have a **main** function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only **main** function, it leads to a number of problems. The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult. If a

program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These independently coded programs are called *subprograms* that are much easier to understand, debug, and test. In C, such subprograms are referred to as **'functions'**.

There are times when certain type of operations or calculations are repeated at many points throughout a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements wherever they are needed. Another approach is to design a function that can be called and used whenever required. This saves both time and space.

This "division" approach clearly results in a number of advantages.

1. It facilitates top-down modular programming as shown in Fig. 10.1. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what others have already done, instead of starting all over again from scratch.

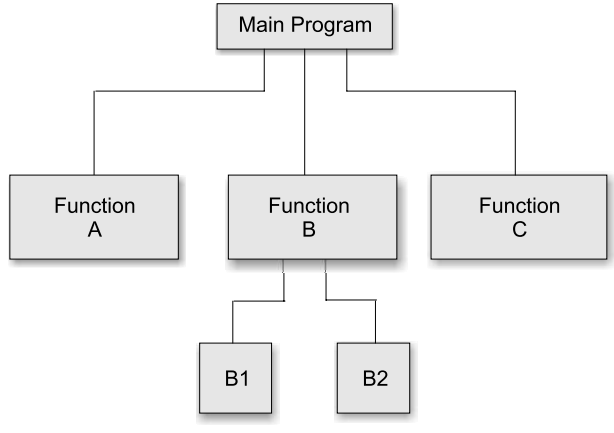


Fig. 10.1 *Top-down modular programming using functions*

10.3 A Multi-function Program

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. All that the program knows about a function is: What goes in and what comes out. Every C program can be designed using a collection of these black boxes known as *functions*.

Consider a set of statements as shown below:

```
void printline(void)
{
   int i;
   for (i=1; i<40; i++)
         printf("-");
   printf("\n");
}
```

The above set of statements defines a function called **printline,** which could print a line of 39-character length. This function can be used in a program as follows:

```
void printline(void); /* declaration  */
 main( )
 {
      printline( );
      printf("This illustrates the use of C functions\n");
      printline();
 }
   void printline(void)
 {
 int i;
 for(i=1; i<40; i++)
 printf("-");
 printf("\n");
 }
```

This program will print the following output:

```
This illustrates the use of C functions
```

The above program contains two user-defined functions:

main() function
printline() function

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

printline( );

which indicates that the function **printline** is to be executed. At this point, the program control is transferred to the function **printline.** After executing the **printline** function, which outputs a line of 39 character length, the control is transferred back to the **main.** Now, the execution continues at the point where the function call was executed. After executing the **printf** statement, the control is again transferred to the **printline** function for printing the line once more.

The **main** function calls the user-defined **printline** function two times and the library function **printf** once. We may notice that the **printline** function itself calls the library function **printf** 39 times repeatedly.

Any function can call any other function. In fact, it can call itself. A 'called function' can also call another function. A function can be called more than once. In fact, this is one of the main features of using functions. Figure 10.2 illustrates the flow of control in a multi-function program.

Except the starting point, there are no other predetermined relationships, rules of precedence, or hierarchies among the functions that make up a complete program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end. See the box "Modular Programming"

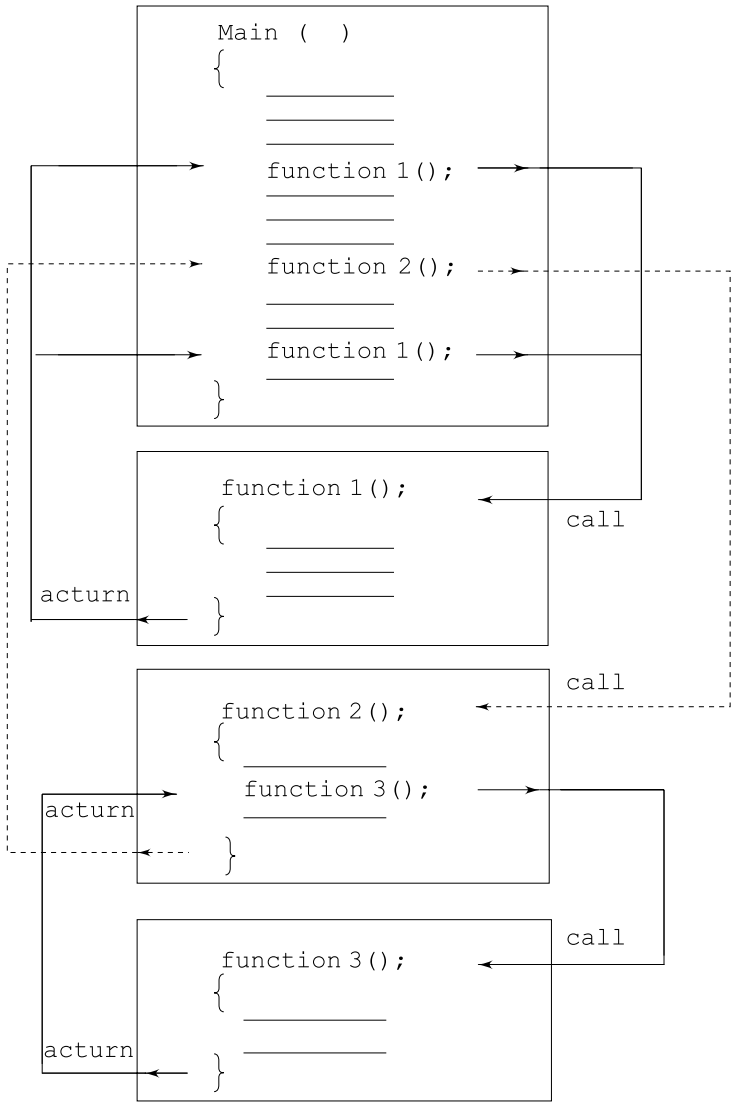


Fig. 10.2 *Flow of control in a multi-function program*

Modular Programming

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called ***modules*** that

Contd

are separately named and individually callable *program units*. These modules are carefully integrated to become a software system that satisfies the system requirements. It is basically a "divide-and-conquer" approach to problem solving.

Modules are identified and designed such that they can be organized into a top-down hierarchical structure (similar to an organization chart). In C, each module refers to a function that is responsible for a single task.

Some characteristics of modular programming are:

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module.
4. No communication can take place directly between modules that do not have calling-called relationship.
5. All modules are designed as *single-entry, single-exit* systems using control structures.

10.4 Elements of User-defined Functions

We have discussed and used a variety of data types and variables in our programs so far. However, declaration and use of these variables were primarily done inside the **main** function. As we mentioned in Chapter 5, functions are classified as one of the derived data types in C. We can therefore define functions and use them like any other variables in C programs. It is therefore not a surprise to note that there exist some similarities between functions and variables in C.

- Both function names and variable names are considered identifiers and therefore they must adhere to the rules for identifiers.
- Like variables, functions have types (such as int) associated with them.
- Like variables, function names and their types must be declared and defined before they are used in a program.

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition.
2. Function call.
3. Function declaration.

The *function definition* is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required place in the program. This is known as the *function call.* The program (or a function) that calls the function is referred to as the *calling program* or *calling function.* The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as the *function declaration* or *function prototype.*

10.5 Definition of Functions

A *function definition,* also known as *function implementation* shall include the following elements;

1. function name;
2. function type;

3. list of parameters;
4. local variable declarations;
5. function statements; and
6. a return statement.

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

- function header (First three elements); and
- function body (Second three elements).

A general format of a function definition to implement these two parts is given below:

```
function_type function_name(parameter list)
{
  local variable declaration;
  executable statement1;
  executable statement2;
  . . . . .
  . . . . .
  return statement;
}
```

The first line **function_type function_name(parameter list)** is known as the *function header* and the statements within the opening and closing braces constitute the *function body,* which is a compound statement.

10.5.1 Function Header

The function header consists of three parts: the function type (also known as *return* type), the function name and the *formal* parameter list. Note that a semicolon is not used at the end of the function header.

10.5.2 Name and Type

The *function type* specifies the type of value (*like float or double*) that the function is expected to return to the program calling the function. If the return type is not explicitly specified, C will assume that it is an integer type. If the function is not returning anything, then we need to specify the return type as **void**. Remember, **void** is one of the fundamental data types in C. It is a good programming practice to code explicitly the return type, even when it is an integer. The value returned is the output produced by the function.

The *function name* is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function. However, care must be exercised to avoid duplicating library routine names or operating system commands.

10.5.3 Formal Parameter List

The *parameter list* declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as *formal* parameters. These parameters can also be used to send values to the calling programs. This aspect will be covered later when we discuss more about functions. The parameters are also known as *arguments*.

The parameter list contains declaration of variables separated by commas and surrounded by parentheses. Examples:

float quadratic (int a, int b, int c) {. . . . }
double power (double x, int n) {. . . ..}
float mul (float x, float y) {. . . . }
int sum (int a, int b) {. . . . }

Remember, there is no semicolon after the closing parenthesis. Note that the declaration of parameter variables cannot be combined. That is, **int sum (int a,b)** is illegal.

A function need not always receive values from the calling program. In such cases, functions have no formal parameters. To indicate that the parameter list is empty, we use the keyword **void** between the parentheses as in

```
void printline (void)
{
        ....
}
```

This function neither receives any input values nor returns back any value. Many compilers accept an empty set of parentheses, without specifying anything as in

void printline ( )

But, it is a good programming style to use **void** to indicate a nill parameter list.

10.5.4 Function Body

The *function body* contains the declarations and statements necessary for performing the required task. The body enclosed in braces, contains three parts, in the order given below:

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. A **return** statement that returns the value evaluated by the function.

If a function does not return any value (like the **printline** function), we can omit the **return** statement. However, note that its return type should be specified as **void.** Again, it is nice to have a return statement even for **void** functions.

Some examples of typical function definitions are:

```
(a)   float mul (float x, float y)
      {
         float result;                 /* local variable */
         result = x * y;               /* computes the product */
         return (result);              /* returns the result */
      }
(b)   void sum (int a, int b)
      {
         printf ("sum = %s", a + b); /* no local variables */
         return;                    /* optional */
      }
```

Contd.

```
(c)   void display (void)
         {                                  /* no local variables */
            printf ("No type, no parameters");
                                            /* no return statement */
         }
```

Note

1. *When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a void return.*
2. *A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.*

10.6 Return Values and Their Types

As pointed out earlier, a function may or may not send back any value to the calling function. If it does, it is done through the **return** statement. While it is possible to pass to the called function any number of values, the called function can only return *one value* per call, at the most.

The **return** statement can take one of the following forms:

```
return;
or
return(expression);
```

The first, the 'plain' **return** does not return any value; it acts much as the closing brace of the function. When a **return** is encountered, the control is immediately passed back to the calling function. An example of the use of a simple **return** is as follows:

```
if(error)
return;
```

Note *In C99, if a function is specified as returning a value, the **return** must have value associated with it.*

The second form of **return** with an expression returns the value of the expression. For example, the function

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

returns the value of **p** which is the product of the values of **x** and **y.** The last two statements can be combined into one statement as follows:

```
return (x*y);
```

A function may have more than one **return** statements. This situation arises when the value returned is based on certain conditions. For example:

```
if( x <= 0 )
  return(0);
```

```
else
  return(1);
```

What type of data does a function return? All functions by default return **int** type data. But what happens if a function must return some other type? We can force a function to return a particular type of data by using a *type specifier* in the function header as discussed earlier.

When a value is returned, it is automatically cast to the function's type. In functions that do computations using **doubles**, yet return **ints**, the returned value will be truncated to an integer. For instance, the function

```
int product (void)
{
    return (2.5 * 3.0);
}
```

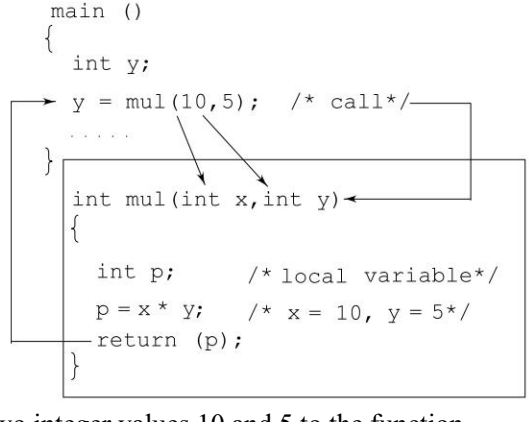
will return the value 7, only the integer part of the result.

10.7 Function Calls

A function can be called by simply using the function name followed by a list of *actual parameters* (or arguments), if any, enclosed in parentheses. Example:

```
main( )
{
  int y;
  y = mul(10,5);      /* Function call */
  printf("%d\n", y);
}
```

When the compiler encounters a function call, the control is transferred to the function **mul()**. This function is then executed line by line as described and a value is returned when a **return** statement is encountered. This value is assigned to **y**. This is illustrated below:



The function call sends two integer values 10 and 5 to the function.

int mul(int x, int y)

which are assigned to **x** and **y** respectively. The function computes the product **x** and **y**, assigns the result to the local variable **p**, and then returns the value 25 to the **main** where it is assigned to **y** again.

There are many different ways to call a function. Listed below are some of the ways the function **mul** can be invoked.

mul (10, 5)
mul (m, 5)
mul (10, n)
mul (m, n)
mul (m + 5, 10)
mul (10, mul(m,n))
mul (expression1, expression2)

Note that the sixth call uses its own call as its one of the parameters. When we use expressions, they should be evaluated to single values that can be passed as actual parameters.

A function which returns a value can be used in expressions like any other variable. Each of the following statements is valid:

```
printf("%d\n", mul(p,q));
y = mul(p,q) / (p+q);
if (mul(m,n)>total) printf("large");
```

However, a function cannot be used on the right side of an assignment statement. For instance,

```
mul(a,b) = 15;
```

is invalid.

A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function. The function **printline( )** discussed in Section 10.3 belongs to this category. Such functions may be called in by simply stating their names as independent statements.

Example:

```
main( )
{
  printline( );
}
```

Note the presence of a semicolon at the end.

Function Call

A function call is a postfix expression. The operator (. .) is at a very high level of precedence. (See Table 4.8) Therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.

In a function call, the function name is the operand and the parentheses set (. .) which contains the *actual parameters* is the operator. The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

Note

1. *If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.*
2. *On the other hand, if the actuals are less than the formals, the unmatched formal arguments will be initialized to some garbage.*
3. *Any mismatch in data types may also result in some garbage values.*

10.8 Function Declaration

Like variables, all functions in a C program must be declared, before they are invoked. A *function declaration* (also known as *function prototype*) consists of four parts.

- Function type (return type).
- Function name.
- Parameter list.
- Terminating semicolon.

They are coded in the following format:

```
Function-type function-name (parameter list);
```

This is very similar to the function header line except the terminating semicolon. For example, **mul** function defined in the previous section will be declared as:

```
int mul (int m, int n); /* Function prototype */
```

Points to note:

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns **int** type data.
7. The retype must be **void** if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

Equally acceptable forms of declaration of **mul** function are:

```
int  mul  (int, int);
     mul  (int a, int b);
     mul  (int, int);
```

When a function does not take any parameters and does not return any value, its prototype is written as:

```
void display (void);
```

A prototype declaration may be placed in two places in a program.

1. Above all the functions (including **main**).
2. Inside a function definition.

When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a *global prototype*. Such declarations are available for all the functions in the program.

When we place it in a function definition (in the local declaration section), the prototype is called a *local prototype*. Such declarations are primarily used by the functions containing them.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the *scope* of the function. (Scope is discussed

later in this chapter.) It is a good programming style to declare prototypes in the global declaration section before **main**. It adds flexibility, provides an excellent quick reference to the functions used in the program, and enhances documentation.

Prototypes: Yes or No

Prototype declarations are not essential. If a function has not been declared before it is used, C will assume that its details available at the time of linking. Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definitions. If these assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations, preferably in global declaration section.

Parameters Everywhere!

Parameters (also known as arguments) are used in three places:

1. in declaration (prototypes),
2. in function call, and
3. in function definition.

The parameters used in prototypes and function definitions are called *formal parameters* and those used in function calls are called *actual parameters*. Actual parameters used in a calling statement may be simple constants, variables or expressions.

The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

10.9 Category of Functions

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

Category 1: Functions with no arguments and no return values.
Category 2: Functions with arguments and no return values.
Category 3: Functions with arguments and one return value.
Category 4: Functions with no arguments but return a value.
Category 5: Functions that return multiple values.

In the sections to follow, we shall discuss these categories with examples. Note that, from now on, we shall use the term arguments (rather than parameters) more frequently:

10.10 No Arguments and no Return Values

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function. This is depicted in Fig. 10.3. The dotted lines indicate that there is only a transfer of control but not data.

As pointed out earlier, a function that does not return any value cannot be used in an expression. It can only be used as an independent statement.

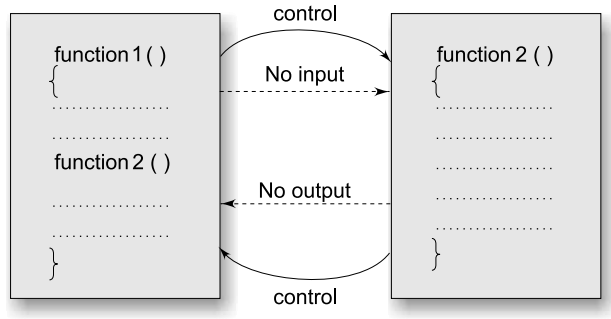


Fig. 10.3 *No data communication between functions*

Example 10.1 Write a program with multiple functions that do not communicate any data between them.

A program with three user-defined functions is given in Fig. 10.4. **main** is the calling function that calls **printline** and **value** functions. Since both the called functions contain no arguments, there are no argument declarations. The **printline** function, when encountered, prints a line with a length of 35 characters as prescribed in the function. The **value** function calculates the value of principal amount after a certain period of years and prints the results. The following equation is evaluated repeatedly:

value = principal(1+interest-rate)

```
Program
     /* Function declaration */
     void printline (void);
     void value (void);
  main()
  {
     printline();
     value();
     printline();
  }
  /* Function1: printline( ) */
  void printline(void) /* contains no arguments */
  {
     int i ;
     for(i=1; i <= 35; i++)
         printf("%c",'-');
     printf("\n");
  }
  /*    Function2: value( ) */
  void value(void)       /* contains no arguments */
  {
     int year, period;
     float inrate, sum, principal;
```

Contd

```
  printf("Principal amount?");
     scanf("%f", &principal);
     printf("Interest rate? ");
     scanf("%f", &inrate);
     printf("Period? ");
     scanf("%d", &period);

  sum = principal;
     year = 1;
     while(year <= period)
     {
        sum = sum *(1+inrate);
        year = year +1;
     }
     printf("\n%8.2f %5.2f %5d %12.2f\n",
         principal,inrate,period,sum);
  }
```

Output

```
  - - - - - - - - - - - - - - - - - - - - -
  Principal amount?     5000
  Interest rate?        0.12
  Period?               5
  5000.00 0.12          5     8811.71
  - - - - - - - - - - - - - - - - - - - - -
```

Fig. 10.4 *Functions with no arguments and no return values*

It is important to note that the function **value** receives its data directly from the terminal. The input data include principal amount, interest rate and the period for which the final value is to be calculated. The **while** loop calculates the final value and the results are printed by the library function **printf.** When the closing brace of **value( )** is reached, the control is transferred back to the calling function **main.** Since everything is done by the value itself there is in fact nothing left to be sent back to the called function. Return types of both **printline** and **value** are declared as **void.**

Note that no **return** statement is employed. When there is nothing to be returned, the **return** statement is optional. The closing brace of the function signals the end of execution of the function, thus returning the control, back to the calling function.

10.11 Arguments but no Return Values

In Fig. 10.4 the **main** function has no control over the way the functions receive input data. For example, the function **printline** will print the same line each time it is called. Same is the case with the function **value.** We could make the calling function to read data from the terminal and pass it on to the called function. This approach seems to be wiser because the calling function can check for the validity of data, if necessary, before it is handed over to the called function.

The nature of data communication between the *calling function* and the *called function* with

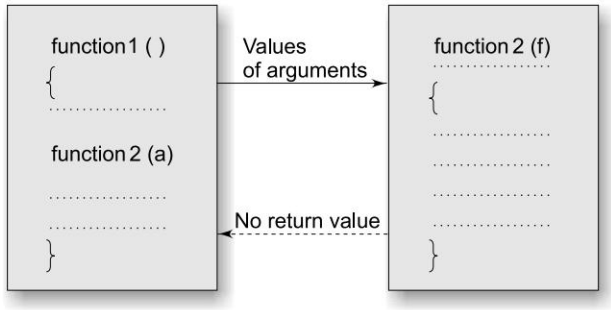


Fig. 10.5 *One-way data communication*

We shall modify the definitions of both the called functions to include arguments as follows:

void printline(char ch)

void value(float p, float r, int n)

The arguments **ch, p, r,** and **n** are called the *formal arguments.* The calling function can now send values to these arguments using function calls containing appropriate arguments. For example, the function call

value(500,0.12,5)

would send the values 500, 0.12 and 5 to the function

void value(float p, float r, int n)

and assign 500 to **p**, 0.12 to **r** and 5 to **n**. The values 500, 0.12 and 5 are the *actual arguments*, which become the values of the *formal arguments* inside the called function.

The *actual* and *formal* arguments should match in number, type, and order. The values of actual arguments are assigned to the formal arguments on a *one to one* basis, starting with the first argument as shown in Fig. 10.6.

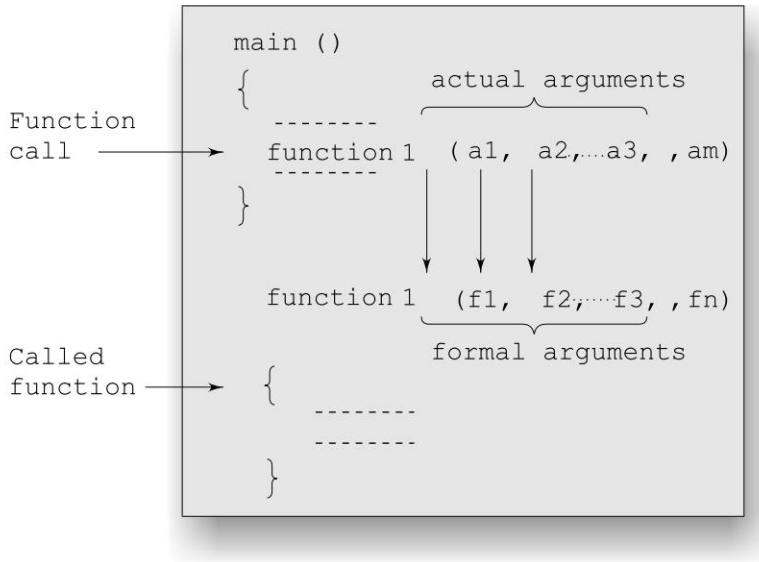


Fig. 10.6 *Arguments matching between the function call and the called function*

We should ensure that the function call has matching arguments. In case, the actual arguments are more than the formal arguments (m > n), the extra actual arguments are discarded. On the other hand, if the actual arguments are less than the formal arguments, the unmatched formal arguments are initialized to some garbage values. Any mismatch in data type may also result in passing of garbage values. Remember, no error message will be generated.

While the formal arguments must be valid variable names, the actual arguments may be variable names, expressions, or constants. The variables used in actual arguments must be assigned values before the function call is made.

Remember that, when a function call is made, only *a copy of the values of actual arguments is passed into the called function.* What occurs inside the function will have no effect on the variables used in the actual argument list.

Example 10.2 Modify the program of Example 10.1 to include the arguments in the function calls.

The modified program with function arguments is presented in Fig. 10.7. Most of the program is identical to the program in Fig. 10.4. The input prompt and **scanf** assignment statement have been moved from **value** function to **main.** The variables **principal, inrate,** and **period** are declared in **main** because they are used in main to receive data. The function call

```
value(principal, inrate, period);
```

passes information it contains to the function **value.**

The function header of **value** has three formal arguments **p,r,** and **n** which correspond to the actual arguments in the function call, namely, **principal, inrate,** and **period.** On execution of the function call, the values of the actual arguments are assigned to the corresponding formal arguments. In fact, the following assignments are accomplished across the function boundaries:

```
p = principal;
r = inrate;
n = period;
```

Program

```
/* prototypes */
void printline (char c);
void value (float, float, int);
main( )
{
      float principal, inrate;
      int period;

      printf("Enter principal amount, interest");
      printf(" rate, and period \n");
      scanf("%f %f %d",&principal, &inrate, &period);
      printline('Z');
      value(principal,inrate,period);
      printline('C');
}
void printline(char ch)
{
```

Contd.

```
            int i ;
            for(i=1; i <= 52; i++)
                  printf("%c",ch);
            printf("\n");
      }
      void value(float p, float r, int n)
      {
            int year ;
            float sum ;
            sum = p ;
            year = 1;
            while(year <= n)
         {
               sum = sum * (1+r);
               year = year +1;
         }
               printf("%f\t%f\t%d\t%f\n",p,r,n,sum);
   }
Output
   Enter principal amount, interest rate, and period
   5000 0.12 5
   ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
   5000.000000           0.120000 5          8811.708984
   CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Fig. 10.7 *Functions with arguments but no return values*

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

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

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

Variable Number of Arguments

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

double area(float d,…)

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

10.12 Arguments with Return Values

The function **value** in Fig. 10.7 receives data from the calling function through arguments, but does not send back any value. Rather, it displays the results of calculations at the terminal. However, we may not always wish to have the result of a function displayed. We may use it in the calling function for further processing. Moreover, to assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations. For example, different programs may require different output formats for display of results. These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such functions will have two-way data communication as shown in Fig. 10.8.

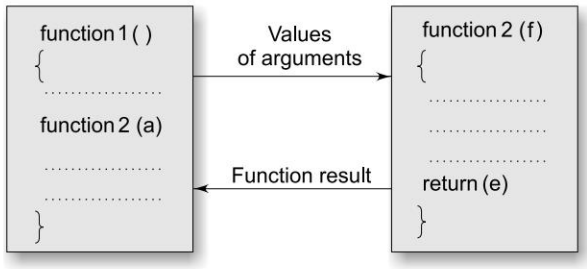


Fig. 10.8 *Two-way data communication between functions.*

We shall modify the program in Fig. 10.7 to illustrate the use of two-way data communication between the *calling* and the *called functions*.

Example 10.3 In the program presented in Fig. 10.7 modify the function **value**, to return the final amount calculated to the **main**, which will display the required output at the terminal. Also extend the versatility of the function **printline** by having it to take the length of the line as an argument.

The modified program with the proposed changes is presented in Fig. 10.9. One major change is the movement of the **printf** statement from **value** to **main.**

```
Program
  void printline (char ch, int len);
  value (float, float, int);

     main( )
     {
        float principal, inrate, amount;
        int period;
        printf("Enter principal amount, interest");
        printf("rate, and period\n");
        scanf(%f %f %d", &principal, &inrate, &period);
```

```
        printline ('*' , 52);
        amount = value (principal, inrate, period);
        printf("\n%f\t%f\t%d\t%f\n\n",principal,
           inrate,period,amount);
        printline('=',52);
     }
  void printline(char ch, int len)
  {
        int i;
        for (i=1;i<=len;i++) printf("%c",ch);
        printf("\n");
  }
  value(float p, float r, int n) /* default return type */
  {
     int year;
     float sum;
        sum = p; year = 1;
        while(year <=n)
        {
           sum = sum * (1+r);
           year = year +1;
        }
        return(sum);  /* returns int part of sum */
  }
Output

  Enter principal amount, interest rate, and period
  5000      0.12    5
  ************************************************
  5000.000000     0.1200000    5      8811.000000
  = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

Fig. 10.9 *Functions with arguments and return values*

The calculated value is passed on to **main** through statement:

```
return(sum);
```

Since, by default, the return type of **value** function is **int,** the 'integer' value of **sum** at this point is returned to **main** and assigned to the variable **amount** by the functional call

```
amount = value (principal, inrate, period);
```

The following events occur, in order, when the above function call is executed:

1. The function call transfers the control along with copies of the values of the actual arguments to the function **value** where the formal arguments **p, r**, and **n** are assigned the actual values of **principal, inrate** and **period** respectively.
2. The called function **value** is executed line by line in a normal fashion until the **return(sum);** statement is encountered. At this point, the integer value of **sum** is passed back to the function-call in the **main** and the following indirect assignment occurs:

```
value(principal, inrate, period) = sum;
```

3. The calling statement is executed normally and the returned value is thus assigned to **amount**, a **float** variable.
4. Since **amount** is a **float** variable, the returned integer part of sum is converted to floating-point value. See the output.

Another important change is the inclusion of second argument to **printline** function to receive the value of length of the line from the calling function. Thus, the function call

```
printline('*', 52);
```

will transfer the control to the function **printline** and assign the following values to the formal arguments **ch**, and **len;**

```
ch = '*' ;
len = 52;
```

10.12.1 Returning Float Values

We mentioned earlier that a C function returns a value of the type **int** as the default case when no other type is specified explicitly. For example, the function **value** of Example 10.3 does all calculations using **floats** but the return statement

```
return(sum);
```

returns only the integer part of **sum.** This is due to the absence of the *type-specifier* in the function header. In this case, we can accept the integer value of **sum** because the truncated decimal part is insignificant compared to the integer part. However, there will be times when we may find it necessary to receive the **float** or **double** type of data. For example, a function that calculates the mean or standard deviation of a set of values should return the function value in either **float** or **double.**

In all such cases, we must explicitly specify the *return type* in both the function definition and the prototype declaration.

If we have a mismatch between the type of data that the called function returns and the type of data that the calling function expects, we will have unpredictable results. We must, therefore, be very careful to make sure that both types are compatible.

Example 10.4 Write a function **power** that computes x raised to the power y for integers x and y and returns double-type value.

Figure. 10.10 shows a **power** function that returns a **double.** The prototype declaration

```
double power(int, int);
```

appears in **main**, before **power** is called.

Program

```
main( )
{
   int x,y;                    /*input data */
   double power(int, int);     /* prototype declaration*/
   printf("Enter x,y:");
   scanf("%d %d" , &x,&y);
   printf("%d to power %d is %f\n", x,y,power (x,y));
```

Contd.

```
        }
        double power (int x, int y);
        {
           double p;
           p = 1.0 ;        /* x to power zero */
           if(y >=0)
              while(y--)      /* computes positive powers */
              p *= x;
           else
              while (y++) /* computes negative powers */
              p /= x;
           return(p);       /* returns double type */
     }
Output
     Enter x,y:16 2
     16 to power 2 is 256.000000

     Enter x,y:16 -2
     16 to power -2 is 0.003906
```

Fig. 10.10 *Power fuctions: Illustration of return of float values*

Another way to guarantee that **power**'s type is declared before it is called in **main** is to define the **power** function before we define **main. Power**'s type is then known from its definition, so we no longer need its type declaration in **main.**

10.13 No Arguments but Returns a Value

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A typical example is the **getchar** function declared in the header file **<stdio.h>.** We have used this function earlier in a number of places. The **getchar** function has no parameters but it returns an integer type data that represents a character.

We can design similar functions and use in our programs. Example:

```
int get_number(void);
main
{
    int m = get_number( );
    printf("%d",m);
}
int get_number(void)
{
    int number;
    scanf("%d", &number);
    return(number);
}
```

10.14 Functions that Return Multiple Values

Up till now, we have illustrated functions that return just one value using a return statement. That is because, a return statement can return only one value. Suppose, however, that we want to get more information from a function. We can achieve this in C using the arguments not only to receive information but also to send back information to the calling function. The arguments that are used to "send out" information are called *output parameters.*

The mechanism of sending back information through arguments is achieved using what are known as the *address operator* (&) and *indirection operator* (*). Let us consider an example to illustrate this.

```
void mathoperation (int x, int y, int *s, int *d);
main( )
{
      int x = 20, y = 10, s, d;
      mathoperation(x,y, &s, &d);

      printf("s=%d\n d=%d\n", s,d);
}
void mathoperation (int a, int b, int *sum, int *diff)
{
   *sum = a+b;
   *diff = a-b;
}
```

The actual arguments **x** and **y** are input arguments, **s** and **d** are output arguments. In the function call, while we pass the actual values of **x** and **y** to the function, we pass the addresses of locations where the values of **s** and **d** are stored in the memory. (That is why, the operator & is called the address operator.) When the function is called the following assignments occur:

value of	x to a
value of	y to b
address of	s to sum
address of	d to diff

Note that indirection operator * in the declaration of **sum** and **diff** in the header indicates these variables are to store addresses, not actual values of variables. Now, the variables **sum** and **diff** point to the memory locations of **s** and **d** respectively.

(The operator * is known as indirection operator because it gives an indirect reference to a variable through its address.)

In the body of the function, we have two statements:

```
* sum  = a+b;
* diff = a-b;
```

The first one adds the values **a** and **b** and the result is stored in the memory location pointed to by **sum.** Remember, this memory location is the same as the memory location of **s.** Therefore, the value stored in the location pointed to by **sum** is the value of **s.**

Similarly, the value of a–b is stored in the location pointed to by **diff,** which is the same as the location **d.** After the function call is implemented, the value of **s** is a+b and the value of **d** is a–b. Output will be:

s = 30
d = 10

The variables ***sum** and ***diff** are known as *pointers* and **sum** and **diff** as *pointer* variables. Since they are declared as **int**, they can point to locations of **int** type data.

The use of pointer variables as actual parameters for communicating data between functions is called "pass by pointers" or "call by address or reference".

Rules for Pass by Pointers

1. The types of the actual and formal arguments must be same.
2. The actual arguments (in the function call) must be the addresses of variables that are local to the calling function.
3. The formal arguments in the function header must be prefixed by the indirection operatior *.
4. In the prototype, the arguments must be prefixed by the symbol *.
5. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed with the indirection operator *.

10.15 Nesting of Functions

C permits nesting of functions freely. **main** can call **function1,** which calls **function2,** which calls **function3,** ……. and so on. There is in principle no limit as to how deeply functions can be nested.

Consider the following program:

```
float ratio (int x, int y, int z);
int difference (int x, int y);
main( )
{
   int a, b, c;
   scanf("%d %d %d", &a, &b, &c);
   printf("%f \n", ratio(a,b,c));
}
float ratio(int x, int y, int z)
{
   if(difference(y, z))
      return(x/(y-z));
   else
      return(0.0);
}
int difference(int p, int q)
{
   if(p != q)
       return (1);
   else
    return(0);
}
```

The above program calculates the ratio

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

and prints the result. We have the following three functions:

main( )
ratio( )
difference( )

main reads the values of a, b and c and calls the function **ratio** to calculate the value a/(b–c). This ratio cannot be evaluated if (b–c) = 0. Therefore, **ratio** calls another function **difference** to test whether the difference (b–c) is zero or not; **difference** returns 1, if b is not equal to c; otherwise returns zero to the function **ratio**. In turn, **ratio** calculates the value a/(b–c) if it receives 1 and returns the result in **float.** In case, **ratio** receives zero from **difference,** it sends back 0.0 to **main** indicating that (b–c) = 0.

Nesting of function calls is also possible. For example, a statement like

```
P = mul(mul(5,2),6);
```

is valid. This represents two sequential function calls. The inner function call is evaluated first and the returned value is again used as an actual argument in the outer function call. If **mul** returns the product of its arguments, then the value of **p** would be 60 (= 5×2×6).

Note that the nesting does not mean defining one function within another. Doing this is illegal.

10.16 Recursion

When a called function in turn calls another function a process of 'chaining' occurs. *Recursion* is a special case of this process, where a function calls itself. A very simple example of recursion is presented below:

```
main( )
{
     printf("This is an example of recursion\n")
     main( );
}
```

When executed, this program will produce an output something like this:

```
This is an example of recursion
This is an example of recursion
This is an example of recursion
This is an ex
```

Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Another useful example of recursion is the evaluation of factorials of a given number. The factorial of a number n is expressed as a series of repetitive multiplications as shown below:

factorial of n = n(n–1)(n–2).....1.

For example,

factorial of 4 = 4×3×2×1 = 24

A function to evaluate factorial of n is as follows:

```
factorial(int n)
   {
      int fact;
      if (n==1)
         return(1);
      else
         fact = n*factorial(n-1);
      return(fact);
   }
```

Let us see how the recursion works. Assume n = 3. Since the value of n is not 1, the statement

```
fact = n * factorial(n-1);
```

will be executed with n = 3. That is,

```
fact = 3 * factorial(2);
```

will be evaluated. The expression on the right-hand side includes a call to **factorial** with n = 2. This call will return the following value:

2 * factorial(1)

Once again, **factorial** is called with n = 1. This time, the function returns 1. The sequence of operations can be summarized as follows:

fact = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1
= 6

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an **if** statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

10.17 Passing Arrays to Functions

10.17.1 One-Dimensional Arrays

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one-dimensional an array to a called function, it is sufficient to list the name of the array, *without any subscripts,* and the size of the array as arguments. For example, the call

largest(a,n)

will pass the whole array **a** to the called function. The called function expecting this call must be appropriately defined. The **largest** function header might look like:

float largest(float array[ ], int size)

The function **largest** is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```
float array[ ];
```

The pair of brackets informs the compiler that the argument **array** is an array of numbers. It is not necessary to specify the size of the **array** here.

Let us consider a problem of finding the largest value in an array of elements. The program is as follows:

```
main( )
{
     float largest(float a[ ], int n);
     float value[4] = {2.5,-4.75,1.2,3.67};
     printf("%f\n", largest(value,4));
}
float largest(float a[], int n)
{
     int i;
     float max;
     max = a[0];
     for(i = 1; i < n; i++)
          if(max < a[i])
     max = a[i];
    return(max);
}
```

When the function call **largest**(value,4) is made, the values of all elements of array **value** become the corresponding elements of array **a** in the called function. The **largest** function finds the largest value in the array and returns the result to the **main.**

In C, the name of the array represents the address of its first element. By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the functions is referred to as *pass by address* (or pass by pointers). Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

Example 10.5 Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use functions to calculate standard deviation and mean.

Standard deviation of a set of n values is given by

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

where $\bar{x}$ is the mean of the values.

Program

```
#include      <math.h>
#define SIZE   5
float std_dev(float a[], int n);
float mean (float a[], int n);
```

```
{
        float value[SIZE];
        int i;

        printf("Enter %d float values\n", SIZE);
        for (i=0 ;i < SIZE ; i++)
           scanf("%f", &value[i]);
        printf("Std.deviation is %f\n", std_dev(value,SIZE));
}
float std_dev(float a[], int n)
{
      int i;

      float x, sum = 0.0;
      x = mean (a,n);
      for(i=0; i < n; i++)
        sum += (x-a[i])*(x-a[i]);
      return(sqrt(sum/(float)n));
}
float mean(float a[],int n)
{
      int i ;
      float sum = 0.0;
      for(i=0 ; i < n ; i++)
         sum = sum + a[i];
      return(sum/(float)n);
}
```

Output

```
Enter 5 float values
35.0 67.0 79.5 14.20 55.75

Std.deviation is 23.231582
```

Fig. 10.11 *Passing of arrays to a function*

A multifunction program consisting of **main, std_dev,** and **mean** functions is shown in Fig. 10.11. **main** reads the elements of the array **value** from the terminal and calls the function **std_dev** to print the standard deviation of the array elements. **Std_dev,** in turn, calls another function **mean** to supply the average value of the array elements.

Both **std_dev** and **mean** are defined as **floats** and therefore they are declared as **floats** in the global section of the program.

Three Rules to Pass an Array to a Function

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

When dealing with array arguments, we should remember one major distinction. If a function changes the values of the elements of an array, then these changes will be made to the original array that passed to the function. When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array; instead, information about the addresses of array elements are passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on as argument. Example 10.6 highlights these concepts.

Example 10.6 Write a program that uses a function to sort an array of integers.

A program to sort an array of integers using the function **sort()** is given in Fig. 10.12. Its output clearly shows that a function can change the values in an array passed as an argument.

Program

```
void sort(int m, int x[ ]);
main()
{
      int i;
      int marks[5] = {40, 90, 73, 81, 35};

      printf("Marks before sorting\n");
      for(i = 0; i < 5; i++)
         printf("%d ", marks[i]);
      printf("\n\n");

      sort (5, marks);

      printf("Marks after sorting\n");
      for(i = 0; i < 5; i++)
         printf("%4d", marks[i]);
      printf("\n");
}
void sort(int m, int x[ ])
{
      int i, j, t;

      for(i = 1; i <= m-1; i++)
         for(j = 1; j <= m-i; j++)
          if(x[j-1] >= x[j])
          {
             t = x[j-1];
             x[j-1] = x[j];
          x[j] = t;
      }
}
```

Output

```
      Marks before sorting
```

Contd

```
40 90 73 81 35

Marks after sorting
35 40 73 81 90
```

Fig. 10.12 *Sorting of array elements using a function*

10.17.2 Two-Dimensional Arrays

Like simple arrays, we can also pass multi-dimensional arrays to functions. The approach is similar to the one we did with one-dimensional arrays. The rules are simple.

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function header.

The function given below calculates the average of the values in a two-dimensional matrix.

```
double average(int x[][N], int M, int N)
{
     int i, j;
     double sum = 0.0;
     for (i=0; i<M; i++)
          for(j=1; j<N; j++)
        sum += x[i][j];
     return(sum/(M*N));
}
```

This function can be used in a main function as illustrated below:

```
main( )
{
      int M=3, N=2;
      double average(int [ ] [N], int, int);
      double mean;
      int matrix [M][N]=
               {
               {1,2},
               {3,4},
               {5,6}
               };

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

10.18 Passing Strings to Functions

The strings are treated as character arrays in C and therfore the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is defined. Example:

```
void display(char item_name[ ])
{
        ......
        ......
}
```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[ ]);
```

3. A call to the function must have a string array name without subscripts as its actual argument. Example:

```
display (names);
```

where **names** is a properly declared string array in the calling function.

We must note here that, like arrays, strings in C cannot be passed by value to functions.

Pass by Value versus Pass by Pointers

The technique used to pass data from one function to another is known as *parameter passing*. Parameter passing can be done in two ways:

- Pass by value (also known as call by value).
- Pass by pointers (also known as call by pointers).

In *pass by value,* values of actual parameters are copied to the variables in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters. This ensures that the original data in the calling function cannot be changed accidentally.

In *pass by pointers* (also known as pass by address), the memory addresses of the variables rather than the copies of values are sent to the called function. In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.

Pass by pointers method is often used when manipulating arrays and strings. This method is also used when we require multiple values to be returned by the called function.

10.19 The Scope, Visibility and Lifetime of Variables

Variables in C differ in behaviour from those in most other languages. For example, in a BASIC program, a variable retains its value throughout the program. It is not always the case in C. It all depends on the 'storage' class a variable may assume.

In C not only do all variables have a data type, they also have a *storage class*. The following variable storage classes are most relevant to functions:

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

We shall briefly discuss the *scope, visibility* and *longevity* of each of the above class of variables. The *scope* of variable determines over what region of the program a variable is actually available for use ('active'). *Longevity* refers to the period during which a variable retains a given value during execution of a program ('alive'). So longevity has a direct effect on the utility of a given variable. The *visibility* refers to the accessibility of a variable from the memory.

The variables may also be broadly categorized, depending on the place of their declaration, as *internal* (local) or *external* (global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

It is very important to understand the concept of storage classes and their utility in order to develop efficient multifunction programs.

10.19.1 Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are *created* when the function is called and *destroyed* automatically when the function is exited, hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as *local* or *internal* variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable. For instance, the storage class of the variable **number** in the example below is automatic.

```
main( )
{
     int number;
     -----
     -----
}
```

We may also use the keyword **auto** to declare automatic variables explicitly.

```
main( )
{
    auto int number;
    -----
    -----
}
```

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

Example 10.7 Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 10.13. **m** is an automatic variable and it is declared at the beginning of each function. **m** is initialized to 10, 100, and 1000 in function1, function2, and **main** respectively.

When executed, **main** calls **function2** which in turn calls **function1**. When **main** is active, m = 1000; but when **function2** is called, the **main**'s **m** is temporarily put on the shelf and the new local **m** = 100 becomes active. Similarly, when **function1** is called, both the previous values of **m** are put on the shelf and the latest value of **m** (=10) becomes active. As soon as **function1** (m=10) is finished, **function2** (m=100) takes over again. As soon it is done, **main** (m=1000) takes over. The output clearly shows that the value assigned to **m** in one function does not affect its value in the other functions; and the local value of **m** is destroyed when it leaves a function.

```
Program
   void function1(void);
   void function2(void);
   main( )
   {
         int m = 1000;
         function2();

         printf("%d\n",m);  /* Third output */
   }
      void function1(void)
   {
      int m = 10;

      printf("%d\n",m);  /* First output */
}

   void function2(void)
{
      int m = 100;
      function1();
      printf("%d\n",m);  /* Second output */
}

Output
   10
   100
   1000
```

Fig. 10.13 *Working of automatic variables*

There are two consequences of the scope and longevity of **auto** variables worth remembering. First, any variable local to **main** will be normally *alive* throughout the whole program, although it is *active* only in **main**. Secondly, during recursion, the nested variables are unique **auto** variables, a situation similar to function-nested **auto** variables with identical names.

10.19.2 External Variables

Variables that are both *alive* and *active* throughout the entire program are known as *external* variables. They are also known as *global* variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function. For example, the external declaration of integer **number** and float **length** might appear as:

```
int number;
float length = 7.5;
main( )
{
 -------
 -------
}
function1( )
{
 -------
 -------
}
function2( )
{
 -------
 -------
}
```

The variables **number** and **length** are available for use in all the three functions. In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared. Consider the following example:

```
int count;
main( )
{
   count = 10;
   -----
   -----
}
function( )
{
   int count = 0;
   ------
   ------
   count = count+1;
}
```

When the **function** references the variable **count,** it will be referencing only its local variable, not the global one. The value of **count** in **main** will not be affected.

Example 10.8 Write a multifunction program to illustrate the properties of global variables.

A program to illustrate the properties of global variables is presented in Fig. 10.14. Note that variable **x** is used in all functions but none except **fun2,** has a definition for **x.** Because **x** has been declared

'above' all the functions, it is available to each function without having to pass x as a function argument. Further, since the value of **x** is directly available, we need not use **return(x)** statements in **fun1** and **fun3.** However, since **fun2** has a definition of **x,** it returns its local value of **x** and therefore uses a **return** statement. In **fun2**, the global **x** is not visible. The local **x** hides its visibility here.

```
Program
    int fun1(void);
    int fun2(void);
    int fun3(void);
    int x ;          /* global */
    main( )
    {
         x = 10 ;        /* global x */
         printf("x = %d\n", x);
         printf("x = %d\n", fun1());
         printf("x = %d\n", fun2());
         printf("x = %d\n", fun3());
}
fun1(void)
{
         x = x + 10 ;
}
int fun2(void)
{
         int x ;             /* local */
         x = 1 ;
         return (x);
}
fun3(void)
{
         x = x + 10 ;        /* global x */
}

Output
         x = 10
         x = 20
         x = 1
         x = 30
```

Fig. 10.14 *Illustration of properties of global variables*

Once a variable has been declared as global, any function can use it and change its value. Then, subsequent functions can reference only that new value.

Global Variables as Parameters

Since all functions in a program source file can access global variables, they can be used for passing values between the functions. However, using global variables as parameters for passing values poses certain problems.

- The values of global variables which are sent to the called function may be changed inadvertently by the called function.

Contd

- Functions are supposed to be independent and isolated modules. This character is lost, if they use global variables.
- It is not immediately apparent to the reader which values are being sent to the called function.
- A function that uses global variables suffers from reusability.

One other aspect of a global variable is that it is available only from the point of declaration to the end of the program. Consider a program segment as shown below:

```
main( )
{
    y = 5;
    . . . .
    . . . .
}
int y;       /* global declaration */
func1( )
{
      y = y+1;
}
```

We have a problem here. As far as **main** is concerned, **y** is not defined. So, the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement

```
y = y+1;
```

in **fun1** will, therefore, assign 1 to y.

10.19.3 External Declaration

In the program segment above, the **main** cannot access the variable y as it has been declared after the **main** function. This problem can be solved by declaring the variable with the storage class **extern.**

For example:

```
main( )
{
    extern int y;   /* external declaration */
    . . . . .
    . . . . .
}
func1( )
{
    extern int y;   /* external declaration */
    . . . . .
    . . . . .
}
int y;                /* definition */
```

Although the variable **y** has been defined after both the functions, the *external declaration* of **y** inside the functions informs the compiler that y is an integer type defined somewhere else in the program. Note

that **extern** declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.

Example:

```
main( )
{   int i;
    void print_out(void);
    extern float height [ ];
    . . . . .
    . . . . .
    print_out( );
}
void print_out(void)
{
    extern float height [ ];
    int i;
    . . . . .
    . . . . .
}
float height[SIZE];
```

An **extern** within a function provides the type information to just that one function. We can provide type information to all functions within a file by placing external declarations before any of them.

Example:

```
extern float height[ ];
main( )
{
    int i;
    void print_out(void);
    . . . . .
    . . . . .
    print_out( );
}
void print_out(void)
{
    int i;
    . . . . .
    . . . . .
}
float height[SIZE];
```

The distinction between definition and declaration also applies to functions. A function is defined when its parameters and function body are specified. This tells the compiler to allocate space for the function code and provides type information for the parameters. Since functions are external by default, we declare them (in the calling functions) without the qualifier **extern.** Therefore, the declaration

```
void print_out(void);
```

is equivalent to

```
extern void print_out(void);
```

Function declarations outside of any function behave the same way as variable declarations.

10.19.4 Static Variables

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared *static* using the keyword **static** like

```
static int x;
static float y;
```

A static variable may be either an internal type or an external type depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extend up to the end of the function in which they are defined. Therefore, internal **static** variables are similar to **auto** variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal **static** variables can be used to retain values between function calls. For example, it can be used to count the number of calls made to a function.

Example 10.9 Write a program to illustrate the properties of a static variable.

The program in Fig. 10.15 explains the behavior of a static variable.

```
Program
   void stat(void);
   main ( )
   {
      int i;
      for(i=1; i<=3; i++)
      stat( );
   }
   void stat(void)
   {
      static int x = 0;

      x = x+1;
      printf("x = %d\n", x);
   }
Output
      x = 1
      x = 2
      x = 3
```

Fig. 10.15 *Illustration of static variable*

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to **stat, x** is incremented to 1. Because **x** is static, this value persists and therefore, the next call adds another 1 to **x** giving it a value of 2. The value of **x** becomes three when the third call is made.

Had we declared **x** as an **auto** variable, the output would have been:

x = 1
x = 1
x = 1

This is because each time **stat** is called, the auto variable x is initialized to zero. When the function terminates, its value of 1 is lost.

An external **static** variable is declared outside of all functions and is available to all the functions in that program. The difference between a **static** external variable and a simple external variable is that the **static** external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining 'that' function with the storage class **static**.

10.19.5 Register Variables

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables (e.g. loop control variables) in the register will lead to faster execution of programs. This is done as follows:

```
register int count;
```

Although, ANSI standard does not restrict its application to any particular data type, most compilers allow only **int** or **char** variables to be placed in the register.

Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, C will automatically convert **register** variables into non-register variables once the limit is reached.

Table 10.1 summarizes the information on the visibility and lifetime of variables in functions and files.

Table 10.1 ***Scope and Lifetime of Variables***

Storage Class	Where declared	Visibility (Active)	Lifetime (Alive)
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with **extern**	Entire program (Global)
extern	Before all functions in a file (cannot be initialized) **extern** and the file where originally declared as global.	Entire file plus other files where variable is declared	Global
static	Before all functions in a file	Only in that file	Global
None or **auto**	Inside a function (or a block)	Only in that function or block	Until end of function or block

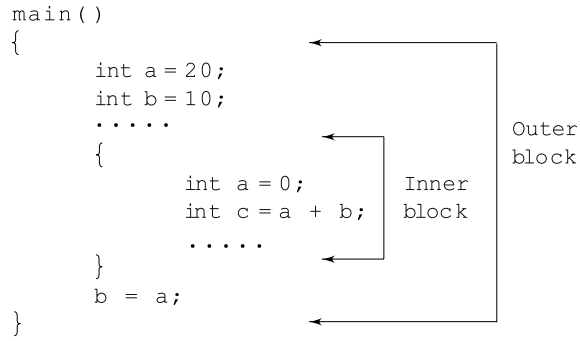
Contd

Table 10.1 *(Contd)*

Storage Class	Where declared	Visibility (Active)	Lifetime (Alive)
register	Inside a function or block	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global

10.19.6 Nested Blocks

A set of statements enclosed in a set of braces is known a *block* or a *compound* statement. Note that all functions including the **main** use compound *statement.* A block can have its own declarations and other statements. It is also possible to have a block of such statements inside the body of a function or another block, thus creating what is known as *nested blocks* as shown below:



When this program is executed, the value c will be 10, not 30. The statement b = a; assigns a value of 20 to **b** and not zero. Although the scope of **a** extends up to the end of **main** it is not "visible" inside the inner block where the variable **a** has been declared again. The inner **a** hides the visibility of the outer **a** in the inner block. However, when we leave the inner block, the inner **a** is no longer in scope and the outer **a** becomes visible again.

Remember, the variable **b** is not re-declared in the inner block and therefore it is visible in both the blocks. That is why when the statement

```
int c = a + b;
```

is evaluated, **a** assumes a values of 0 and **b** assumes a value of 10.

Although main's variables are visible inside the nested block, the reverse is not true.

Scope Rules

Scope

The region of a program in which a variable is available for use.

Visibility

The program's ability to access a variable from the memory.

Lifetime

The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an **auto** variable declared in **main** is the entire program execution time, although its scope is only the **main** function.
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A **static** local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.
7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

Just Remember

- It is a syntax error if the types in the declaration and function definition do not match.
- It is a syntax error if the number of actual parameters in the function call do not match the number in the declaration statement.
- It is a logic error if the parameters in the function call are placed in the wrong order.
- It is illegal to use the name of a formal argument as the name of a local variable.
- Using **void** as return type when the function is expected to return a value is an error.
- Trying to return a value when the function type is marked **void** is an error.
- Variables in the parameter list must be individually declared for their types. We cannot use multiple declarations (like we do with local or global variables).
- A **return** statement is required if the return type is anything other than **void.**
- If a function does not return any value, the return type must be declared **void.**
- If a function has no parameters, the parameter list must be declared **void.**
- Placing a semicolon at the end of header line is illegal.
- Forgetting the semicolon at the end of a prototype declaration is an error.
- Defining a function within the body of another function is not allowed.
- It is an error if the type of data returned does not match the return type of the function.
- It will most likely result in logic error if there is a mismatch in data types between the actual and formal arguments.
- Functions return integer value by default.
- A function without a return statement cannot return a value, when the parameters are passed by value.
- A function that returns a value can be used in expressions like any other C variable.
- When the value returned is assigned to a variable, the value will be converted to the type of the variable receiving it.
- Function cannot be the target of an assignment.
- A function with void return type cannot be used in the right-hand side of an assignment statement. It can be used only as a stand-alone statement.

- A function that returns a value cannot be used as a stand-alone statement.
- A **return** statement can occur anywhere within the body of a function.
- A function can have more than one return statement.
- A function definition may be placed either after or before the **main** function.
- Where more functions are used, they may be placed in any order.
- A global variable used in a function will retain its value for future use.
- A local variable defined inside a function is known only to that function. It is destroyed when the function is exited.
- A global variable is visible only from the point of its declaration to the end of the program.
- When a variable is redeclared within its scope either in a function or in a block, the original variable is not visible within the scope of the redeclared variable.
- A local variable declared **static** retains its value even after the function is exited.
- Static variables are initialized at compile time and therefore they are initialized only once.
- Use parameter passing by values as far as possible to avoid inadvertent changes to variables of calling function in the called function.
- Although not essential, include parameter names in the prototype declarations for documentation purposes.
- Avoid the use of names that hide names in outer scope.

Review Questions

10.1 State whether the following statements are *true* or *false*.

(a) C functions can return only one value under their function name.
(b) A function in C should have at least one argument.
(c) A function can be defined and placed before the **main** function.
(d) A function can be defined within the **main** function.
(e) An user-defined function must be called at least once; otherwise a warning message will be issued.
(f) Any name can be used as a function name.
(g) Only a **void** type function can have **void** as its argument.
(h) When variable values are passed to functions, a copy of them are created in the memory.
(i) Program execution always begins in the main function irrespective of its location in the program.
(j) Global variables are visible in all blocks and functions in the program.
(k) A function can call itself.
(l) A function without a **return** statement is illegal.
(m) Global variables cannot be declared as **auto** variables.
(n) A function prototype must always be placed outside the calling function.
(o) The return type of a function is **int** by default.
(p) The variable names used in prototype should match those used in the function definition.
(q) In parameter passing by pointers, the formal parameters must be prefixed with the symbol * in their declarations.

(r) In parameter passing by pointers, the actual parameters in the function call may be variables or constants.

(s) In passing arrays to functions, the function call must have the name of the array to be passed without brackets.

(t) In passing strings to functions, the actual parameter must be name of the string post-fixed with size in brackets.

10.2 Fill in the blanks in the following statements.

(a) The parameters used in a function call are called _____.

(b) A variable declared inside a function is called _____.

(c) By default, _____ is the return type of a C function.

(d) In passing by pointers, the variables of the formal parameters must be prefixed with _____ in their declaration.

(e) In prototype declaration, specifying is _____ optional.

(f) _____ refers to the region where a variable is actually available for use.

(g) A function that calls itself is known as a _____ function.

(h) If a local variable has to retain its value between calls to the function, it must be declared as _____.

(i) A _____ aids the compiler to check the matching between the actual arguments and the formal ones.

(j) A variable declared inside a function by default assumes _____ storage class.

10.3 The **main** is a user-defined function. How does it differ from other user-defined functions?

10.4 Describe the two ways of passing parameters to functions. When do you prefer to use each of them?

10.5 What is prototyping? Why is it necessary?

10.6 Distinguish between the following:

(a) Actual and formal arguments

(b) Global and local variables

(c) Automatic and static variables

(d) Scope and visibility of variables

(e) & operator and * operator

10.7 Explain what is likely to happen when the following situations are encountered in a program.

(a) Actual arguments are less than the formal arguments in a function.

(b) Data type of one of the actual arguments does not match with the type of the corresponding formal argument.

(c) Data type of one of the arguments in a prototype does not match with the type of the corresponding formal parameter in the header line.

(d) The order of actual parameters in the function call is different from the order of formal parameters in a function where all the parameters are of the same type.

(e) The type of expression used in **return** statement does not match with the type of the function.

10.8 Which of the following prototype declarations are invalid? Why?

(a) `int (fun) void;`

(b) `double fun (void)`

(c) `float fun (x, y, n);`

(d) `void fun (void, void);`

(e) `int fun (int a, b);`

(f) `fun (int, float, char);`

(g) `void fun (int a, int &b);`

10.9 Which of the following header lines are invalid? Why?

(a) `float average (float x, float y, float z);`

(b) `double power (double a, int n - 1)`

(c) `int product (int m, 10)`

(d) `double minimum (double x; double y;)`

(e) `int mul (int x, y)`

(f) `exchange (int *a, int *b)`

(g) `void sum (int a, int b, int &c)`

10.10 Find errors, if any, in the following function definitions:

(a)
```
void abc (int a, int b)
{
     int c;
     . . . .
     return (c);
}
```

(b)
```
int abc (int a, int b)
{
     . . . .
     . . . .
}
```

(c)
```
int abc (int a, int b)
{
     double c = a + b;
     return (c);
}
```

(d)
```
void abc (void)
{
     . . . .
     . . . .
     return;
}
```

(e)
```
int abc(void)
{
     . . . .
```

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

10.10 Find errors in the following function calls:

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

10.12 A function to divide two floating point numbers is as follows:

```
divide (float x, float y)
{
        return (x / y);
}
```

What will be the value of the following function calls:

(a) `divide ( 10, 2)`
(b) `divide ( 9, 2 )`
(c) `divide ( 4.5, 1.5 )`
(d) `divide ( 2.0, 3.0 )`

10.13 What will be the effect on the above function calls if we change the header line as follows:

(a) int divide (int x, int y)
(b) double divide (float x, float y)

10.14 Determine the output of the following program?

```
int prod( int m, int n);
main ( )
{
        int x = 10;
        int y = 20;
        int p, q;
        p = prod (x,y);
        q = prod (p, prod (x,z));
        printf ("%d %d\n", p,q);
}
int prod( int a, int b)
{
        return (a * b);
}
```

10.15 What will be the output of the following program?

```
void test (int *a);
main ( )
{
        int x = 50;
        test ( &x);
```

```
        printf("%d\n", x);
}
void test (int *a);
{
        *a = *a + 50;
}
```

10.16 The function **test** is coded as follows:

```
int test (int number)
{
        int m, n = 0;
        while (number)
        {
                m = number % 10;
                if (m % 2)
                n = n + 1;
                        number = number /10;
                }
                return (n);
}
```

What will be the values of **x** and **y** when the following statements are executed?

```
int x = test (135);
int y = test (246);
```

10.17 Enumerate the rules that apply to a function call.

10.18 Summarize the rules for passing parameters to functions by pointers.

10.19 What are the rules that govern the passing of arrays to function?

10.20 State the problems we are likely to encounter when we pass global variables as parameters to functions.

Programming Exercises

10.1 Write a function **exchange** to interchange the values of two variables, say **x** and **y.** Illustrate the use of this function, in a calling function. Assume that **x** and **y** are defined as global variables.

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

10.3 Use recursive function calls to evaluate

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

10.4 An n_order polynomial can be evaluated as follows:

$$P = (.....(((a_0x+a_1)x+a_2)x+a_3)x+\cdots+a_n)$$

Write a function to evaluate the polynomial, using an array variable. Test it using a main program.

10.5 The Fibonacci numbers are defined recursively as follows:

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$, $n > 2$

Write a function that will generate and print the first n Fibonacci numbers. Test the function for n = 5, 10, and 15.

10.6 Write a function that will round a floating-point number to an indicated decimal place. For example the number 17.457 would yield the value 17.46 when it is rounded off to two decimal places.

10.7 Write a function **prime** that returns 1 if its argument is a prime number and returns zero otherwise.

10.8 Write a function that will scan a character string passed as an argument and convert all lowercase characters into their uppercase equivalents.

10.9 Develop a top_down modular program to implement a calculator. The program should request the user to input two numbers and display one of the following as per the desire of the user:

(a) Sum of the numbers
(b) Difference of the numbers
(c) Product of the numbers
(d) Division of the numbers

Provide separate functions for performing various tasks such as reading, calculating and displaying. Calculating module should call second level modules to perform the individual mathematical operations. The main function should have only function calls.

10.10 Develop a modular interactive program using functions that reads the values of three sides of a triangle and displays either its area or its perimeter as per the request of the user. Given the three sides a, b and c.

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

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

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

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

10.12 Write a function that can be called to compute the product of two matrices of size m by n and n by m. The main function provides the values for m and n and two matrices.

10.13 Design and code an interactive modular program that will use functions to a matrix of m by n size, compute column averages and row averages, and then print the entire matrix with averages shown in respective rows and columns.

10.14 Develop a top-down modular program that will perform the following tasks:

(a) Read two integer arrays with unsorted elements.
(b) Sort them in ascending order
(c) Merge the sorted arrays
(d) Print the sorted list

Use functions for carrying out each of the above tasks. The main function should have only function calls.

10.15 Develop your own functions for performing following operations on strings:

(a) Copying one string to another

(b) Comparing two strings
(c) Adding a string to the end of another string

Write a driver program to test your functions.

10.16 Write a program that invokes a function called **find( )** to perform the following tasks:
(a) Receives a character array and a single character.
(b) Returns 1 if the specified character is found in the array, 0 otherwise.

10.17 Design a function **locate ( )** that takes two character arrays **s1** and **s2** and one integer value **m** as parameters and inserts the string **s2** into **s1** immediately after the index **m.**

Write a program to test the function using a real-life situation. (**Hint:** s2 may be a missing word in s1 that represents a line of text.)

10.18 Write a function that takes an integer parameter **m** representing the month number of the year and returns the corresponding name of the month. For instance, if m = 3, the month is March.

Test your program.

10.19 In preparing the calendar for a year we need to know whether that particular year is leap year or not. Design a function **leap( )** that receives the year as a parameter and returns an appropriate message.

What modifications are required if we want to use the function in preparing the actual calendar?

10.20 Write a function that receives a floating point value **x** and returns it as a value rounded to two nearest decimal places. For example, the value 123.4567 will be rounded to 123.46. (**Hint:** Seek help of one of the math functions available in math library.)

Debugging Exercises

10.1 Identify the error in the following code that multiplies two 3 X 3 matrices:

```
void multiply(int x[3][],int y[3][])
{
 int c[3][3],i,j,k;

/*Multiplying the two matrices*/
 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
  {
   c[i][j]=0;
   for(k=0;k<3;k++)
    c[i][j]=c[i][j]+x[i][k]*y[k][j];
  }
 printf("\n\nThe product of the two matrices is shown below: \n");

 for(i=0;i<3;i++)
 {
  printf("\n\t\t    ");
  for(j=0;j<3;j++)
  {
```

```
   printf("%d\t",c[i][j]); /*Displaying the result*/
  }
 }
}
```

10.2 Identify the error in the following program that finds the area of a rectangle:

```
#include <stdio.h>

void main()
{
 area(); /*Calling the area function*/
}

void area()
{
 float length, breadth;

 printf("Enter the length and breadth of the rectangle:\n");
 scanf("%f %f",&length,&breadth) ;

 /*Computing and displaying the area of the rectangle*/
 printf("The area of the rectangle with length %.2f and breadth %.2f
is = %.2f", length,breadth,(length*breadth));
}
```

10.3 Identify the error in the following program, if any:

```
#include <stdio.h>

void Func_A(void);
void Func_B(void);
void Func_C(void);

void main( )
{
 printf("In main function; Calling Func_A()..\n");
 Func_A();/*Calling Function A*/
 printf("Back in main()..\n");

}

void Func_A(void)
{
 printf("Inside Func_A(); Calling Func_B()..\n");
 Func_B();/*Function A calls Function B*/
 printf("Back in Func_A()..\n");
}

void Func_B(void)
```

```
{
 printf("Inside Func_B(); Calling Func_C()..\n");
 Func_C(); /*Function B in turn calls Function C*/
 printf("Back in Func_B()..\n");
}

void Func_C(void)
{
 printf("Inside Func_C()..\n"); /*print statement inside Function
C*/
}
```

10.4 Identify the error in the following program that draws a circle on the console:

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

void main()
{
 int gd = DETECT, gm;
 initgraph(gd, gm, "..\\bgi"); /*Initializing the graphics system*/

 circle(320, 225, 50); /*Calling the built in function circle()*/
 getch();
 closegraph(); /*Shutting the graphics system*/
}
Program
```

10.5 Identify the error in the following program, if any:

```
#include <stdio.h>

void main()
{
 char ch;
 void ASCII(char);
 printf("Enter any character:\n");
 scanf("%c",&ch) ;

 ASCII(ch);

}

void ASCII(int c)
{
 printf("The ASCII value of %c is %d",c,c);/*Displaying the ASCII
value*/
}
```



11 Pointers

11.1 Introduction

A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are undoubtedly one of the most distinct and exciting features of C language. It has added power and flexibility to the language. Although they appear little confusing and difficult to understand for a beginner, they are a powerful tool and handy to use once they are mastered.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They include:

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Of course, the real power of C lies in the proper use of pointers. In this chapter, we will examine the pointers in detail and illustrate how to use them in program development.

11.2 Understanding Pointers

The computer's memory is a sequential collection of *storage cells* as shown in Fig. 11.1. Each cell, commonly known as a *byte,* has a number called *address* associated with it. Typically, the addresses are numbered consecutively, starting from zero. The last address depends on the memory size. A computer system having 64 K memory will have its last address as 65,535.

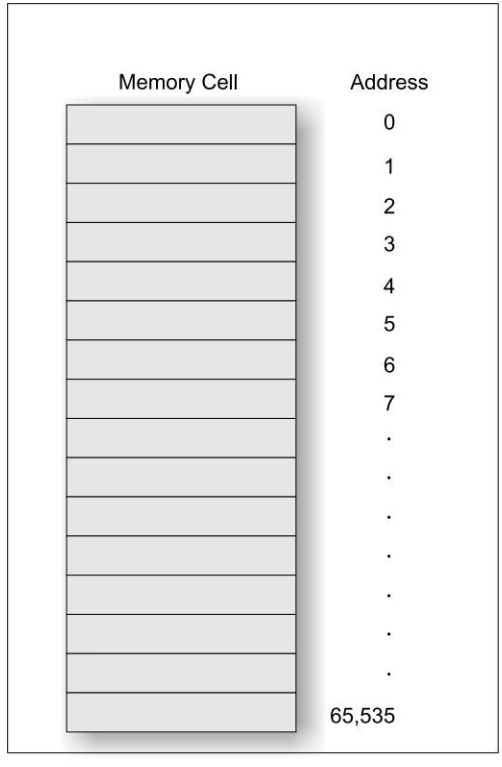


Fig. 11.1 *Memory organisation*

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number. Consider the following statement

```
int quantity = 179;
```

This statement instructs the system to find a location for the integer variable **quantity** and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for **quantity.** We may represent this as shown in Fig. 11.2. (Note that the address of a variable is the address of the first byte occupied by that variable.)

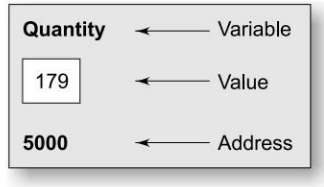


Fig. 11.2 *Representation of a variable*

During execution of the program, the system always associates the name **quantity** with the address 5000. (This is something similar to having a house number as well as a house name.) We may have access to the value 179 by using either the name **quantity** or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables, that can be stored in memory, like any other variable. Such variables that hold memory addresses are called *pointer variables*. A pointer variable is, therefore, nothing but a variable that contains an address, which is a location of another variable in memory.

Remember, since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of **quantity** to a variable **p**. The link between the variables **p** and **quantity** can be visualized as shown in Fig.11.3. The address of **p** is 5048.

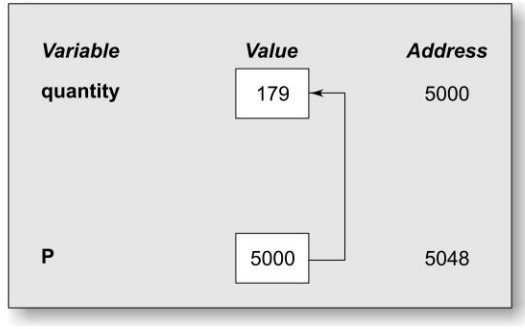
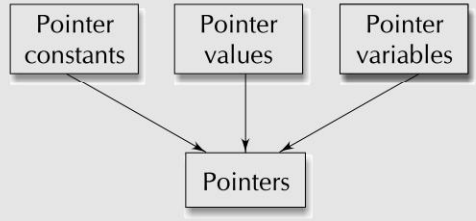


Fig. 11.3 *Pointer variable*

Since the value of the variable **p** is the address of the variable **quantity,** we may access the value of **quantity** by using the value of **p** and therefore, we say that the variable **p** 'points' to the variable **quantity.** Thus, **p** gets the name 'pointer'. (We are not really concerned about the actual values of pointer variables. They may be different everytime we run the program. What we are concerned about is the relationship between the variables **p** and **quantity**.)

Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below:



Memory addresses within a computer are referred to as *pointer constants.* We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as *pointer value.* The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a *pointer variable.*

11.3 Accessing the Address of a Variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable? This can be done with the help of the operator **&** available in C. We have already seen the use of this *address operator* in the **scanf** function. The operator **&** immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
p = &quantity;
```

would assign the address 5000 (the location of **quantity)** to the variable **p**. The **&** operator can be remembered as 'address of'.

The **&** operator can be used only with a simple variable or an array element. The following are illegal uses of address operator:

1. **&125** (pointing at constants).
2. int x[10];
   &x (pointing at array names).
3. **&(x+y)** (pointing at expressions).

If **x** is an array, then expressions such as

&x[0] and &x[i+3]

are valid and represent the addresses of 0th and (i+3)th elements of **x**.

Example 11.1 Write a program to print the address of a variable along with its value.

The program shown in Fig. 11.4, declares and initializes four variables and then prints out these values with their respective storage locations. Note that we have used %u format for printing address values. Memory addresses are unsigned integers.

Program

```
main()
{
   char a;
   int x;
   float p, q;

   a = 'A';
   x = 125;
   p = 10.25, q = 18.76;
   printf("%c is stored at addr %u.\n", a, &a);
   printf("%d is stored at addr %u.\n", x, &x);
   printf("%f is stored at addr %u.\n", p, &p);
```

Contd

```
    printf("%f is stored at addr %u.\n", q, &q);
    }
Output
  A is stored at addr 4436.
  125 is stored at addr 4434.
  10.250000 is stored at addr 4442.
  18.760000 is stored at addr 4438.
```

Fig. 11.4 *Accessing the address of a variable*

11.4 Declaring Pointer Variables

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

This tells the compiler three things about the variable **pt_name.**

1. The asterisk (*) tells that the variable **pt_name** is a pointer variable.
2. **pt_name** needs a memory location.
3. **pt_name** points to a variable of type *data_type*.

For example,

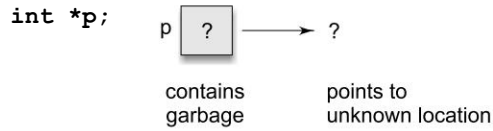
```
int *p;    /* integer pointer */
```

declares the variable **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

```
float *x;    / * float pointer */
```

declares **x** as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables **p** and **x.** Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:



Pointer Declaration Style

Pointer variables are declared similarly as normal variables except for the addition of the unary * operator. This symbol can appear anywhere between the type name and the printer variable name. Programmers use the following styles:

```
int*      p;    /* style 1 */
int*      p;    /*  style 2 */
int*      p;    /*  style 3 */
```

However, the style2 is becoming increasingly popular due to the following reasons:

1. This style is convenient to have multiple declarations in the same statement. Example:

```
int *p, x, *q;
```

2. This style matches with the format used for accessing the target values. Example:

```
int x, *p, y;
x = 10;
p = & x;
y = *p;        /* accessing x through p */
*p = 20;       /* assigning 20 to x */
```

We use in this book the style 2, namely,

```
int *p;
```

11.5 Initialization of Pointer Variables

The process of assigning the address of a variable to a pointer variable is known as *initialization*. As pointed out earlier, all uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;
int *p;                /* declaration     */
p = &quantity;         /* initialization  */
```

We can also combine the initialization with the declaration. That is,

```
int *p = &quantity;
```

is allowed. The only requirement here is that the variable **quantity** must be declared before the initialization takes place. Remember, this is an initialization of **p** and not ***p**.

We must ensure that the pointer variables always point to the corresponding type of data. For example,

```
float a, b;
int x,  *p;
p = &a; /* wrong */
b = *p;
```

will result in erroneous output because we are trying to assign the address of a **float** variable to an **integer pointer**. When we declare a pointer to be of **int** type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

```
int x, *p = &x;      /* three in one */
```

is perfectly valid. It declares **x** as an integer variable and **p** as a pointer variable and then initializes **p** to the address of **x.** And also remember that the target variable **x** is declared first. The statement

```
int *p = &x, x;
```

is not valid.

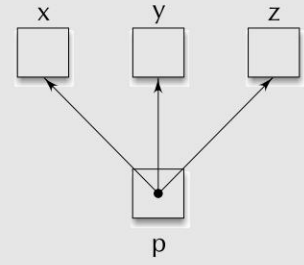
We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued

```
int *p = NULL;
int *p = 0;
```

Pointer Flexibility

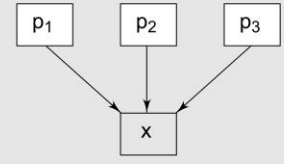
Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example;

```
int x, y, z, *p;
. . . . .
p = &x;
. . . . .
p = &y;
. . . . .
p = &z;
. . . . .
```



We can also use different pointers to point to the same data variable. Example.

```
int x;
int *p1 = &x;
int *p2 = &x;
int *p3 = &x;
. . . . .
. . . . .
```



With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;      / *absolute address */
```

11.6 Accessing a Variable through its Pointer

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer? This is done by using another unary operator * (asterisk), usually known as the *indirection operator.* Another name for the indirection operator is the *dereferencing operator*. Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
```

The first line declares **quantity** and **n** as integer variables and **p** as a pointer variable pointing to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to the pointer variable **p**. The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, ***p** returns the value of the variable **quantity**, because **p** is the address of **quantity.** The * can be remembered as 'value at address'. Thus the value of **n** would be 179. The two statements

```
p = &quantity;
n = *p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

```
n = quantity;
```

In C, the assignment of pointers and addresses is always done symbolically, by means of symbolic names. You cannot access the value stored at the address 5368 by writing *5368. It will not work. Example 11.2 illustrates the distinction between pointer value and the value it points to.

Example 11.2 Write a program to illustrate the use of indirection operator '*' to access the value pointed to by a printer.

The program and output are shown in Fig. 11.5. The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
x = *(&x) = *ptr = y
&x = &*ptr
```

Program

```
main()
{
   int x, y;
   int *ptr;
   x = 10;
   ptr = &x;
   y = *ptr;
   printf("Value of x is %d\n\n",x);
   printf("%d is stored at addr %u\n", x, &x);
   printf("%d is stored at addr %u\n", *&x, &x);
   printf("%d is stored at addr %u\n", *ptr, ptr);
   printf("%d is stored at addr %u\n", ptr, &ptr);
   printf("%d is stored at addr %u\n", y, &y);
   *ptr = 25;
   printf("\nNow x = %d\n",x);
}
```

Output

```
Value of x is 10
10       is stored at addr 4104
10       is stored at addr 4104
10       is stored at addr 4104
4104     is stored at addr 4106
10       is stored at addr 4108
Now x = 25
```

Fig. 11.5 *Accessing a variable through its pointer*

The actions performed by the program are illustrated in Fig. 11.6. The statement **ptr = &x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y**.

Note the use of the assignment statement

```
*ptr = 25;
```

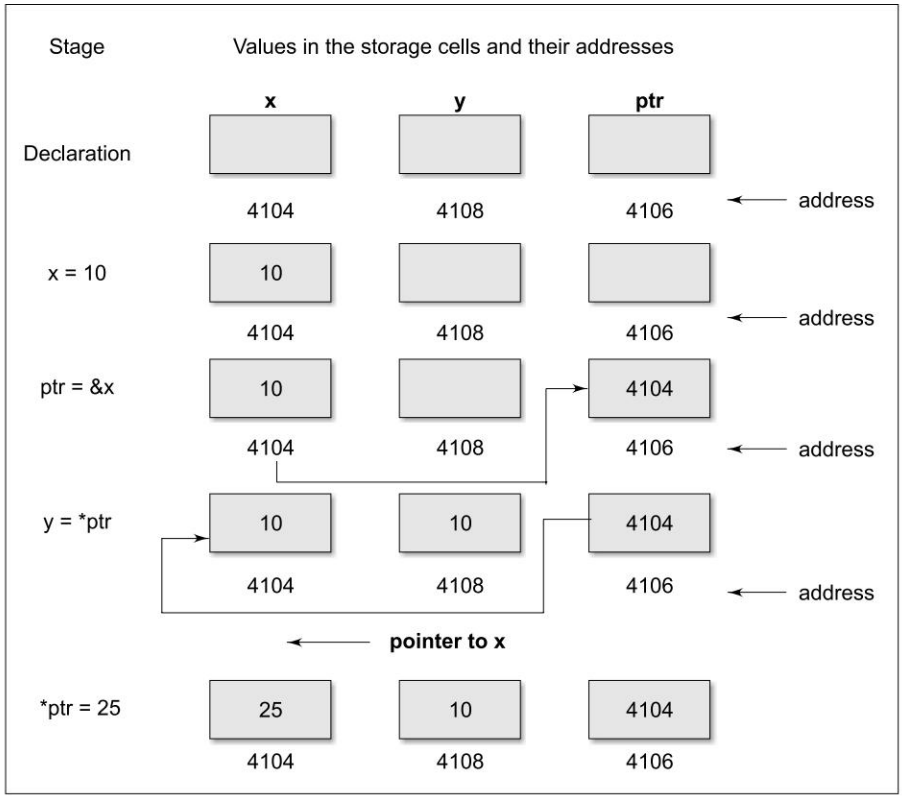
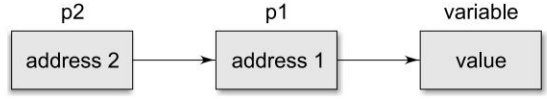


Fig. 11.6 *Illustration of pointer assignments*

This statement puts the value of 25 at the memory location whose address is the value of **ptr**. We know that the value of **ptr** is the address of **x** and therefore, the old value of **x** is replaced by 25. This, in effect, is equivalent to assigning 25 to **x**. This shows how we can change the value of a variable *indirectly* using a pointer and the *indirection operator*.

11.7 Chain of Pointers

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown.



Here, the pointer variable **p2** contains the address of the pointer variable **p1**, which points to the location that contains the desired value. This is known as *multiple indirections*.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Example:

```
int **p2;
```

This declaration tells the compiler that **p2** is a pointer to a pointer of **int** type. Remember, the pointer **p2** is not a pointer to an integer, but rather a pointer to an integer pointer.

We can access the target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice. Consider the following code:

```
main ( )
{
     int x, *p1,    **p2;
     x = 100;
     p1 = &x;       /* address of x */
     p2 = &p1       /* address of p1 */
     printf ("%d",  **p2);
}
```

This code will display the value 100. Here, **p1** is declared as a pointer to an integer and **p2** as a pointer to a pointer to an integer.

11.8 Pointer Expressions

Like other variables, pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then the following statements are valid.

```
y = *p1 * *p2;          same as (*p1) * (*p2)
sum = sum + *p1;
z = 5* - *p2/ *p1;      same as (5 * (- (*p2)))/(*p1)
*p2 = *p2 + 10;
```

Note that there is a blank space between / and * in the item3 above. The following is wrong.

```
z = 5* - *p2 /*p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. p1 + 4, p2–2 and p1 – p2 are all allowed. If p1 and p2 are both pointers to the same array, then **p2 – p1** gives the number of elements between **p1** and **p2**.

We may also use short-hand operators with the pointers.

```
p1++;
–p2;
sum += *p2;
```

In addition to arithmetic operations discussed above, pointers can also be compared using the relational operators. The expressions such as **p1 > p2, p1 == p2**, and **p1 != p2** are allowed. However, any comparison of pointers that refer to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.

We may not use pointers in division or multiplication. For example, expressions such as

```
p1 / p2 or p1 * p2 or p1 / 3
```

are not allowed. Similarly, two pointers cannot be added. That is, p1 + p2 is illegal.

Example 11.3 Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig.11.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

```
4* - *p2 / *p1 + 10
```

is evaluated as follows:

```
((4 * (-(*p2))) / (*p1)) + 10
```

When *p1 = 12 and *p2 = 4, this expression evaluates to 9. Remember, since all the variables are of type int, the entire evaluation is carried out using the integer arithmetic.

```
Program
  main()
  {
     int a, b, *p1, *p2, x, y, z;
     a  = 12;
     b  = 4;
     p1 = &a;
     p2 = &b;
     x  = *p1 * *p2 - 6;
     y  = 4* - *p2 / *p1 + 10;
     printf("Address of a = %u\n", p1);
     printf("Address of b = %u\n", p2);
     printf("\n");
     printf("a = %d, b = %d\n", a, b);
     printf("x = %d, y = %d\n", x, y);
     *p2 = *p2 + 3;
     *p1 = *p2 - 5;
     z   = *p1 * *p2 - 6;
     printf("\na = %d, b = %d,", a, b);
     printf(" z = %d\n", z);
  }
Output
  Address of a = 4020
  Address of b = 4016
  a = 12, b = 4
  x = 42, y = 9
  a = 2, b = 7, z = 8
```

Fig. 11.7 *Evaluation of pointer expressions*

11.9 Pointer Increments and Scale Factor

We have seen that the pointers can be incremented like

```
p1 = p2 + 2;
p1 = p1 + 1;
```

and so on. Remember, however, an expression like

```
p1++;
```

will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1 = p1 + 1**, the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the 'length' of the data type that it points to. This length called the *scale factor*.

For an IBM PC, the length of various data types are as follows:

characters	1 byte
integers	2 bytes

floats	4 bytes
long integers	4 bytes
doubles	8 bytes

The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if **x** is a variable, then **sizeof(x)** returns the number of bytes needed for the variable. (Systems like Pentium use 4 bytes for storing integers and 2 bytes for short integers.)

Rules of Pointer Operations

The following rules apply when performing operations on pointer variables.

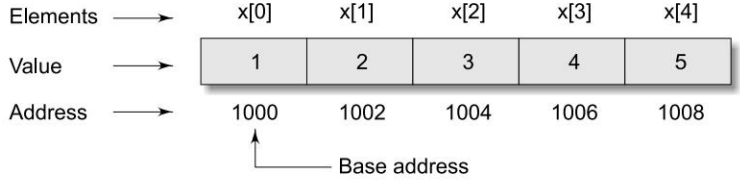
1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e &x = 10; is illegal).

11.10 Pointers and Arrays

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array **x** as follows:

```
int x[5] = {1, 2, 3, 4, 5};
```

Suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:



The name **x** is defined as a constant pointer pointing to the first element, **x[0]** and therefore the value of **x** is 1000, the location where **x[0]** is stored. That is,

```
x = &x[0] = 1000
```

If we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array **x** by the following assignment:

```
p = x;
```

This is equivalent to

```
p = &x[0];
```

Now, we can access every value of **x** using p++ to move from one element to another. The relationship between **p** and **x** is shown as:

p = &x[0] (= 1000)
p+1 = &x[1] (= 1002)
p+2 = &x[2] (= 1004)
p+3 = &x[3] (= 1006)
p+4 = &x[4] (= 1008)

You may notice that the address of an element is calculated using its index and the scale factor of the data type. For instance,

address of **x[3]** = base address + (3 x scale factor of **int**)
= 1000 + (3 x 2) = 1006

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that ***(p+3)** gives the value of **x[3].** The pointer accessing method is much faster than array indexing.

Example 11.4 illustrates the use of pointer accessing method.

Example 11.4 Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig. 11.8 illustrates how a pointer can be used to traverse an array element. Since incrementing an array pointer causes it to point to the next element, we need only to add one to **p** each time we go through the loop.

Program

```
main()
{
   int *p, sum, i;
   int x[5] = {5,9,6,3,7};
   i = 0;
   p = x;     /* initializing with base address of x */
   printf("Element Value Address\n\n");
   while(i < 5)
   {
      printf(" x[%d] %d %u\n", i, *p, p);
      sum = sum + *p; /* accessing array element  */
      i++, p++;       /* incrementing pointer     */
   }
   printf("\n Sum     = %d\n", sum);
   printf("\n &x[0]   = %u\n", &x[0]);
   printf("\n p       = %u\n", p);
}
```

Output

```
          Element      Value     Address
          x[0]         5         166
          x[1]         9         168
          x[2]         6         170
          x[3]         3         172
          x[4]         7         174
```

Contd

```
Sum    = 55
&x[0] = 166
p      = 176
```

Fig. 11.8 *Accessing one-dimensional array elements using the pointer*

It is possible to avoid the loop control variable **i** as shown:

```
.....
p = x;
while(p <= &x[4])
{
sum += *p;
p++;
}
.....
```

Here, we compare the pointer **p** with the address of the last element to determine when the array has been traversed.

Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array **x,** the expression

```
*(x+i) or *(p+i)
```

represents the element **x[i].** Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

```
*(*(a+i)+j) or *(*(p+i)+j)
```

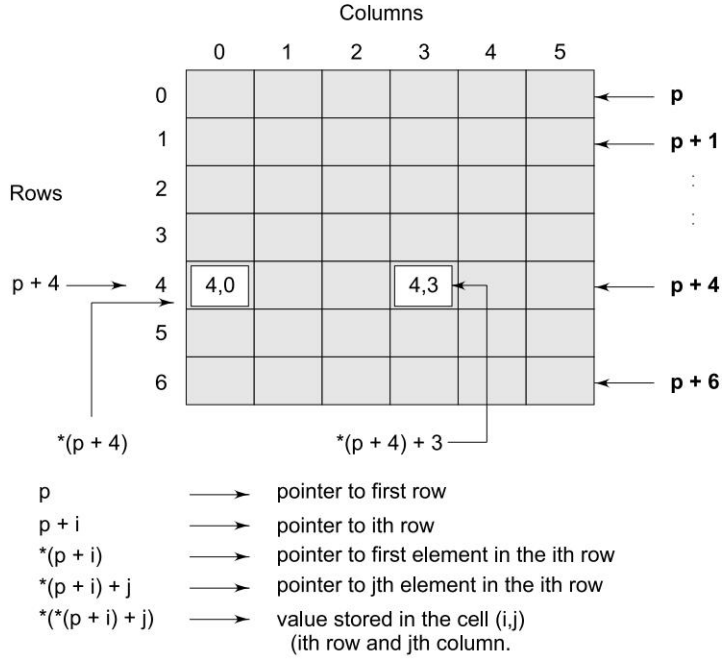
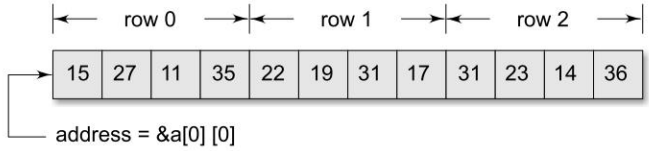


Fig. 11.9 *Pointers to two-dimensional arrays*

Figure 11.9 illustrates how this expression represents the element **a[i][j].** The base address of the array **a** is &a[0][0] and starting at this address, the compiler allocates contiguous space for all the elements *row-wise*. That is, the first element of the second row is placed immediately after the last element of the first row, and so on. Suppose we declare an array **a** as follows:

```
int a[3][4] = { {15,27,11,35},
                {22,19,31,17},
                {31,23,14,36}
                };
```

The elements of **a** will be stored as:



If we declare **p** as an **int** pointer with the initial address of &a[0][0], then

a[i][j] is equivalent to *(p+4 × i+j)

You may notice that, if we increment **i** by 1, the **p** is incremented by 4, the size of each row. Then the element **a[2][3]** is given by ***(p+2 × 4+3) = *(p+11).**

This is the reason why, when a two-dimensional array is declared, we must specify the size of each row so that the compiler can determine the correct storage mapping.

11.11 Pointers and Character Strings

We have seen in chapter of that strings are treated like character arrays and therefore, they are declared and initialized as follows:

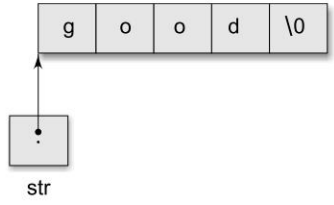
```
char str [5] = "good";
```

The compiler automatically inserts the null character '\0' at the end of the string. C supports an alternative method to create strings using pointer variables of type **char.** Example:

```
char *str = "good";
```

This creates a string for the literal and then stores its address in the pointer variable **str.**

The pointer **str** now points to the first character of the string "good" as:



We can also use the run-time assignment for giving values to a string pointer. Example

```
char * string1;
string1 = "good";
```

Note that the assignment

```
string1 = "good";
```

is not a string copy, because the variable **string1** is a pointer, not a string.

We can print the content of the string **string1** using either **printf** or **puts** functions as follows:

```
printf("%s", string1);
puts (string1);
```

Remember, although **string1** is a pointer to the string, it is also the name of the string. Therefore, we do not need to use indirection operator * here.

Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated by Example 11.5.

Example 11.5 Write a program using pointers to determine the length of a character string.

A program to count the length of a string is shown in Fig.11.10. The statement

```
char *cptr = name;
```

declares **cptr** as a pointer to a character and assigns the address of the first character of **name** as the initial value. Since a string is always terminated by the null character, the statement

```
while(*cptr != '\0')
```

is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr – name;
```

gives the length of the string **name.**



The output also shows the address location of each character. Note that each character occupies one memory cell (byte).

Program

```
main()
{
   char *name;
   int length;
   char *cptr = name;
   name = "DELHI";
   printf ("%s\n", name);
   while(*cptr != '\0')
   {
      printf("%c is stored at address %u\n", *cptr, cptr);
      cptr++;
   }
```

Contd.

```
    length = cptr - name;
    printf("\nLength of the string = %d\n", length);
}
```

Output

```
DELHI
D is stored at address 54
E is stored at address 55
L is stored at address 56
H is stored at address 57
I is stored at address 58
Length of the string = 5
```

Fig. 11.10 *String handling by pointers*

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

```
char *name;
name = "Delhi";
```

These statements will declare **name** as a pointer to character and assign to **name** the constant character string "Delhi". You might remember that this type of assignment does not apply to character arrays. The statements like

```
char name[20];
name = "Delhi";
```

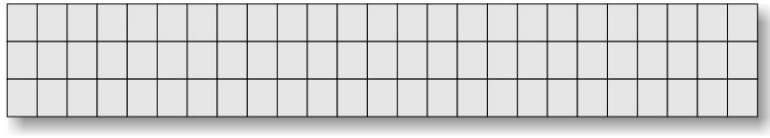
do not work.

11.12 Array of Pointers

One important use of pointers is in handling of a table of strings. Consider the following array of strings:

```
char name [3][25];
```

This says that the **name** is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the **name** table are 75 bytes.



We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. For example,

```
char *name[3] = {
                    "New Zealand",
                    "Australia",
                    "India"
                };
```

declares **name** to be an *array of three pointers* to characters, each pointer pointing to a particular name as:

name [0] ⟶ New Zealand
name [1] ⟶ Australia
name [2] ⟶ India

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	i	a	\0						

The following statement would print out all the three names:

```
for(i = 0; i <= 2; i++)
  printf("%s\n", name[i]);
```

To access the jth character in the ith name, we may write as

```
*(name[i]+j)
```

The character arrays with the rows of varying length are called 'ragged arrays' and are better handled by pointers.

Remember the difference between the notations ***p[3]** and **(*p)[3].** Since * has a lower precedence than [ ], *p[3] declares p as an array of 3 pointers while (*****p)[3] declares **p** as a pointer to an array of three elements.

11.13 Pointers as Function Arguments

We have seen earlier that when an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. If **x** is an array, when we call **sort(x),** the address of **x[0]** is passed to the function **sort**. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal fashion. We used this method when discussing functions that return multiple values.

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as '*call by reference'*. (You know, the process of passing the actual value of variables is known as "call by value".) The function which is called by 'reference' can change the value of the variable used in the call.

Consider the following code:

```
main()
{
        int x;
        x = 20;
        change(&x);  /* call by reference or address */
        printf("%d\n",x);
}
change(int *p)
```

```
{
        *p = *p + 10;
}
```

When the function **change( )** is called, the address of the variable **x**, not its value, is passed into the function **change().** Inside **change( ),** the variable **p** is declared as a pointer and therefore **p** is the address of the variable **x.** The statement,

```
*p = *p + 10;
```

means 'add 10 to the value stored at the address **p'**. Since **p** represents the address of **x,** the value of **x** is changed from 20 to 30. Therefore, the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function. Note that this mechanism is also known as "*call by address*" or "*pass by pointers*"

Note *C99 adds a new qualifier* ***restrict*** *to the pointers passed as function parameters.*

Example 11.6 Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig. 11.11 shows how the contents of two locations can be exchanged using their address locations. The function **exchange()** receives the addresses of the variables **x** and **y** and exchanges their contents.

```
Program
  void exchange (int *, int *);    /* prototype */
  main()
  {
      int x, y;
      x = 100;
      y = 200;
      printf("Before exchange : x = %d y = %d\n\n", x, y);
      exchange(&x,&y); /* call */
      printf("After exchange : x = %d y = %d\n\n", x, y);
  }
  exchange (int *a, int *b)
  {
      int t;
      t = *a;     /* Assign the value at address a to t */
      *a = *b;    /* put b into a */
      *b = t;     /* put t into b */
  }
Output
   Before exchange   : x = 100 y = 200
   After exchange    : x = 200 y = 100
```

Fig. 11.11 *Passing of pointers as function parameters*

You may note the following points:

1. The function parameters are declared as pointers.
2. The dereferenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

The use of pointers to access array elements is very common in C. We have used a pointer to traverse array elements in Example 11.4. We can also use this technique in designing user-defined functions discussed in Chapter 10. Let us consider the problem sorting an array of integers discussed in Example 10.6.

The function **sort** may be written using pointers (instead of array indexing) as shown:

```
void sort (int m, int *x)
{  int i j, temp;
   for (i=1; i<= m-1; i++)
      for (j=1; j<= m-1; j++)
      if (*(x+j-1) >= *(x+j))
      {
         temp = *(x+j-   1);
         *(x+j-1) = *(x+j);
         *(x+j) = temp;
      }
}
```

Note that we have used the pointer x (instead of array x[ ]) to receive the address of array passed and therefore the pointer x can be used to access the array elements (as pointed out in Section 11.10). This function can be used to sort an array of integers as follows:

```
                . . . . .
int score[4] = {45, 90, 71, 83};
                . . . . .
sort(4, score);                /* Function call */
                . . . . .
```

The calling function must use the following prototype declaration.

```
void sort (int, int *);
```

This tells the compiler that the formal argument that receives the array is a pointer, not array variable.

Pointer parameters are commonly employed in string functions. Consider the function copy which copies one string to another.

```
copy(char *s1, char *s2)
{
   while( (*s1++ = *s2++) != '\0')
   ;
}
```

This copies the contents of **s2** into the string **s1.** Parameters **s1** and **s2** are the pointers to character strings, whose initial values are passed from the calling function. For example, the calling statement

```
copy(name1, name2);
```

will assign the address of the first element of **name1** to **s1** and the address of the first element of **name2** to **s2.**

Note that the value of ***s2++** is the character that **s2** pointed to before **s2** was incremented. Due to the postfix ++, **s2** is incremented only after the current value has been fetched. Similarly, **s1** is incremented only after the assignment has been completed.

Each character, after it has been copied, is compared with '\0' and therefore copying is terminated as soon as the '\0' is copied.

11.14 Functions Returning Pointers

We have seen so far that a function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in C, we can also force a function to return a pointer to the calling function. Consider the following code:

```
int *larger (int *, int *);        /* prototype */
main ( )
{
       int a = 10;
       int b = 20;
       int *p;
       p = larger(&a, &b);    /Function call */
       printf ("%d", *p);
}
int *larger (int *x, int *y)
{
       if (*x>*y)
          return (x);  / *address of a */
       else
          return (y);  /* address of b */
}
```

The function **larger** receives the addresses of the variables **a** and **b**, decides which one is larger using the pointers **x** and **y** and then returns the address of its location. The returned value is then assigned to the pointer variable **p** in the calling function. In this case, the address of **b** is returned and assigned to **p** and therefore the output will be the value of **b**, namely, 20.

Note that the address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

11.15 Pointers to Functions

A function, like a variable, has a type and an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type (*fptr) ();
```

This tells the compiler that **fptr** is a pointer to a function, which returns *type* value. The parentheses around ***fptr** are necessary. Remember that a statement like

```
type *gptr();
```

would declare **gptr** as a function returning a pointer to *type*.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer. For example, the statements

```
double mul(int, int);
double (*p1)();
p1 = mul;
```

declare **p1** as a pointer to a function and **mul** as a function and then make **p1** to point to the function **mul**. To call the function **mul,** we may now use the pointer **p1** with the list of parameters. That is,

(*p1)(x,y) /* Function call */

is equivalent to

mul(x,y)

Note the parentheses around ***p1.**

Example 11.7 Write a program that uses a function pointer as a function argument.

A program to print the function values over a given range of values is shown in Fig. 11.12. The printing is done by the function **table** by evaluating the function passed to it by the **main.**

With **table,** we declare the parameter **f** as a pointer to a function as follows:

```
double (*f)();
```

The value returned by the function is of type **double.** When **table** is called in the statement

```
table (y, 0.0, 2, 0.5);
```

we pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list.

During the execution of **table,** the statement

```
value = (*f)(a);
```

calls the function **y** which is pointed to by **f,** passing it the parameter **a.** Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

```
table (cos, 0.0, PI, 0.5);
```

passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

Program

```
#include <math.h>
#define PI 3.1415926
double y(double);
double cos(double);
double table (double(*f)(), double, double, double);
main()
{  printf("Table of y(x) = 2*x*x-x+1\n\n");
   table(y, 0.0, 2.0, 0.5);
   printf("\nTable of cos(x)\n\n");
```

Cont'd

```
    table(cos, 0.0, PI, 0.5);
}
double table(double(*f)(),double min, double max, double step)
{  double a, value;
   for(a = min; a <= max; a += step)
   {
      value = (*f)(a);
      printf("%5.2f %10.4f\n", a, value);
   }
}
double y(double x)
{
   return(2*x*x-x+1);
}
```

Output

```
        Table of y(x) = 2*x*x-x+1
           0.00      1.0000
           0.50      1.0000
           1.00      2.0000
           1.50      4.0000
           2.00      7.0000
        Table of cos(x)
           0.00      1.0000
           0.50      0.8776
           1.00      0.5403
           1.50      0.0707
           2.00     -0.4161
           2.50     -0.8011
           3.00     -0.9900
```

Fig. 11.12 *Use of pointers to functions*

Compatibility and Casting

A variable declared as a pointer is not just a *pointer type* variable. It is also a pointer to a *specific* fundamental data type, such as a character. A pointer therefore always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as *incompatibility* of pointers.

All the pointer variables store memory addresses, which are compatible, but what is not compatible is the underlying data type to which they point to. We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using **cast** operator, as we do with the fundamental types. Example:

```
int x;
char *p;
p = (char *) & x;
```

In such cases, we must ensure that all operations that use the pointer **p** must apply casting properly.

Contd.

We have an exception. The exception is the void pointer (void *). The void pointer is a ***generic pointer*** that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

```
void *vp;
```

Remember that since a void pointer has no object type, it cannot be de-referenced.

11.16 Pointers and Structures

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
    char   name[30];
    int number;
    float  price;
} product[2], *ptr;
```

This statement declares **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to data objects of the type **struct inventory.** The assignment

```
ptr = product;
```

would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product[0].** Its members can be accessed using the following notation.

```
ptr -> name
ptr -> number
ptr -> price
```

The symbol –> is called the *arrow operator* (also known as *member selection operator*) and is made up of a minus sign and a greater than sign. Note that **ptr–>** is simply another way of writing **product[0].**

When the pointer **ptr** is incremented by one, it is made to point to the next record, i.e., product[1]. The following **for** statement will print the values of members of all the elements of **product** array.

```
for(ptr = product; ptr < product+2; ptr++)
        printf ("%s %d %f\n", ptr->name, ptr->number, ptr->price);
```

We could also use the notation

```
(*ptr).number
```

to access the member **number.** The parentheses around ***ptr** are necessary because the member operator '.' has a higher precedence than the operator *.

Example 11.8 Write a program to illustrate the use of structure pointers.

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 11.13. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent)** is also used as the loop control index in **for** loops.

Program

```
struct invent
{
   char *name[20];
   int number;
   float price;
};
main()
{
   struct invent product[3], *ptr;
   printf("INPUT\n\n");
   for(ptr = product; ptr < product+3; ptr++)
      scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);
   printf("\nOUTPUT\n\n");
    ptr = product;
    while(ptr < product + 3)
    {
      printf("%-20s %5d %10.2f\n",
                ptr->name,
                ptr->number,
                ptr->price);
      ptr++;
   }
}
```

Output

```
INPUT
Washing_machine 5      7500
Electric_iron      12     350
Two_in_one          7     1250

OUTPUT
Washing machine    5      7500.00
Electric_iron      12      350.00
Two_in_one         7      1250.00
```

Fig. 11.13 *Pointer to structure variables*

While using structure pointers, we should take care of the precedence of operators.

The operators '–>' and '**.**', and () and [] enjoy the highest priority among the operators. They bind very tightly with their operands. For example, given the definition

```
struct
{
   int count;
   float *p;      /* pointer inside the struct */
}  ptr;           /* struct type pointer */
```

then the statement

```
++ptr->count;
```

increments **count,** not **ptr.** However,

```
(++ptr)->count;
```

increments **ptr** first, and then links **count.** The statement

```
ptr++ -> count;
```

is legal and increments **ptr** after accessing **count.**

The following statements also behave in the similar fashion.

***ptr–>p**	Fetches whatever **p** points to.
***ptr–>p++**	Increments **p** after accessing whatever it points to.
(*ptr–>p)++	Increments whatever **p** points to.
***ptr++–>p**	Increments **ptr** after accessing whatever it points to.

In the previous chapter, we discussed about passing of a structure as an argument to a function. We also saw an example where a function receives a copy of an entire structure and returns it after working on it. As we mentioned earlier, this method is inefficient in terms of both, the execution speed and memory. We can overcome this drawback by passing a pointer to the structure and then using this pointer to work on the structure members. Consider the following function:

```
print_invent(struct invent *item)
   {
      printf("Name: %s\n", item->name);
      printf("Price: %f\n", item->price);
   }
```

This function can be called by

```
print_invent(&product);
```

The formal argument **item** receives the address of the structure **product** and therefore it must be declared as a pointer of type **struct invent**, which represents the structure of **product.**

11.17 Troubles with Pointers

Pointers give us tremendous power and flexibility. However, they could become a nightmare when they are not used correctly. The major problem with wrong use of pointers is that the complier may not detect the error in most cases and therefore the program is likely to produce unexpected results. The output may not given us any clue regarding the use of a bad pointer. Debugging therefore becomes a difficult task.

We list here some pointer errors that are more commonly committed by the programmers.

- Assigning values to uninitialized pointers

```
int * p, m = 100;
*p = m;                 /* Error */
```

- Assigning value to a pointer variable

```
int *p, m = 100;
p = m;              /* Error */
```

- Not dereferencing a pointer when required

```
int *p, x = 100;
p = &x;
printf("%d, p");        /* Error */
```

- Assigning the address of an uninitialized variable

```
int m, *p
p = &m;                    /* Error */
```

- Comparing pointers that point to different objects

```
char name1 [20], name2 [30];
     char *p1 = name1;
     char *p2 = name2;
     if 9p1 > p20 ......          /* Error */
```

We must be careful in declaring and assigning values to pointers correctly before using them. We must also make sure that we apply the address operator & and referencing operator * correctly to the pointers. That will save us from sleepless nights.

Just Remember

- Only an address of a variable can be stored in a pointer variable.
- Do not store the address of a variable of one type into a pointer variable of another type.
- The value of a variable cannot be assigned to a pointer variable.
- A pointer variable contains garbage until it is initialized. Therefore we must not use a pointer variable before it is assigned, the address of a variable.
- Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.
- If we want a called function to change the value of a variable in the calling function, we must pass the address of that variable to the called function.
- When we pass a parameter by address, the corresponding formal parameter must be a pointer variable.
- It is an error to assign a numeric constant to a pointer variable.
- It is an error to assign the address of a variable to a variable of any basic data types.
- It is an error to assign a pointer of one type to a pointer of another type without a cast (with an exception of void pointer).
- A proper understanding of a precedence and associativity rules is very important in pointer applications. For example, expressions like *p++, *p[ ], (*p)[ ], (p).member should be carefully used.
- When an array is passed as an argument to a function, a pointer is actually passed. In the header function, we must declare such arrays with proper size, except the first, which is optional.
- A very common error is to use (or not to use) the address operator (&) and the indirection operator (*) in certain places. Be careful. The compiler may not warn such mistakes.

Review Questions

11.1 State whether the following statements are *true* or *false*.

(a) Pointer constants are the addresses of memory locations.
(b) Pointer variables are declared using the address operator.

(c) The underlying type of a pointer variable is void.
(d) Pointers to pointers is a term used to describe pointers whose contents are the address of another pointer.
(e) It is possible to cast a pointer to float as a pointer to integer.
(f) An integer can be added to a pointer.
(g) A pointer can never be subtracted from another pointer.
(h) When an array is passed as an argument to a function, a pointer is passed.
(i) Pointers cannot be used as formal parameters in headers to function definitions.
(j) Value of a local variable in a function can be changed by another function.

11.2 Fill in the blanks in the following statements:

(a) A pointer variable contains as its value the _____ of another variable.
(b) The _____ operator is used with a pointer to de-reference the address contained in the pointer.
(c) The _____ operator returns the value of the variable to which its operand points.
(d) The only integer that can be assigned to a pointer variable is _____.
(e) The pointer that is declared as _____ cannot be de-referenced.

11.3 What is a pointer?

11.4 How is a pointer initialized?

11.5 Explain the effects of the following statements:

(a) `int a, *b = &a;`
(b) `int p, *p;`
(c) `char *s;`
(d) `a = (float *) &x);`
(e) `double(*f)();`

11.6 If **m** and **n** have been declared as integers and **p1** and **p2** as pointers to integers, then state errors, if any, in the following statements.

(a) `p1 = &m;`
(b) `p2 = n;`
(c) `*p1 = &n;`
(d) `p2 = &*&m;`
(e) `m = p2-p1;`
(f) `p1 = &p2;`
(g) `m = *p1 + *p2++;`

11.7 Distinguish between (*m)[5] and *m[5].

11.8 Find the error, if any, in each of the following statements:

(a) `int x = 10;`
(b) `int *y = 10;`
(c) `int a, *b = &a;`
(d)
```
int m;
int **x = &m;
```

11.9 Given the following declarations:

```
int x = 10, y = 10;
int *p1 = &x, *p2 = &y;
```

What is the value of each of the following expressions?

(a) (*p1) ++

(b) -- (*p2)

(c) *p1 + (*p2) --

(d) + + (*p2) – *p1

11.10 Describe typical applications of pointers in developing programs.

11.11 What are the arithmetic operators that are permitted on pointers?

11.12 What is printed by the following program?

```
int m = 100';
int * p1 = &m;
int **p2 = &p1;
printf("%d", **p2);
```

11.13 What is wrong with the following code?

```
int **p1, *p2;
p2 = &p1;
```

11.14 Assuming **name** as an array of 15 character length, what is the difference between the following two expressions?

(a) name + 10; and

(b) *(name + 10).

11.15 What is the output of the following segment?

int m[2];

```
*(m+1) = 100;
*m = *(m+1);
printf("%d", m [0]);
```

11.16 What is the output of the following code?

```
int m [2];
int *p = m;
m [0] = 100 ;
m [1] = 200 ;
printf("%d %d", ++*p, *p);
```

11.17 What is the output of the following program?

```
int f(char *p);
main ( )
{
       char str[ ] = "ANSI";
       printf("%d", f(str) );
}
int f(char *p)
{
```

```
        char *q = p;
        while (*++p)
              ;
        return (p-q);
}
```

11.18 Given below are two different definitions of the function **search( )**

(a) void search (int* m[ ], int x)
{
}

(b) void search (int ** m, int x)
{
}

Are they equivalent? Explain.

11.19 Do the declarations

```
char s [ 5 ] ;
char *s;
represent the same? Explain.
```

11.20 Which one of the following is the correct way of declaring a pointer to a function? Why?

(a) int (*p) (void) ;
(b) int *p (void);

Programming Exercises

11.1 Write a program using pointers to read in an array of integers and print its elements in reverse order.

11.2 We know that the roots of a quadratic equation of the form

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

are given by the following equations:

$$x_1 = \frac{-b + \text{square} - \text{root}(b^2 - 4ac)}{2a}$$

$$x_2 = \frac{-b - \text{square} - \text{root}(b^2 - 4ac)}{2a}$$

Write a function to calculate the roots. The function must use two pointer parameters, one to receive the coefficients a, b, and c, and the other to send the roots to the calling function.

11.3 Write a function that receives a sorted array of integers and an integer value, and inserts the value in its correct place.

11.4 Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.

11.5 Using pointers, write a function that receives a character string and a character as argument and deletes all occurrences of this character in the string. The function should return the corrected string with no holes.

11.6 Write a function **day_name** that receives a number n and returns a pointer to a character string containing the name of the corresponding day. The day names should be kept in a **static** table of character strings local to the function.

11.7 Write a program to read in an array of names and to sort them in alphabetical order. Use **sort** function that receives pointers to the functions **strcmp** and **swap.sort** in turn should call these functions via the pointers.

11.8 Given an array of sorted list of integer numbers, write a function to search for a particular item, using the method of *binary search*. And also show how this function may be used in a program. Use pointers and pointer arithmetic.

(**Hint:** In binary search, the target value is compared with the array's middle element. Since the table is sorted, if the required value is smaller, we know that all values greater than the middle element can be ignored. That is, in one attempt, we eliminate one half the list. This search can be applied recursively till the target value is found.)

11.9 Write a function (using a pointer parameter) that reverses the elements of a given array.

11.10 Write a function (using pointer parameters) that compares two integer arrays to see whether they are identical. The function returns 1 if they are identical, 0 otherwise.

Debugging Exercises

11.1 Identify the error in the following structure declaration:

```
struct employee
 {
  int emp_id;
  struct name
  {
   char First[20];
   char Middle[20];
   char Last[20];
  }emp_name;
  char doj[20];
  struct G_Sal
  {
   float basic;
   float hra;
   float spl_allow;
  }emp_sal;
 };
 struct employee emp1;
```

11.2 Identify the error in the following program, if any:

```
#include <stdio.h>

void main ()
{
 struct s
 {
  int a;
  char b;
  float c;
```

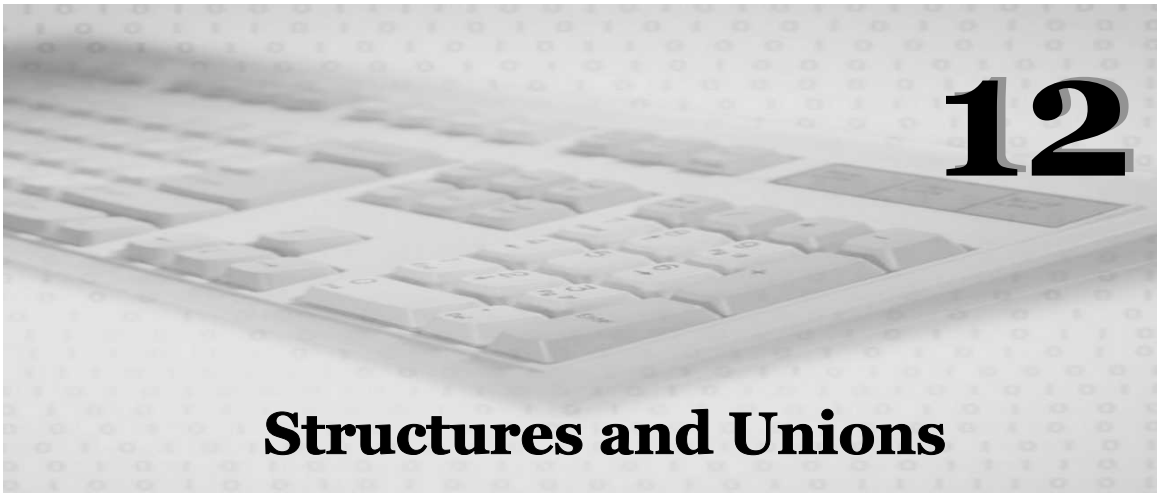
```
  long d;
 }s1;
printf("\nSize of structure s = %d",sizeof(s));
}
```

11.3 Identify the error in the following code, if any:

```
.
.
struct A
{
 int member;
}*Aptr;
Aptr.member=5;
.
.
```

11.4 Identify the error in the following structure declaration, if any:

```
struct node
{
 int node_value;
 struct node *ptr;
};
```



12 Structures and Unions

12.1 Introduction

We have seen that arrays can be used to represent a group of data items that belong to the same type, such as **int** or **float**. However, we cannot use an array if we want to represent a collection of data items of different types using a single name. Fortunately, C supports a constructed data type known as *structures,* a mechanism for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks. The concept of a structure is analogous to that of a 'record' in many other languages. More examples of such structures are:

time	:	seconds, minutes, hours
date	:	day, month, year
book	:	author, title, price, year
city	:	name, country, population
address	:	name, door-number, street, city
inventory	:	item, stock, value
customer	:	name, telephone, city, category

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design. This chapter is devoted to the study of structures and their applications in program development. Another related concept known as *unions* is also discussed.

12.2 Defining a Structure

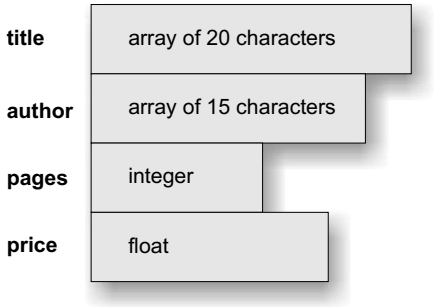
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of

structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book_bank
{
    char    title[20];
    char    author[15];
    int     pages;
    float   price;
};
```

The keyword **struct** declares a structure to hold the details of four data fields, namely **title, author, pages**, and **price**. These fields are called *structure elements* or *members.* Each member may belong to a different type of data. **book_bank** is the name of the structure and is called the *structure tag*. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above definition has not declared any variables. It simply describes a format called *template* to represent information as shown below:



The general format of a structure definition is as follows:

```
struct          tag_name
{
    data_type               member1;
    data_type               member2;
       ----                 ----
       ----                 ----
};
```

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as **book_bank** can be used to declare structure variables of its type, later in the program.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

Contd.

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

12.3 Declaring Structure Variables

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements:

1. The keyword **struct.**
2. The structure tag name.
3. List of variable names separated by commas.
4. A terminating semicolon.

For example, the statement

```
struct book_bank, book1, book2, book3;
```

declares **book1, book2**, and **book3** as variables of type **struct book_bank.**

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

```
struct book_bank
{
   char   title[20];
   char   author[15];
   int    pages;
   float  price;
};
struct book_bank book1, book2, book3;
```

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **book1.** When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration

```
struct book_bank
{
    char title[20];
    char author[15];
    int pages;
    flat price;
} book1, book2, book3;
```

is valid. The use of tag name is optional here. For example:

```
struct
{   ........
    ........
    ........
}   book1, book2, book3;
```

declares **book1, book2,** and **book3** as structure variables representing three books, but does not include a tag name. However, this approach is not recommended for two reasons.

1. Without a tag name, we cannot use it for future declarations:
2. Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the **main**, along with macro definitions, such as **#define.** In such cases, the definition is *global* and can be used by other functions as well.

Type-Defined Structures

We can use the keyword **typedef** to define a structure as follows:

```
typedef struct
{ . . . . .
    type member1;
    type member2;
    . . . . .
    . . . . .
} type_name;
```

The type_name represents structure definition associated with it and therefore can be used to declare structure variables as shown below:

```
type_name variable1, variable2, . . . . . ;
```

Remember that (1) the name *type_name* is the type definition name, not a variable and (2) we cannot define a variable with *typedef* declaration.

12.4 Accessing Structure Members

We can access and assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word **title,** has no meaning whereas the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the *member operator* '.' which is also known as 'dot operator' or 'period operator'. For example,

```
book1.price
```

is the variable representing the price of **book1** and can be treated like any other ordinary variable. Here is how we would assign values to the members of **book1**:

```
strcpy(book1.title, "BASIC");
strcpy(book1.author, "Balagurusamy");
book1.pages = 250;
book1.price = 120.50;
```

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s\n", book1.title);
scanf("%d\n", &book1.pages);
```

are valid input statements.

Example 12.1 Defisne a **structure type**, struct personal that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

Structure definition along with the program is shown in Fig. 12.1. The **scanf** and **printf** functions illustrate how the member operator '.' is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

Program

```
struct  personal
{
      char  name[20];
      int   day;
      char  month[10];
      int   year;
      float salary;
};
main()
{
        struct personal person;
      printf("Input Values\n");
      scanf("%s %d %s %d %f",
                       person.name,
                      &person.day,
                       person.month,
                      &person.year,
                      &person.salary);
   printf("%s %d %s %d %f\n",
                      person.name,
                      person.day,
                      person.month,
                      person.year,
                      person.salary);
}
```

Output

```
Input Values
   M.L.Goel 10 January 1945 4500
   M.L.Goel 10 January 1945 4500.00
```

Fig. 12.1 *Defining and accessing structure members*

12.5 Structure Initialization

Like any other data type, a structure variable can be initialized at compile time.

```
main()
{
   struct
   {
      int weight;
      float height;
   }
   student = {60, 180.75};
  .....
  .....
}
```

This assigns the value 60 to **student. weight** and 180.75 to **student. height.** There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
struct st_record
{
   int weight;
   float height;
};
struct st_record student1 = { 60, 180.75 };
struct st_record student2 = { 53, 170.60 };
.....
.....
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record
{
   int weight;
   float height;
}  student1 = {60, 180.75};
main()
{
   struct st_record student2 = {53, 170.60};
   .....
   .....
}
```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

Note that the compile-time initialization of a structure variable must have the following elements:

1. The keyword **struct**.
2. The structure tag name.
3. The name of the variable to be declared.
4. The assignment operator =.
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

Rules for Initializing Structures

There are a few rules to keep in mind while initializing structure variables at compile-time.

1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:
   - Zero for integer and floating point numbers.
   - '\0' for characters and strings.

12.6 Copying and Comparing Structure Variables

Two variables of the same structure type can be copied the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following statements are valid:

```
person1 = person2;
person2 = person1;
```

However, the statements such as

```
person1 == person2
person1 != person2
```

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

Example 12.2 Write a program to illustrate the comparison of structure variables.

The program shown in Fig. 12.2 illustrates how a structure variable can be copied into another of the same type. It also performs member-wise comparison to decide whether two structure variables are identical.

```
program
     struct class
     {
           int  number;
           char name[20];
           float marks;
     };
```

Contd.

```
main()
{
        int  x;
        struct class student1 = {111,"Rao",72.50};
        struct class student2 = {222,"Reddy", 67.00};
        struct class student3;

        student3 = student2;

        x = ((student3.number ==  student2.number) &&
             (student3.marks  ==  student2.marks)) ? 1 : 0;

        if(x == 1)
        {
           printf("\nstudent2 and student3 are same\n\n");
           printf("%d %s %f\n",  student3.number,
                                 student3.name,
                                 student3.marks);
        }
        else
            printf("\nstudent2 and student3 are different\n\n");

}
```

Output

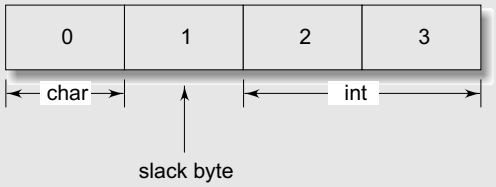
```
student2 and student3 are same

222 Reddy 67.000000
```

Fig. 12.2 *Comparing and copying structure variables*

Word Boundaries and Slack Bytes

Computer stores structures using the concept of "word boundary". The size of a word boundary is machine dependent. In a computer with two bytes word boundary, the members of a structure are stored left_aligned on the word boundary, as shown below. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as the *slack byte.*



When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined. Due to this, even if the members of two variables are equal, their

Contd.

structures do not necessarily compare equal. C, therefore, does not permit comparison of structures. However, we can design our own function that could compare individual members to decide whether the structures are equal or not.

12.7 Operations on Individual Members

As pointed out earlier, the individual members are identified using the member operator, the *dot*. A member with the *dot operator* along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators. Consider the program in Fig. 12.2. We can perform the following operations:

```
if (student1.number == 111)
   student1.marks += 10.00;
float sum = student1.marks + student2.marks;
student2.marks * = 0.5;
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

```
student1.number ++;
++ student1.number;
```

The precedence of the *member* operator is higher than all *arithmetic* and *relational* operators and therefore no parentheses are required.

Three Ways to Access Members

We have used the dot operator to access the members of structure variables. In fact, there are two other ways. Consider the following structure:

```
typedef struct
{
       int x;
       int y;
} VECTOR;
VECTOR v, *ptr;
ptr = & v;
```

The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable n. Now, the members can be accessed in three ways:

- using dot notation : v**.**x
- using indirection notation : (*ptr)**.**x
- using selection notation : ptr –> x

The second and third methods are considered in Chapter 11.

12.8 Arrays of Structures

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example:

```
struct class student[100];
```

defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class.** Consider the following declaration:

```
struct marks
{
   int  subject1;
   int  subject2;
   int  subject3;
};
main()
{
   struct marks student[3] =
      {{45,68,81}, {75,53,69}, {57,36,71}};
```

This declares the **student** as an array of three elements **student[0], student[1],** and **student[2]** and initializes their members as follows:

```
student[0].subject1 = 45;
student[0].subject2 = 65;
   ....
   ....
student[2].subject3 = 71;
```

Note that the array is declared just as it would have been with any other array. Since **student** is an array, we use the usual array-accessing methods to access individual elements and then the member operator to access members. Remember, each element of **student** array is a structure variable with three members.

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array **student** actually looks as shown in Fig. 12.3.

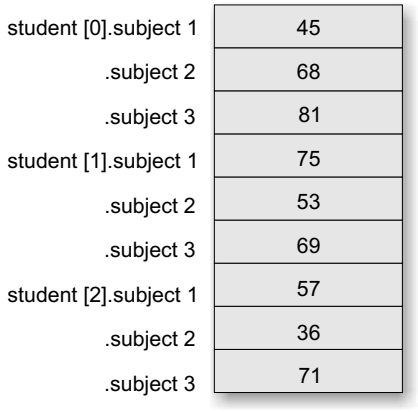


Fig. 12.3 *The array student inside memory*

Example 12.3 For the **student** array discussed above, write a program to calculate the subject-wise and student-wise totals and store them as a part of the structure.

The program is shown in Fig. 12.4. We have declared a four-member structure, the fourth one for keeping the student-totals. We have also declared an **array** total to keep the subject-totals and the grand-total. The grand-total is given by **total.total.** Note that a member name can be any valid C name and can be the same as an existing structure variable name. The linked name **total.total** represents the **total** member of the structure variable **total**.

Program

```
  struct marks
     {
           int  sub1;
           int  sub2;
           int  sub3;
           int  total;
     };
     main()
     {
           int  i;
           struct marks student[3] =  {{45,67,81,0},
                                       {75,53,69,0},
                                       {57,36,71,0}};
           struct marks total;
           for(i = 0; i <= 2; i++)
           {
               student[i].total = student[i].sub1 +
                                  student[i].sub2 +
                                  student[i].sub3;
              total.sub1 = total.sub1 + student[i].sub1;
              total.sub2 = total.sub2 + student[i].sub2;
              total.sub3 = total.sub3 + student[i].sub3;
              total.total = total.total + student[i].total;
        }
        printf(" STUDENT          TOTAL\n\n");
        for(i = 0; i <= 2; i++)
          printf("Student[%d]      %d\n", i+1,student[i].total);
        printf("\n SUBJECT          TOTAL\n\n");
        printf("%s       %d\n%s       %d\n%s       %d\n",
              "Subject 1   ", total.sub1,
              "Subject 2   ", total.sub2,
              "Subject 3   ", total.sub3);

       printf("\nGrand Total = %d\n", total.total);
     }
```

Output

```
        STUDENT           TOTAL
        Student[1]         193
```

Contd.

```
Student[2]          197

Student[3]          164

SUBJECT            TOTAL
Subject 1           177
Subject 2           156
Subject 3           221

Grand Total  = 554
```

Fig. 12.4 *Arrays of structures: Illustration of subscripted structure variables*

12.9 Arrays within Structures

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single-dimensional or multi-dimensional arrays of type **int** or **float.** For example, the following structure declaration is valid:

```
struct marks
{
   int number;
   float subject[3];
}  student[2];
```

Here, the member **subject** contains three elements, **subject[0], subject[1]** and **subject[2].** These elements can be accessed using appropriate subscripts. For example, the name

```
student[1].subject[2];
```

would refer to the marks obtained in the third subject by the second student.

Example 12.4 Rewrite the program of Example 12.3 using an array member to represent the three subjects.

The modified program is shown in Fig. 12.5. You may notice that the use of array name for subjects has simplified in code.

Program

```
main()
{
    struct  marks
    {
        int  sub[3];
        int  total;
    };
    struct marks student[3] =
    {45,67,81,0,75,53,69,0,57,36,71,0};
```

Contd.

```
        struct marks total;
        int  i,j;

        for(i = 0; i <= 2; i++)
        {
           for(j = 0; j <= 2; j++)
           {
              student[i].total += student[i].sub[j];
              total.sub[j] += student[i].sub[j];
           }
           total.total += student[i].total;
        }
        printf("STUDENT          TOTAL\n\n");
        for(i = 0; i <= 2; i++)
           printf("Student[%d]      %d\n", i+1, student[i].total);

        printf("\nSUBJECT          TOTAL\n\n");
        for(j = 0; j <= 2; j++)
           printf("Subject-%d        %d\n", j+1, total.sub[j]);

        printf("\nGrand Total  =   %d\n", total.total);
    }
```

Output

```
    STUDENT          TOTAL
    Student[1]        193
    Student[2]        197
    Student[3]        164

    STUDENT          TOTAL
    Student-1         177
    Student-2         156
    Student-3         221

    Grand Total  =    554
```

Fig. 12.5 *Use of subscripted members arrays in structures*

12.10 Structures within Structures

Structures within a structure means *nesting* of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
   char name;
   char department;
   int  basic_pay;
   int  dearness_allowance;
```

```
    int  house_rent_allowance;
    int  city_allowance;
}
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below:

```
struct salary
{
    char name;
    char department;
    struct
    {
        int dearness;
        int house_rent;
        int city;
    }
    allowance;
}
employee;
```

The salary structure contains a member named **allowance**, which itself is a structure with three members. The members contained in the inner structure namely **dearness, house_rent**, and **city** can be referred to as:

employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following are invalid:

employee.allowance (actual member is missing)
employee.house_rent (inner structure variable is missing)

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
    .....
    struct
    {
        int dearness;
        .....
    }
    allowance,
    arrears;
}
employee[100];
```

The inner structure has two variables, **allowance** and **arrears**. This implies that both of them have the same structure template. Note the comma after the name **allowance.** A base member can be accessed as follows:

employee[1].allowance.dearness
employee[1].arrears.dearness

We can also use tag names to define inner structures. Example:

```
struct pay
   {
      int dearness;
      int house_rent;
      int city;
   };
   struct salary
   {
      char name;
      char department;
      struct pay allowance;
      struct pay arrears;
   };
   struct salary employee[100];
```

pay template is defined outside the **salary** template and is used to define the structure of **allowance** and **arrears** inside the **salary** structure.

It is also permissible to nest more than one type of structures.

```
struct personal_record
{
   struct name_part name;
   struct addr_part address;
   struct date date_of_birth;
   .....
   .....
};
struct personal_record person1;
```

The first member of this structure is **name**, which is of the type **struct name_part.** Similarly, other members have their structure types.

Note *C permits nesting up to 15 levels. However, C99 allows 63 levels of nesting.*

12.11 Structures and Functions

We know that the main philosophy of C language is the use of functions. And therefore, it is natural that C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within

the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.

3. The third approach employs a concept called *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function. This method is more efficient as compared to the second one.

In this section, we discuss in detail the second method, while the third approach using pointers is discussed in the next chapter, where pointers are dealt in detail.

The general format of sending a copy of a structure to the called function is:

```
function_name(structure_variable_name);
```

The called function takes the following form:

```
data_type function_name(struct_type st_name)
{
    ......
    ......
    return(expression);
}
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data back to the calling function. The *expression* may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called functions must be declared in the calling function appropriately.

Example 12.5 Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

A program to update an item is shown in Fig. 12.6. The function **update** receives a copy of the structure variable **item** as one of its parameters. Note that both the function **update** and the formal parameter **product** are declared as type **struct stores.** It is done so because the function uses the parameter **product** to receive the structure variable **item** and also to return the updated values of **item**.

The function **mul** is of type **float** because it returns the product of **price** and **quantity.** However, the parameter **stock,** which receives the structure variable **item** is declared as type **struct stores.**

The entire structure returned by **update** can be copied into a structure of identical type. The statement

```
item = update(item,p_increment,q_increment);
```

replaces the old values of **item** by the new ones.

Program

```
/*        Passing a copy of the entire structure        */
  struct stores
  {
      char  name[20];
      float price;
      int   quantity;
  };
  struct stores update (struct stores product, float p, int q);
  float mul (struct stores stock);
  main()
  {
      float    p_increment, value;
      int      q_increment;

      struct stores item = {"XYZ", 25.75, 12};

      printf("\nInput increment values:");
      printf("   price increment and quantity increment\n");
      scanf("%f %d", &p_increment, &q_increment);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      item  = update(item, p_increment, q_increment);
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      printf("Updated values of item\n\n");
      printf("Name      : %s\n",item.name);
      printf("Price     : %f\n",item.price);
      printf("Quantity  : %d\n",item.quantity);
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      value  = mul(item);
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

      printf("\nValue of the item  =  %f\n", value);
  }
  struct stores update(struct stores product, float p, int q)
  {
      product.price += p;
      product.quantity += q;
      return(product);
  }
  float mul(struct stores stock)
  {
      return(stock.price * stock.quantity);
  }
```

Contd.

```
Output
Input increment values:   price increment and quantity increment
10 12
Updated values of item
Name      : XYZ
Price     : 35.750000
Quantity  : 24
Value of the item  =  858.000000
```

Fig. 12.6 *Using structure as a function parameter*

You may notice that the template of **stores** is defined before **main().** This has made the data type **struct stores** as *global* and has enabled the functions **update** and **mul** to make use of this definition.

12.12 Pointers and Structures

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
    char   name[30];
    int    number;
    float  price;
} product[2], *ptr;
```

This statement declares **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to data objects of the type **struct inventory.** The assignment

```
ptr = product;
```

would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product[0].** Its members can be accessed using the following notation.

```
ptr -> name
ptr -> number
ptr -> price
```

The symbol –> is called the *arrow operator* (also known as *member selection operator*) and is made up of a minus sign and a greater than sign. Note that **ptr–>** is simply another way of writing **product[0].**

When the pointer **ptr** is incremented by one, it is made to point to the next record, i.e., product[1]. The following **for** statement will print the values of members of all the elements of **product** array.

```
for(ptr = product; ptr < product+2; ptr++)
    printf ("%s %d %f\n", ptr->name, ptr->number, ptr->price);
```

We could also use the notation

```
(*ptr).number
```

to access the member **number.** The parentheses around ***ptr** are necessary because the member operator '.' has a higher precedence than the operator *.

Example 12.6 Write a program to illustrate the use of structure pointers.

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 12.7. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent)** is also used as the loop control index in **for** loops.

```
Program
   struct invent
   {
       char  *name[20];
       int   number;
       float price;
   };
   main()
   {
      struct invent product[3], *ptr;
      printf("INPUT\n\n");
      for(ptr = product; ptr < product+3; ptr++)
        scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);
      printf("\nOUTPUT\n\n");
       ptr = product;
       while(ptr < product + 3)
       {
           printf("%-20s %5d %10.2f\n",
                    ptr->name,
                    ptr->number,
                    ptr->price);
           ptr++;
       }
   }
Output
   INPUT
   Washing_machine   5    7500
   Electric_iron    12     350
   Two_in_one        7    1250

   OUTPUT
   Washing machine      5   7500.00
   Electric_iron        12  350.00
   Two_in_one           7   1250.00
```

Fig. 12.7 *Pointer to structure variables*

While using structure pointers, we should take care of the precedence of operators.

The operators '–>' and '**.**', and () and [] enjoy the highest priority among the operators. They bind very tightly with their operands. For example, given the definition

```
struct
{
   int count;
   float *p;     /* pointer inside the struct */
}  ptr;          /* struct type pointer */
```

then the statement

```
++ptr->count;
```

increments **count,** not **ptr.** However,

```
(++ptr)->count;
```

increments **ptr** first, and then links **count.** The statement

```
ptr++ -> count;
```

is legal and increments **ptr** after accessing **count.**

The following statements also behave in the similar fashion.

***ptr–>p**	Fetches whatever **p** points to.
***ptr–>p++**	Increments **p** after accessing whatever it points to.
(*ptr–>p)++	Increments whatever **p** points to.
***ptr++–>p**	Increments **ptr** after accessing whatever it points to.

In the previous chapter, we discussed about passing of a structure as an argument to a function. We also saw an example where a function receives a copy of an entire structure and returns it after working on it. As we mentioned earlier, this method is inefficient in terms of both, the execution speed and memory. We can overcome this drawback by passing a pointer to the structure and then using this pointer to work on the structure members. Consider the following function:

```
print_invent(struct invent *item)
{
        printf("Name: %s\n", item->name);
        printf("Price: %f\n", item->price);
}
```

This function can be called by

```
print_invent(&product);
```

The formal argument **item** receives the address of the structure **product** and therefore it must be declared as a pointer of type **struct invent**, which represents the structure of **product.**

12.13 Unions

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

```
union item
{
   int m;
   float x;
   char c;
}  code;
```

This declares a variable **code** of type **union item.** The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

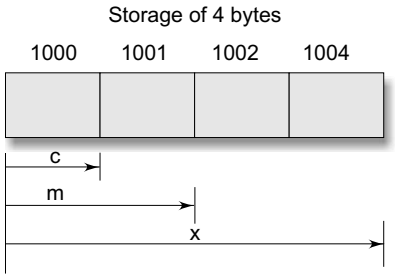


Fig.12.8 *Sharing of a storage locating by union members*

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. Figure 12.8 shows how all the three variables share the same address. This assumes that a float variable requires 4 bytes of storage.

To access a union member, we can use the same syntax that we use for structure members. That is,

code.m
code.x
code.c

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m = 379;
code.x = 7859.36;
printf("%d", code.m);
```

would produce erroneous output (which is machine dependent).

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

Unions may be initialized when the variable is declared. But, unlike structures, it can be initialized only with a value of the same type as the first union member. For example, with the preceding, the declaration

```
union item abc = {100};
```

is valid but the declaration

```
union item abc = {10.75};
```

is invalid. This is because the type of the first member is **int.** Other members can be initialized by either assigning values or reading from the keyboard.

12.14 Size of Structures

We normally use structures, unions, and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The expression

```
sizeof(struct x)
```

will evaluate the number of bytes required to hold all the members of the structure **x.** If **y** is a simple structure variable of type **struct x**, then the expression

```
sizeof(y)
```

would also give the same answer. However, if **y** is an array variable of type **struct x,** then

```
sizeof(y)
```

would give the total number of bytes the array **y** requires.

This kind of information would be useful to determine the number of records in a database. For example, the expression

```
sizeof(y)/sizeof(x)
```

would give the number of elements in the array **y.**

12.15 Bit Fields

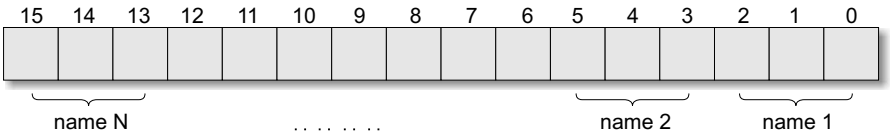
So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small *bit fields* to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits as if it represented an integral quantity.

A *bit field* is a set of adjacent bits whose size can be from 1 to 16 bits in length. A word can therefore be divided into a number of bit fields. The name and size of bit fields are defined using a structure. The general form of bit field definition is:

```
struct tag-name
{
    data-type name1: bit-length;
    data-type name2: bit-length;
    . . . . . .
    . . . . . .
    data-type nameN: bit-length;
}
```

The *data-type* is either **int** or **unsigned int** or **signed int** and the *bit-length* is the number of bits used for the specified name. Remember that a **signed** bit field should have at least 2 bits (one bit for sign). Note that the field name is followed by a colon. The *bit-length* is decided by the range of value to be stored. The largest value that can be stored is 2^{n-1}, where **n** is bit-length.

The internal representation of bit fields is machine dependent. That is, it depends on the size of **int** and the ordering of bits. Some machines store bits from left to right and others from right to left. The sketch below illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.



There are several specific points to observe:

1. The first field always starts with the first bit of the word.
2. A bit field cannot overlap integer boundaries. That is, the sum of lengths of all the fields in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.
3. There can be unnamed fields declared with size. Example:

   Unsigned : *bit-length*

   Such fields provide padding within the word.
4. There can be unused bits in a word.
5. We cannot take the address of a bit field variable. This means we cannot use **scanf** to read values into bit fields. We can neither use pointer to access the bit fields.
6. Bit fields cannot be arrayed.
7. Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behavior would be unpredicted.

Suppose, we want to store and use personal information of employees in compressed form, this can be done as follows:

```
struct personal
{
unsigned sex          :    1
unsigned age          :    7
unsigned m_status     :    1
unsigned children     :    3
unsigned              :    4
} emp;
```

This defines a variable name **emp** with four bit fields. The range of values each field could have is follows:

Bit field	*Bit length*	*Range of value*
sex	1	0 or 1
age	7	0 or 127 ($2^7 - 1$)
m_status	1	0 or 1
children	3	0 to 7 (2^3–1)

Once bit fields are defined, they can be referenced just as any other structure-type data item would be referenced. The following assignment statements are valid.

```
emp.sex = 1;
emp.age = 50;
```

Remember, we cannot use **scanf** to read values into a bit field. We may have to read into a temporary variable and then assign its value to the bit field. For example:

```
scanf(%d %d", &AGE,&CHILDREN);
emp.age = AGE;
emp.children = CHILDREN;
```

One restriction in accessing bit fields is that a pointer cannot be used. However, they can be used in normal expressions like any other variable. For example:

```
sum = sum + emp.age;
if(emp.m_status). . . . .;
printf("%d\n", emp.age);
```

are valid statements.

It is possible to combine normal structure elements with bit field elements. For example:

```
struct personal
{
      char         name[20]; /* normal variable */
      struct addr  address;  /* structure variable */
      unsigned     sex : 1;
      unsigned     age : 7;
      . . . . .
      . . . . .
}
emp[100];
```

This declares **emp** as a 100 element array of type **struct personal.** This combines normal variable name and structure type variable **address** with bit fields.

Bit fields are packed into words as they appear in the definition. Consider the following definition.

```
struct pack
{
      unsigned  a:2;
      int count;
      unsigned  b : 3;
};
```

Here, the bit field **a** will be in one word, the variable **count** will be in the second word and the bit field **b** will be in the third word. The fields **a** and **b** would not get packed into the same word.

Just Remember

- Remember to place a semicolon at the end of definition of structures and unions.
- We can declare a structure variable at the time of definition of a structure by placing it after the closing brace but before the semicolon.
- Do not place the structure tag name after the closing brace in the definition. That will be treated as a structure variable. The tag name must be placed before the opening brace but after the keyword **struct.**

- When we use **typedef** definition, the *type_name* comes after the closing brace but before the semicolon.
- We cannot declare a variable at the time of creating a **typedef** definition. We must use the *type name* to declare a variable in an independent statement.
- It is an error to use a structure variable as a member of its own **struct** type structure.
- Assigning a structure of one type to a structure of another type is an error.
- Declaring a variable using the tag name only (without the keyword struct) is an error.
- It is an error to compare two structure variables.
- It is illegal to refer to a structure member using only the member name.
- When structures are nested, a member must be qualified with all levels of structures nesting it.
- When accessing a member with a pointer and dot notation, parentheses are required around the pointer, like (*ptr).number.
- The selection operator (–>) is a single token. Any space between the symbols – and > is an error.
- When using **scanf** for reading values for members, we must use address operator & with non-string members.
- Forgetting to include the array subscript when referring to individual structures of an array of structures is an error.
- A union can store only one of its members at a time. We must exercise care in accessing the correct member. Accessing a wrong data is a logic error.
- It is an error to initialize a union with data that does not match the type of the first member.
- Always provide a structure tag name when creating a structure. It is convenient to use tag name to declare new structure variables later in the program.
- Use short and meaningful structure tag names.
- Avoid using same names for members of different structures (although it is not illegal).
- Passing structures to functions by pointers is more efficient than passing by value. (Passing by pointers are discussed in Chapter 11.)
- We cannot take the address of a bit field. Therefore, we cannot use **scanf** to read values in bit fields. We can neither use pointer to access the bit fields.
- Bit fields cannot be arrayed.

Review Questions

12.1 State whether the following statements are *true* or *false*.

(a) A **struct** type in C is a built-in data type.

(b) The tag name of a structure is optional.

(c) Structures may contain members of only one data type.

(d) A structure variable is used to declare a data type containing multiple fields.

(e) It is legal to copy a content of a structure variable to another structure variable of the same type.

(f) Structures are always passed to functions by printers.
(g) Pointers can be used to access the members of structure variables.
(h) We can perform mathematical operations on structure variables that contain only numeric type members.
(i) The keyword **typedef** is used to define a new data type.
(j) In accessing a member of a structure using a pointer p, the following two are equivalent: (*p).member_name and p –> member_name
(k) A union may be initialized in the same way a structure is initialized.
(l) A union can have another union as one of the members.
(m) A structure cannot have a union as one of its members.
(n) An array cannot be used as a member of a structure.
(o) A member in a structure can itself be a structure.

12.2 Fill in the blanks in the following statements:
(a) The _____ can be used to create a synonym for a previously defined data type.
(b) A _____ is a collection of data items under one name in which the items share the same storage.
(c) The name of a structure is referred to as _____.
(d) The selection operator –> requires the use of a _____ to access the members of a structure.
(e) The variables declared in a structure definition are called its _____.

12.3 A structure tag name **abc** is used to declare and initialize the structure variables of type **struct abc** in the following statements. Which of them are incorrect? Why? Assume that the structure **abc** has three members, **int, float** and **char** in that order.
(a) `struct a,b,c;`
(b) `struct abc a,b,c`
(c) `abc x,y,z;`
(d) `struct abc a[ ];`
(e) `struct abc a = { };`
(f) `struct abc = b, { 1+2, 3.0, "xyz"}`
(g) `struct abc c = {4,5,6};`
(h) `struct abc a = 4, 5.0, "xyz";`

12.4 Given the declaration

```
struct abc a,b,c;
```

which of the following statements are legal?
(a) `scanf ("%d, &a);`
(b) `printf ("%d", b);`
(c) `a = b;`
(d) `a = b + c;`
(e)
```
if (a>b)
    . . . . .
```

12.5 Given the declaration

```
struct item_bank
{
        int number;
        double cost;
};
```

which of the following are correct statements for declaring one dimensional array of structures of type **struct item_bank?**

(a) `int item_bank items[10];`
(b) `struct items[10] item_bank;`
(c) `struct item_bank items (10);`
(d) `struct item_bank items [10];`
(e) `struct items item_bank [10];`

12.6 Given the following declaration

```
typedef struct abc
{
        char x;
        int y;
        float z[10];
} ABC;
```

State which of the following declarations are invalid? Why?

(a) `struct abc v1;`
(b) `struct abc v2[10];`
(c) `struct ABC v3;`
(d) `ABC a,b,c;`
(e) `ABC a[10];`

12.7 How does a structure differ from an array?

12.8 Explain the meaning and purpose of the following:

(a) Template
(b) **struct** keyword
(c) **typedef** keyword
(d) **sizeof** operator
(e) Tag name

12.9 Explain what is wrong in the following structure declaration:

```
struct
{
        int number;
        float price;
}
main( )
{
```

```
        . . . . .
        . . . . .
}
```

12.10 When do we use the following?
- (a) Unions
- (b) Bit fields
- (c) The **sizeof** operator

12.11 What is meant by the following terms?
- (a) Nested structures
- (b) Array of structures

Give a typical example of use of each of them.

12.12 Given the structure definitions and declarations

```
struct abc
{
        int a;
        float b;
};
struct xyz
{
        int x;
        float y;
};
abc a1, a2;
xyz x1, x2;
```

find errors, if any, in the following statements:

(a) `a1 = x1;`

(b) `abc.a1 = 10.75;`

(c) `int m = a + x;`

(d) `int n = x1.x + 10;`

(e) `a1 = a2;`

(f) `if (a.a1 > x.x1) . . .`

(g) `if (a1.a < x1.x) . . .`

(h) `if (x1 != x2) . . .`

12.13 Describe with examples, the different ways of assigning values to structure members.

12.14 State the rules for initializing structures.

12.15 What is a 'slack byte'? How does it affect the implementation of structures?

12.16 Describe three different approaches that can be used to pass structures as function arguments.

12.17 What are the important points to be considered when implementing bit-fields in structures?

12.18 Define a structure called **complex** consisting of two floating-point numbers **x** and **y** and declare a variable **p** of type **complex**. Assign initial values 0.0 and 1.1 to the members.

12.19 What is the error in the following program?

```
typedef struct product
{
    char name [ 10 ];
    float price ;
} PRODUCT products [ 10 ];
```

12.20 What will be the output of the following program?

```
main ( )
{
  union x
  {
    int a;
    float b;
    double c ;
  };
  printf("%d\n", sizeof(x));
    a.x = 10;
  printf("%d%f%f\n", a.x, b.x, c.x);
    c.x = 1.23;
  printf("%d%f%f\n", a.x, b.x, c.x);
}
```

Programming Exercises

12.1 Define a structure data type called **time_struct** containing three members integer **hour,** integer **minute** and integer **second**. Develop a program that would assign values to the individual members and display the time in the following form:

16 : 40 : 51

12.2 Modify the above program such that a function is used to input values to the members and another function to display the time.

12.3 Design a function **update** that would accept the data structure designed in Exercise 12.1 and increments time by one second and returns the new time. (If the increment results in 60 seconds, then the second member is set to zero and the minute member is incremented by one. Then, if the result is 60 minutes, the minute member is set to zero and the hour member is incremented by one. Finally when the hour becomes 24, it is set to zero.)

12.4 Define a structure data type named **date** containing three integer members **day**, **month** and **year.** Develop an interactive modular program to perform the following tasks;

- To read data into structure members by a function
- To validate the date entered by another function
- To print the date in the format

  April 29, 2002

  by a third function.

The input data should be three integers like 29, 4, and 2002 corresponding to day, month and year. Examples of invalid data:

31, 4, 2002 – April has only 30 days

29, 2, 2002 – 2002 is not a leap year

12.5 Design a function **update** that accepts the **date** structure designed in Exercise 12.4 to increment the date by one day and return the new date. The following rules are applicable:

- If the date is the last day in a month, month should be incremented
- If it is the last day in December, the year should be incremented
- There are 29 days in February of a leap year

12.6 Modify the input function used in Exercise 12.4 such that it reads a value that represents the date in the form of a long integer, like 19450815 for the date 15-8-1945 (August 15, 1945) and assigns suitable values to the members **day, month** and **year.**

Use suitable algorithm to convert the long integer 19450815 into year, month and day.

12.7 Add a function called **nextdate** to the program designed in Exercise 12.4 to perform the following task;

- Accepts two arguments, one of the structure **data** containing the present date and the second an integer that represents the number of days to be added to the present date.
- Adds the days to the present date and returns the structure containing the next date correctly.

Note that the next date may be in the next month or even the next year.

12.8 Use the **date** structure defined in Exercise 12.4 to store two dates. Develop a function that will take these two dates as input and compares them.

- It returns 1, if the **date1** is earlier than **date2**
- It returns 0, if **date1** is later date

12.9 Define a structure to represent a vector (a series of integer values) and write a modular program to perform the following tasks:

- To create a vector
- To modify the value of a given element
- To multiply by a scalar value
- To display the vector in the form
  (10, 20, 30, ...)

12.10 Add a function to the program of Exercise 12.9 that accepts two vectors as input parameters and return the addition of two vectors.

12.11 Create two structures named **metric** and **British** which store the values of distances. The **metric** structure stores the values in meters and centimeters and the British structure stores the values in feet and inches. Write a program that reads values for the structure variables and adds values contained in one variable of **metric** to the contents of another variable of **British.** The program should display the result in the format of feet and inches or metres and centimetres as required.

12.12 Define a structure named **census** with the following three members:

- A character array city [ ] to store names
- A long integer to store population of the city
- A float member to store the literacy level

Write a program to do the following:

- To read details for 5 cities randomly using an array variable
- To sort the list alphabetically
- To sort the list based on literacy level
- To sort the list based on population
- To display sorted lists

12.13 Define a structure that can describe a hotel. It should have members that include the name, address, grade, average room charge, and number of rooms.

Write functions to perform the following operations:

- To print out hotels of a given grade in order of charges
- To print out hotels with room charges less than a given value

12.14 Define a structure called **cricket** that will describe the following information:

player name
team name
batting average

Using **cricket,** declare an array **player** with 50 elements and write a program to read the information about all the 50 players and print a team-wise list containing names of players with their batting average.

12.15 Design a structure **student_record** to contain name, date of birth and total marks obtained. Use the **date** structure designed in Exercise 12.4 to represent the date of birth.

Develop a program to read data for 10 students in a class and list them rank-wise.

Debugging Exercises

12.1 Identify the error in the following code, if any:

```
#include<stdio.h>

void main()
{
 int a;
 int *ptr;
 a=50;
 ptr=&a;

 printf("address of a = %u\tvalue of a = %d\n",ptr,**&ptr);
 printf("address of ptr = %u\tvalue of ptr = %u",&ptr,ptr);

}
```

12.2 Identify the error in the following code, if any:

```
#include<stdio.h>

void main()
```

```
{
 int i;
 char *name[3] =
      {
         "New Zealand",
         "Australia",
         "India"
      };
 for(i=0;i<3;i++)
   printf("Address of name [%d] = %u, which points to the string
\"%s\"\n",i,name[i],*name[i]);

}
```

12.3 Identify the error in the following code, if any:

```
#include<stdio.h>

void main()
{
 int i;
 int num[3] = {22,34,88};
 void print(int *num);
 print(num);
}
void print(int *num1)
{
 int i;
 printf("The values stored in the array are: ");
 for(i=0;i<3;i++)
 printf("%d\t",*num1[i]);
}
```

13

Principles of Object-Oriented Programming

13.1 Software Crisis

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession.This has forced the software engineers and industry to continuously look for new approaches to software design and development, and they are becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. These rapid advances appear to have created a situation of crisis within the industry. The following issues need to be addressed to face this crisis:

- How to represent real-life entities of problems in system design?
- How to design systems with open interfaces?
- How to ensure reusability and extensibility of modules?
- How to develop modules that are tolerant to any changes in future?
- How to improve software productivity and decrease software cost?
- How to improve the quality of software?
- How to manage time schedules?
- How to industrialize the software development process?

Many software products are either not finished, or not used, or else are delivered with major errors. Figure 13.1 shows the fate of the US defence software projects undertaken in the 1970s. Around 50% of the software products were never delivered, and one-third of those which were delivered were never used. It is interesting to note that only 2% were used as delivered, without being subjected to any changes. This illustrates that the software industry has a remarkably bad record in delivering products.

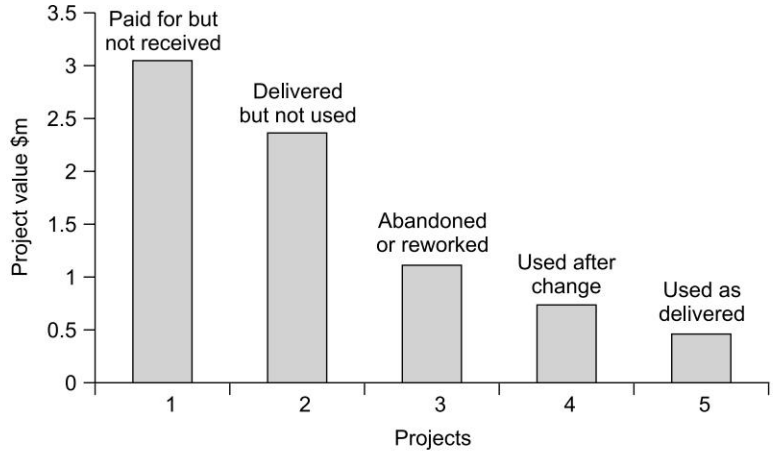


Fig. 13.1 *The state of US defence projects (according to the US government)*

Changes in user requirements have always been a major problem. Another study (Fig. 13.2) shows that more than 50% of the systems required modifications due to changes in user requirements and data formats. It only illustrates that, in a changing world with a dynamic business environment, requests for change are unavoidable and therefore systems must be adaptable and tolerant to changes.

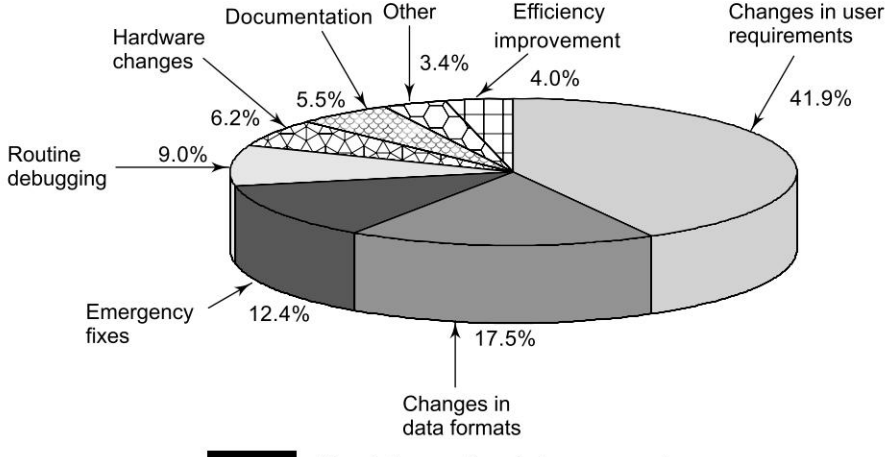


Fig. 13.2 *Breakdown of maintenance costs*

These studies and other reports on software implementation suggest that software products should be evaluated carefully for their quality before they are delivered and implemented. Some of the quality issues that must be considered for critical evaluation are:

1. Correctness
2. Maintainability
3. Reusability
4. Openness and interoperability
5. Portability

6. Security
7. Integrity
8. User friendliness

Selection and use of proper software tools would help resolving some of these issues.

13.2 Software Evolution

Ernest Tello, a well-known writer in the field of artificial intelligence, compared the evolution of software technology to the growth of a tree. Like a tree, the software evolution has had distinct phases or "layers" of growth. These layers were built up one by one over the last five decades as shown in Fig. 13.3, with each layer representing an improvement over the previous one. However, the analogy fails if we consider the life of these layers. In software systems, each of the layers continues to be functional, whereas in the case of trees, only the uppermost layer is functional.

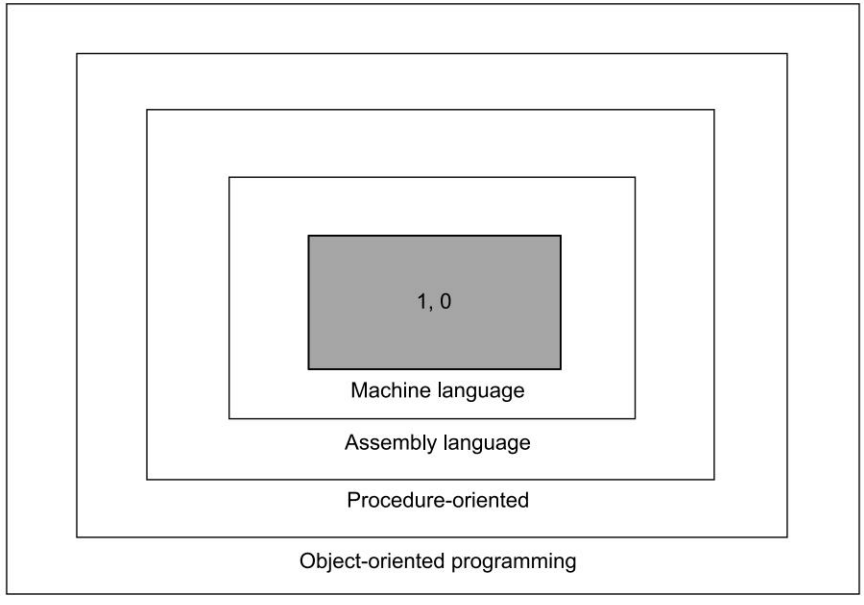


Fig. 13.3 *Layers of computer software*

Alan Kay, one of the promoters of the object-oriented paradigm and the principal designer of Smalltalk, has said: "*As complexity increases, architecture dominates the basic material*". To build today's complex software it is just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate sound construction techniques and program structures that are easy to comprehend, implement and modify.

Since the invention of the computer, many programming approaches have been tried. These include techniques such as *modular programming, top-down programming, bottom-up programming and structured programming*. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable. These techniques have become popular among programmers over the last two decades.

With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980s. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object-Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

13.3 A Look at Procedure-Oriented Programming

Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as *procedure-oriented programming (POP)*. In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical program structure for procedural programming is shown in Fig. 13.4. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

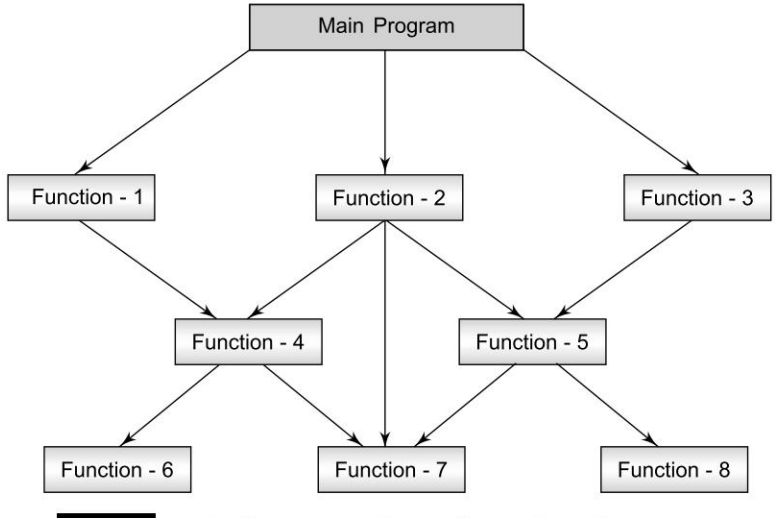


Fig. 13.4 *Typical structure of procedure-oriented programs*

Procedure-oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as *functions*. We normally use a *flowchart* to organize these actions and represent the flow of control from one action to another. While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

In a multi-function program, many important data items are placed as *global* so that they may be accessed by all the functions. Each function may have its own *local data*. Figure 13.5 shows the relationship of data and functions in a procedure-oriented program.

Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

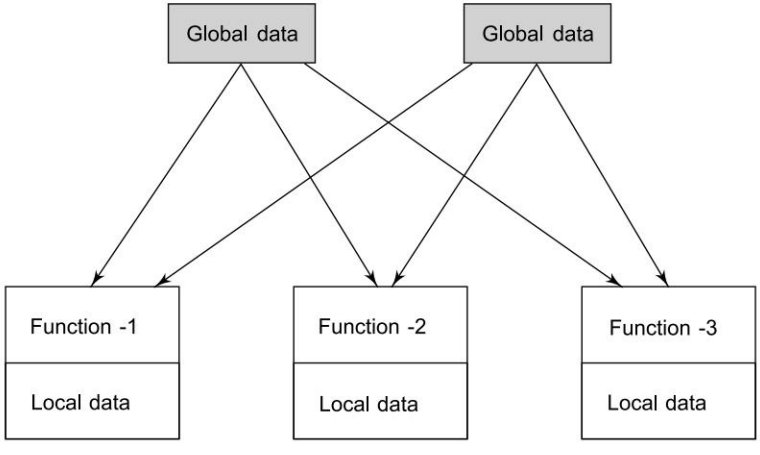


Fig. 13.5 *Relationship of data and functions in procedural programming*

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

13.4 Object-Oriented Programming Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called *objects* and then builds data and functions around these objects. The organization of data and functions in object-oriented programs is shown in Fig. 13.6. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

Some of the striking features of object-oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

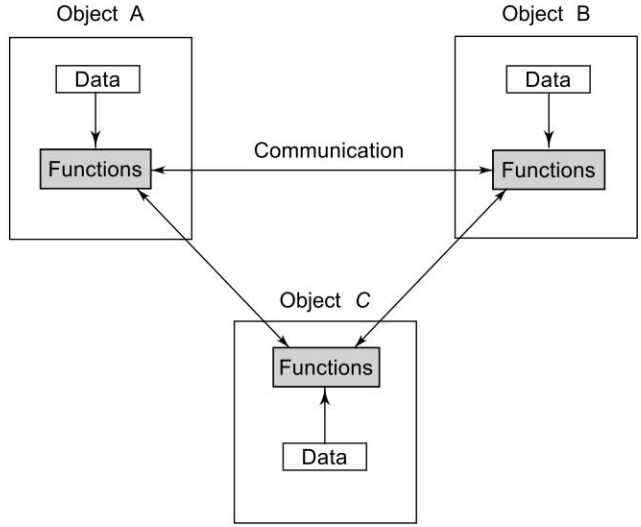


Fig. 13.6 *Organization of data and functions in OOP*

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people. It is therefore important to have a working definition of object-oriented programming before we proceed further. We define "object-oriented programming as *an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.*" Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

13.5 Basic Concepts of Object-Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance

- Polymorphism
- Dynamic binding
- Message passing

We shall discuss these concepts in some detail in this section.

13.5.1 Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer"and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. Although different authors represent them differently, Fig. 13.7 shows two notations that are popularly used in object-oriented analysis and design.

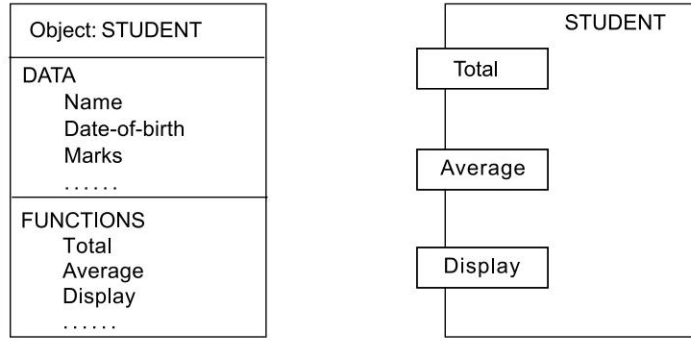


Fig. 13.7 *Two ways of representing an object*

13.5.2 Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a *class*. In fact, objects are variables of the type *class*. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object **mango** belonging to the class **fruit**.

13.5.3 Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as *encapsulation*. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost, and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member functions*.

Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

13.5.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of *hierarchical classification*. For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in Fig. 13.8.

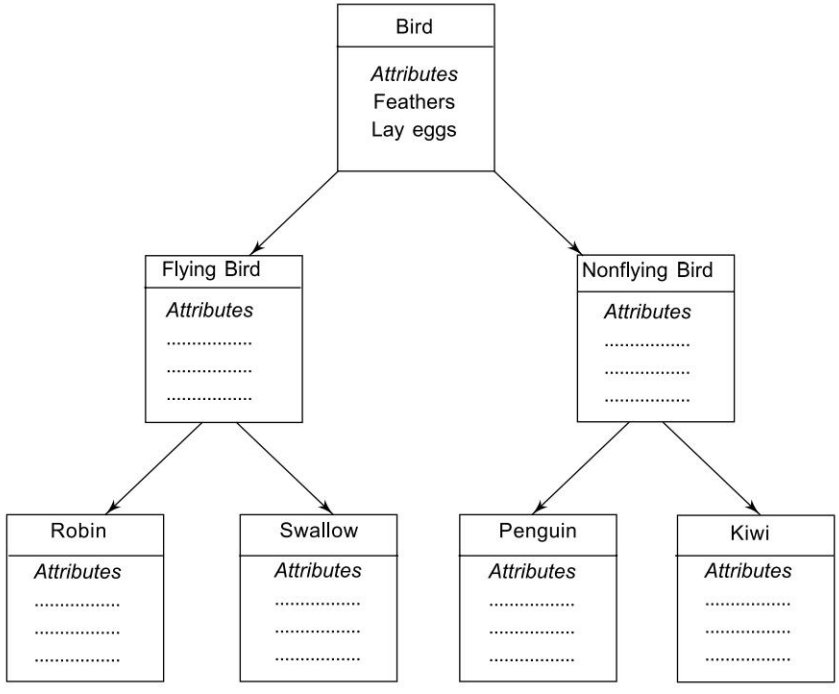


Fig. 13.8 *Property inheritance*

In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

13.5.5 Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading*.

Figure 13.9 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

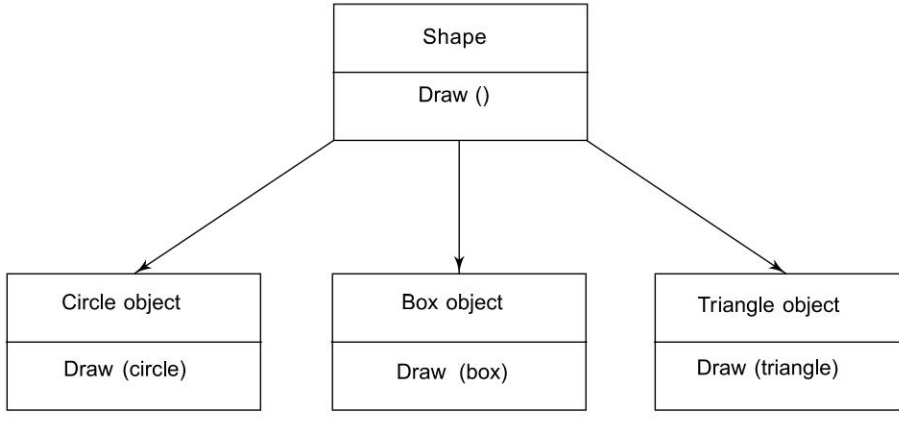


Fig. 13.9 *Polymorphism*

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

13.5.6 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call

is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in Fig. 13.9. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

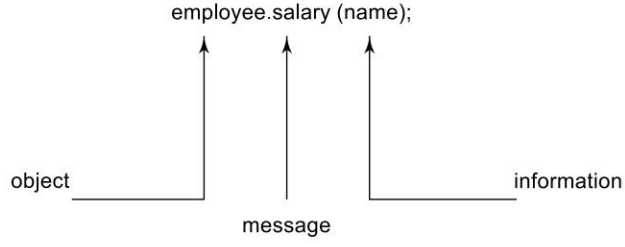
13.5.7 Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent. Example:



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

13.6 Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descri-ptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their impor-tance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

Developing a software that is easy to use makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

13.7 Object-Oriented Languages

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages, and
2. Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statement:

```
Object-based features + inheritance + dynamic binding
```

Languages that support these features include C++, Smalltalk, Object Pascal and Java There are a large number of object-based and object-oriented programming languages. Table 13.1 lists some popular general purpose OOP languages and their characteristics.

Table 13.1 ***Characteristics of some OOP languages***

Characteristics	*Simula* *	*Smalltalk* *	*Objective C*	*C++*	*Ada* **	*Object Pascal*	*Turbo Pascal*	*Eiffel* *	*Java* *
Binding (early or late)	Both √	Late √	Both √	Both √	Early √	Late √	Early √	Early √	Both √
Polymorphism	√	√	√	√	√	√	√	√	√
Data hiding	√	√	√	√	√	√	√	√	√
Concurrencry	√	Poor	Poor	Poor	Difficult	No	No	Promised	√
Inheritance	√	√	√	√	No	√	√	√	√
Multiple Inheritance	No	√	√	√	No	---	---	√	No
Garbage Collection	√	√	√	√	No	√	√	√	√
Persistence	No	Promised	No	No	like 3GL	No	No	Some Support	√
Genericity	No	No	No	√	√	No	No	√	No
Object Libraries	√	√	√	√	Not much	√	√	√	√

* *Pure object-oriented languages*

** *Object-based languages*

Others are extended conventional languages

As seen from Table 13.1, all languages provide for polymorphism and data hiding. However, many of them do not provide facilities for concurrency, persistence and genericity. Eiffel, Ada and C++ provide generic facility which is an important construct for supporting reuse. However, persistence (a process of storing objects) is not fully supported by any of them. In Smalltalk, though the entire current execution state can be saved to disk, yet the individual objects cannot be saved to an external file.

Commercially, C++ is only 10 years old, Smalltalk and Objective C 13 years old, and Java only 5 years old. Although Simula has existed for more than two decades, it has spent most of its life in a research environment. The field is so new, however, that it should not be judged too harshly.

Use of a particular language depends on characteristics and requirements of an application, organizational impact of the choice, and reuse of the existing programs. C++ has now become the most successful, practical, general purpose OOP language, and is widely used in industry today.

13.8 Applications of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The richness of OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

Just Remember

- Software technology has evolved through a series of phases during the last five decades.
- The most popular phase till recently was procedure-oriented programming (POP).
- POP employs *top-down* programming approach where a problem is viewed as a sequence of tasks to be performed. A number of functions are written to implement these tasks.
- POP has two major drawbacks, viz. (1) data move freely around the program and are therefore vulnerable to changes caused by any function in the program, and (2) it does not model very well the real-world problems.
- Object-oriented programming (OOP) was invented to overcome the drawbacks of the POP. It employs the *bottom-up* programming approach. It treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it in a data structure called **class**. This feature is called **data** *encapsulation*.
- In OOP, a problem is considered as a collection of a number of entities called **objects**. Objects are instances of classes.
- Insulation of data from direct access by the program is called *data hiding*.
- *Data abstraction* refers to putting together essential features without including background details.

- *Inheritance* is the process by which objects of one class acquire properties of objects of another class.
- *Polymorphism* means one name, multiple forms. It allows us to have more than one function with the same name in a program. It also allows overloading of operators so that an operation can exhibit different behaviours in different instances.
- *Dynamic binding* means that the code associated with a given procedure is not known until the time of the call at run-time.
- *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Object-oriented technology offers several benefits over the conventional programming methods---the most important one being the reusability.
- Applications of OOP technology has gained importance in almost all areas of computing including real-time business systems.
- There are a number of languages that support object-oriented programming paradigm. Popular among them are C++, Smalltalk and Java. C++ has become an industry standard language today.

Review Questions

13.1 What do you think are the major issues facing the software industry today?
13.2 Briefly discuss the software evolution during the period 1950 – 1990.
13.3 What is procedure-oriented programming? What are its main characteristics?
13.4 Discuss an approach to the development of procedure-oriented programs.
13.5 Describe how data are shared by functions in a procedure-oriented program.
13.6 What is object-oriented programming? How is it different from the procedure-oriented programming?
13.7 How are data and functions organized in an object-oriented program?
13.8 What are the unique advantages of an object-oriented programming paradigm?
13.9 Distinguish between the following terms:
  (a) Objects and classes
  (b) Data abstraction and data encapsulation
  (c) Inheritance and polymorphism
  (d) Dynamic binding and message passing
13.10 What kinds of things can become objects in OOP?
13.11 Describe inheritance as applied to OOP.
13.12 What do you mean by dynamic binding? How is it useful in OOP?
13.13 How does object-oriented approach differ from object-based approach?
13.14 List a few areas of application of OOP technology.
13.15 State whether the following statements are TRUE or FALSE.
  (a) In procedure-oriented programming, all data are shared by all functions.
  (b) The main emphasis of procedure-oriented programming is on algorithms rather than on data.

(c) One of the striking features of object-oriented programming is the division of programs into objects that represent real-world entities.
(d) Wrapping up of data of different types into a single unit is known as encapsulation.
(e) One problem with OOP is that once a class is created it can never be changed.
(f) Inheritance means the ability to reuse the data values of one object by
(g) Polymorphism is extensively used in implementing inheritance.
(h) Object-oriented programs are executed much faster than conventional programs.
(i) Object-oriented systems can scale up better from small to large.
(j) Object-oriented approach cannot be used to create databases.



14 Beginning with C++

14.1 What is C++?

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

During the early 1990's the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardised these changes and added several new features to the language specifications.

C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore, almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler. We shall see these differences later as and when they are encountered.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

The object-oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down, structured design, to one that provides bottom-up, object-oriented design.

14.2 Applications of C++

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

- Since C++ allows us to create hierarchy-related objects, we can buildspecial object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general-purpose language in the near future.

14.3 A Simple C++ Program

Let us begin with a simple example of a C++ program in Fig.14.1 that prints a string on the screen.

```
Program
     PRINTING A STRING
     #include <iostream> // include header file

     using namespace std;

     int main()
     {
       cout << "C++ is better than C.\n"; // C++ statement

         return 0;
     }         // End of example
```

Fig. 14.1 *Simple program demonstrating several C++ features.*

14.3.1 Program Features

Like C, the C++ program is a collection of functions. The above example contains only one function, **main()**. As usual, execution begins at main(). Every C++ program must have a **main()**. C++ is a free-form language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

14.3.2 Comments

C++ introduces a new comment symbol // (double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
// This is an example of
// C++ program to illustrate
// Some of its features
```

The C comment symbols /*, */ are still valid and are more suitable for multiline comments. The following comment is allowed:

```
/*  This is an example of
   C++ program to illustrate
   some of its features
*/
```

We can use either or both styles in our programs. Since this is a book on C++, we will use only the C++ style. However, remember that we can not insert a // style comment within the text of a program line. For example, the double slash comment cannot be used in the manner as shown below:

```
for(j=0; j<n; /* loops n times */ j++)
```

14.3.3 Output Operator

The only statement in the program in Fig. 14.2 is an output statement. The statement

```
cout << "C++ is better than C.";
```

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout (pronounced as 'C out') is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. We shall later discuss streams in detail.

The operator << is called the *insertion or put to* operator. It inserts (or sends) the contents of the variable on its right to the object on its left (Fig. 14.2).

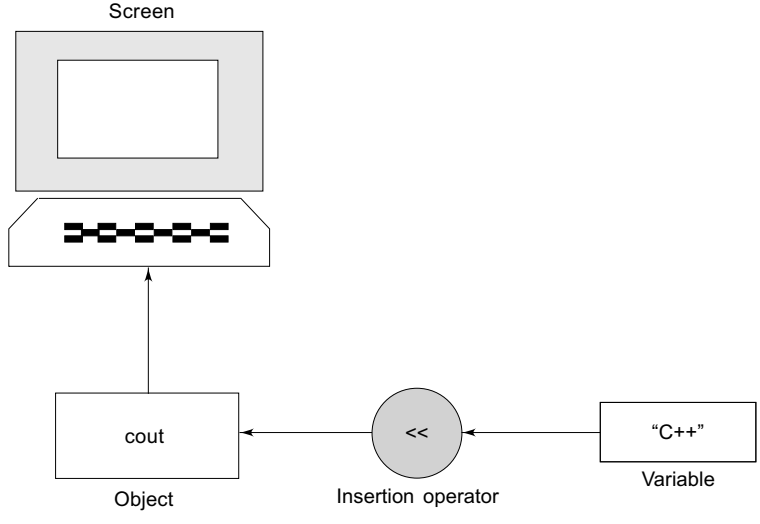


Fig. 14.2 *Output using insertion operator*

The object **cout** has a simple interface. If string represents a string variable, then the following statement will display its contents:

```
cout << string;
```

You may recall that the operator << is the bit-wise left-shift operator and it can still be used for this purpose. This is an example of how one operator can be used for different purposes, depending on the context. This concept is known as *operator overloading,* an important aspect of polymorphism.

It is important to note that we can still use printf() for displaying an output. C++ accepts this notation. However, we will use cout << to maintain the spirit of C++.

14.3.4 The iostream File

We have used the following #include directive in the program:

```
#include <iostream>
```

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier **cout** and the operator <<. Some old versions of C++ use a header file called iostream.h. This is one of the changes introduced by ANSI C++. (We should use iostream.h if the compiler does not support ANSI C++ features.)

The header file **iostream** should be included at the beginning of all programs that use input/output statements. Note that the naming conventions for header files may vary. Some implementations use **iostream.hpp**; yet others **iostream.hxx**. We must include appropriate header files depending on the contents of the program and implementation.

Tables 14.1 and 14.2 provide lists of C++ standard library header files that may be needed in C++ programs. The header files with .h extension are "old style" files which should be used with old compilers. Table 14.1 also gives the version of these files that should be used with the ANSI standard compilers.

Table 14.1 ***Commonly used old-style header files***

Header file	Contents and purpose	New version
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging	<cassert>
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.	<cctype>
<float.h>	Contains the floating-point size limits of the system.	<cfloat>
<limits.h>	Contains the integral size limits of the system.	<climits>
<math.h>	Contains function prototypes for math library functions.	<cmath>
<stdio.h>	Contains function prototypes for the standard input/output library functions and information used by them.	<cstdio>
<stdlib.h>	Contains function prototypes for conversion of numbers to text, text to numbers, memory allocation, random numbers, and various other utility functions.	<cstdlib>

Contd.

Table 14.1 *(Contd)*

Header file	Contents and purpose	New version
<string.h>	Contains function prototypes for C-style string processing functions.	<cstring>
<time.h>	Contains function prototypes and types for manipulating the time and date.	
<iostream.h>	Contains function prototypes for the standard input and standard output functions.	<iostream>
<iomanip.h>	Contains function prototypes for the stream manipulators that enable formatting of streams of data.	<iomanip>
<fstream.h>	Contains function prototypes for functions that perform input from files on disk and output to files on disk.	<fstream>

Table 14.2 *New header files included in ANSI C++*

Header file	Contents and purpose
<utility>	Contains classes and functions that are used by many standard library header files.
<vector>, <list>, <deque> <queue>, <set>, <map>, <stack>, <bitset>	The header files contain classes that implement the standard library containers. Containers store data during a program's execution.
<functional>	Contains classes and functions used by algorithms of the standard library.
<memory>	Contains classes and functions used by the standard library to allocate memory to the standard library containers.
<iterator>	Contains classes for manipulating data in the standard library containers.
<algorithm>	Contains functions for manipulating data in the standard library containers.
<exception>, <stdexcept>	These header files contain classes that are used for exception handling.
<string>	Contains the definition of class string from the standard library.
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.)
<limits>	Contains a class for defining the numerical data type limits on each computer platform.
<typeinfo>	Contains classes for run-time type identification (determining data types at execution time).

14.3.5 Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the **namespace** scope we must include the using directive, like

```
using namespace std;
```

Here, **std** is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in **std** to the current

14.3.6 Return Type of main( )

In C++, main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a return(0) statement; otherwise a warning or an error might occur. Since **main()** returns an integer type value, return type for main() is explicitly specified as **int**. Note that the default return type for all functions in C++ is **int**. The following **main** without type and return will run with a warning:

```
main()
{
      ......
      ......
}
```

14.4 More C++ Statements

Let us consider a slightly more complex C++ program. Assume that we would like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in the program in Fig. 14.3.

Program

```
          AVERAGE OF TWO NUMBERS
          AVERA#include <io=\stream>

          using namespace std;
          int main()
          {
          float number1, number2,
             sum, average;
          cout << "Enter two numbers: ";  // prompt
          cin >> number1;     // Reads numbers
          cin >> number2;     // from keyboard
          sum = number1 + number2;
          average = sum/2;
          cout << "Sum = " << sum << "\n";
          cout << "Average = " << average << "\n";

          return 0;
      }
```

Output

```
          Enter two numbers: 6.5 7.5
          Sum = 14
          Average = 7
```

Fig. 14.3 *Program displaying average of two numbers*

14.4.1 Variables

The program uses four variables number1, number2, sum, and average. They are declared as type float by the statement.

```
float number1, number2, sum, average;
```

All variables must be declared before they are used in the program.

14.4.2 Input Operator

The statement

```
cin >> number1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier **cin** (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as *extraction or get from* operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right (Fig. 14.4). This corresponds to the familiar scanf() operation. Like << , the operator >> can also be overloaded.

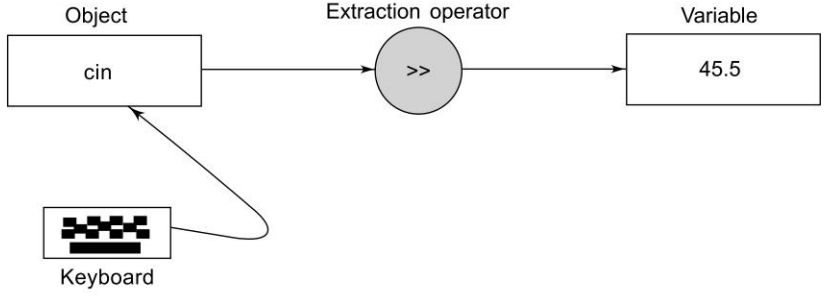


Fig. 14.4 *Input using extraction operator*

14.4.3 Cascading of I/O Operators

We have used the *insertion operator* << repeatedly in the last two statements for printing results.

The statement

```
cout << "Sum = " << sum << "\n";
```

first sends the string "Sum =" to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called *cascading*. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
cout << "Sum = " << sum << "\n"
     << "Average = " << average << "\n";
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be:

```
cout << "Sum = " << sum << ","
     << "Average = " << average << "\n";
```

The output will be:

```
Sum = 14, Average = 7
```

We can also cascade input operator >> as shown below:

```
cin >> number1 >> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to number1 and 20 to number2.

14.5 An Example with Class

One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

The program in Fig. 14.5 shows the use of class in a C++ program.

```
Program
      USE OF CLASS
      #include <iostream>
      using namespace std;
      class person
      {
         char name[30];
         int age;

         public:
           void getdata(void);
           void display(void);
      };
      void person :: getdata(void)
      {
         cout << "Enter name: ";
         cin >> name;
         cout << "Enter age: ";
         cin >> age;
      }
      void person :: display(void)
      {
           cout << "\nName: " << name;
           cout << "\nAge: " << age;
      }

      int main()
      {
           person p;
```

Cont'd

```
            p.getdata();
            p.display();

            return 0;
        }
```

Output

```
Enter Name: Ravinder
Enter Age:  30

Name: Ravinder
Age:  30
```

Fig. 14.5 *Use of class in a C++ program*

Note **cin** *can read only one word and therefore we cannot use names with blank spaces.*

The program defines **person** as a new data of type class. The class person includes two basic data type items and two functions to operate on that data. These functions are called **member functions**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as *objects*. Here, p is an object of type **person**. Class objects are used to invoke the functions defined in that class. More about classes and objects is discussed in Chapter 17.

14.6 Structure of C++ Program

As it can be seen a typical C++ program would contain four sections as shown in Fig. 14.6. These sections may be placed in separate code files and then compiled independently or jointly.

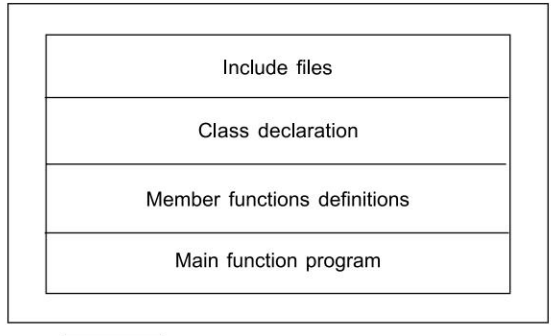


Fig. 14.6 *Structure of a C++ program*

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third file which "includes" the previous two files as well as any other files required.

This approach is based on the concept of client-server model as shown in Fig. 14.7. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

14.7 Creating the Source File

Like C programs, C++ programs can be created using any text editor. For example, on the UNIX, we can use vi or *ed* text editor for creating and editing the source code. On the DOS system, we can use *edlin* or any other editor available or a word processor system under non-document mode.

Some systems such as Turbo C++ provide an integrated environment for developing and editing programs. Appropriate manuals should be consulted for complete details.

The file name should have a proper file extension to indicate that it is a C++ program file. C++ implementations use extensions such as .c, .C, .cc, .cpp and .cxx. Turbo C++ and Borland C++ use .c for C programs and .cpp (C plus plus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extensions to be used.

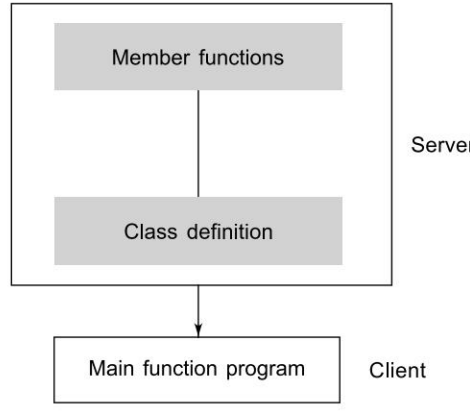


Fig. 14.7 *The client-server model*

14.8 Compiling and Linking

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

14.8.1 Unix AT&T C++

The process of implementation of a C++ program under UNIX is similar to that of a C program.We should use the "CC" (uppercase) command to compile the program. Remember, we use lowercase "cc" for compiling C programs. The command

```
CC example.C
```

at the UNIX prompt would compile the C++ program source code contained in the file **example.C**. The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a.out**.

A program spread over multiple files can be compiled as follows:

```
CC file1.C file2.o
```

The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file. This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

14.8.2 Turbo C++ and Borland C++

Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar which includes options such as File, Edit, Compile and Run.

We can create and save the source files under the **File option**, and edit them under the **Edit option**. We can then compile the program under the **Compile option** and execute it under the **Run option**. The **Run option** can be used without compiling the source code. In this case, the **RUN** command causes the system to compile, link and run the program in one step.

14.8.3 Visual C++

It is a Microsoft application development system for C++ that runs under Windows. Visual C++ is a visual programming environment in which basic program components can be selected through menu choices, buttons, icons, and other predetermined methods.

Just Remember

- C++ is a superset of C language.
- C++ adds a number of object-oriented features such as objects, inheritance, function overloading and operator overloading to C. These features enable building of programs with clarity, extensibility and ease of maintenance.
- C++ can be used to build a variety of systems such as editors, compilers, databases, communication systems, and many more complex real-life application systems.
- C++ supports interactive input and output features and introduces a new comment symbol // that can be used for single line comments. It also supports C-style comments.
- Like C programs, execution of all C++ programs begins at **main( )** function and ends at **return( )** statement. The header file **iostream** should be included at the beginning of all programs that use input/output operations.
- All ANSI C++ programs must include **using namespace std** directive.
- A typical C++ program would contain four basic sections, namely, include files section, class declaration section, member function section and main function section.
- Like C programs, C++ programs can be created using any text editor.
- Most compiler systems provide an integrated environment for developing and executing programs. Popular systems are UNIX AT&T C++, Turbo C++ and Microsoft Visual C++.

Review Questions

14.1 State whether the following statements are TRUE or FALSE.

(a) Since C is a subset of C++, all C programs will run under C++ compilers.
(b) In C++, a function contained within a class is called a member function.
(c) Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.

(d) In C++, it is very easy to add new features to the existing structure of an object.
(e) The concept of using one operator for different purposes is known as oerator overloading.
(f) The output function printf() cannot be used in C++ programs.

14.2 Why do we need the preprocessor directive #include <iostream> ?
14.3 How does a main() function in C++ differ from main() in C?
14.4 What do you think is the main advantage of the comment // in C++ as compared to the old C type comment?
14.5 Describe the major parts of a C++ program.

Programming Exercises

14.1 Write a program to display the following output using a single cout statement.

Maths = 90
Physics = 77
Chemistry = 69

14.2 Write a program to read two numbers from the keyboard and display the larger value on the screen.
14.3 Write a program to input an integer value from keyboard and display on screen "WELL DONE" that many times.
14.4 Write a program to read the values of a, b and c and display the value of x, where

$$x = a / b - c$$

Test your program for the following values:

(a) a = 250, b = 85, c = 25
(b) a = 300, b = 70, c = 70

14.5 Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.
14.6 Redo Exercise 14.5 using a class called temp and member functions.

Debugging Exercises

14.1 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
      int i = 0;
      i = i + 1;
      cout << i << " ";
      /*comment\*//i = i + 1;
      cout << i;
}
```

14.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
      short i=2500, j=3000;
      cout >> "i + j = " >> -(i+j);
}
```

14.3 What will happen when you run the following program?

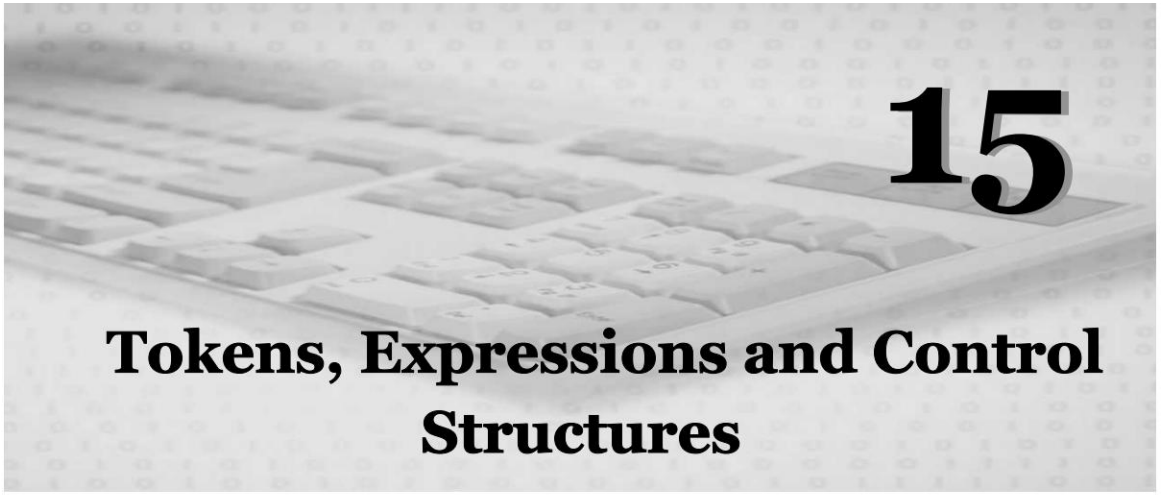
```
#include <iostream.h>
void main()
{
      int i=10, j=5;
      int modResult=0;
      int divResult=0;

      modResult = i%j;
      cout << modResult << " ";

      divResult = i/modResult;
      cout << divResult;
}
```

14.4 Find errors, if any, in the following C++ statements.

(a) cout << "x=" x;
(b) m = 5; // n = 10; // s = m + n;
(c) cin >>x; >>y;
(d) cout << \n "Name:" << name;
(e) cout <<"Enter value:"; cin >> x;
(f) /*Addition*/ z = x + y;



15 Tokens, Expressions and Control Structures

15.1 Introduction

As mentioned earlier, C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions. In this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.

15.2 Tokens

As we know, the smallest individual units in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

15.3 Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Table 15.1 gives the complete set of C++ keywords. Many of them are common to both C and C++. The ANSI C keywords are shown in boldface. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. ANSI C++ standards committee has added some more keywords to make the language more versatile. These are shown separately. Meaning and purpose of all C++ keywords are given in Appendix D.

Table 15.1 ***C++ keywords***

asm	**double**	new	**switch**
auto	**else**	operator	template
break	**enum**	private	this
case	**extern**	protected	throw
catch	**float**	public	try
char	**for**	**register**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Note *The ANSI C keywords are shown in bold face.*

15.4 Identifiers and Constants

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable which is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.

Constants refer to fixed values that do not change during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Examples:

```
123       // decimal integer
12.34     // floating point integer
037       // octal integer
0X2       // hexadecimal integer
"C++"     // string constant
'A'       // character constant
L'ab'     // wide-character constant
```

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

C++ also recognizes all the backslash character constants available in C.

Note *C++ supports two types of string representation — the C-style character string and the string class type introduced with Standard C++. Although the use of the string class type is recommended, it is advisable to understand and use C-style strings in some situations.*

15.5 Basic Data Types

Data types in C++ can be classified under various categories as shown in Fig. 15.1.

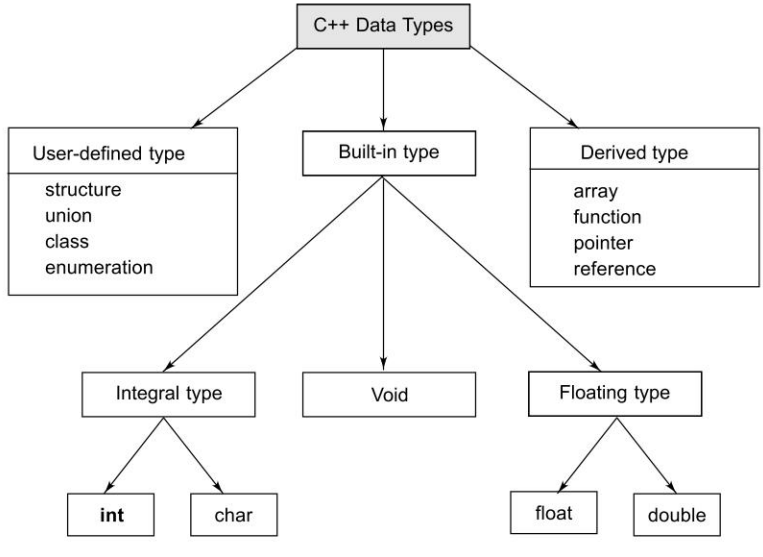


Fig. 15.1 *Hierarchy of C++ data type*

Both C and C++ compilers support all the built-in (also known as *basic* or *fundamental*) data types. With the exception of **void**, the basic data types may have several *modifiers* preceding them to serve the needs of various situations. The modifiers **signed, unsigned, long**, and **short** may be applied to character and integer basic data types. However, the modifier **long** may also be applied to **double**. Data type representation is machine specific in C++. Table 15.2 lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

Table 15.2 ***Size and range of C++ basic data types***

Type	Bytes	Range
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	– 128 to 127
int	2	– 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	– 31768 to 32767
short int	2	– 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
signed long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

ANSI C++ committee has added two more data types, **bool** and **wchar_t**.

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void funct1(void);
```

Another interesting use of **void** is in the declaration of generic pointers. Example:

```
void *gp;    // gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

```
int *ip;     // int pointer
gp = ip;     // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to a **void** value.

Assigning any pointer type to a **void** pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a **void** pointer to a non-**void** pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
void *ptr1;
char *ptr2;
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *)ptr1;
```

15.6 User-Defined Data Types

15.6.1 Structures and Classes

We have used user-defined data types such as **struct** and **union** in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming. More about these data types is discussed later in Chapter 17.

15.6.2 Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position{off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names **shape, colour**, and **position** become new type names. By using these tag names, we can declare new variables. Examples:

```
shape ellipse;        // ellipse is of type shape
colour background; // background is of type colour
```

ANSI C defines the types of **enums** to be **ints**. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an **int** value to be automatically converted to an **enum** value. Examples:

```
colour background = blue;       // allowed
colour background = 7;          // Error in C++
colour background = (colour) 7; // OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c = red;   // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example,

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous **enums** (i.e., **enums** without tag names). Example:

```
enum{off, on};
```

Here, **off** is 0 and **on** is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement as shown in Fig 15.2.

```
enum shape
{
   circle,
   rectangle,
   triangle
};

int main()
{
   cout << "Enter shape code:";
   int code;
   cin >> code;
   while(code >= circle && code <= triangle)
   {
      switch(code)
      {
         case circle:
         ......
         ......
         break;
         case rectangle:
         ......
         ......
         break;
         case triangle:
         ......
         ......
         break;
      }
      cout << "Enter shape code:";
      cin >> code;
   }
```

Contd

```
        cout << "BYE \n";

        return 0;
    }
```

Fig. 15.2 *Defining symbolic constants for a* ***switch*** *statement by enumeration*

ANSI C permits an **enum** to be defined within a structure or a class, but the **enum** is globally visible. In C++, an **enum** defined within a class (or structure) is local to that class (or structure) only.

15.7 Derived Data Types

15.7.1 Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
char string[3] = "xyz";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string[4] = "xyz";  // O.K. for C++
```

15.7.2 Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ program more reliable and readable. All the features of C++ functions are discussed in Chapter 16.

15.7.3 Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip;     // int pointer
ip = &x;     // address of x assigned to ip
*ip = 10;    // 10 assigned to x through indirection
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char * const ptr1 = "GOOD"; // constant pointer
```

We cannot modify the address that **ptr1** is initialized to.

```
int const * ptr2 = &m;    // pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

```
const char * const cp = "xyz";
```

This statement declares cp as a constant pointer to the string which has been declared a constant. In

Note *Pointers are extensively used in C++ for memory management and achieving polymorphism.*

15.8 Symbolic Constants

There are two ways of creating symbolic constants in C++:

- Using the qualifier **const**, and
- Defining a set of integer constants using **enum** keyword.

In both C and C++, any value declared as **const** cannot be modified by the program in any way. However, there are some differences in implementation. In C++, we can use **const** in a constant expression, such as

```
const int size = 10;
char name[size];
```

This would be illegal in C. **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

As with **long** and **short**, if we use the **const** modifier alone, it defaults to **int**. For example,

```
const size = 10;
```

means

```
const int size = 10;
```

The *named constants* are just like variables except that their values cannot be changed.

C++ requires a **const** to be initialized. ANSI C does not require an initializer; if none is given, it initializes the **const** to 0.

The scoping of **const** values differs. A **const** in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, **const** values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as **static**. To give a **const** value an external linkage so that it can be referenced from another file, we must explicitly define it as an **extern** in C++. Example:

```
extern const total = 100;
```

Another method of naming integer constants is by enumeration as under;

```
enum {X,Y,Z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const X = 0;
const Y = 1;
const Z = 2;
```

We can also assign values to X, Y, and Z explicitly. Example:

```
enum{X=100, Y=50, Z=200};
```

Such values can be any integer values. Enumerated data type has been discussed in detail in Section 15.6.2

15.9 Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines **int, short int**, and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char, char**, and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way **char** constants are stored. In C, they are stored as **ints,** and therefore,

```
sizeof ('x')
```

is equivalent to

```
sizeof(int)
```

in C. In C++, however, **char** is not promoted to the size of **int** and therefore

```
sizeof('x')
```

equals

```
sizeof(char)
```

15.10 Declaration of Variables

We know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

Figure 15.3 illustrates this point.

```
int main()
{
   float x;                    // declaration
   float sum = 0;

   for(int i=1; i<5; i++)      // declaration
   {
           cin >> x;
           sum = sum +x;
   }
```

Contd.

```
        float average;              // declaration
        average = sum/(i-1);
        cout << average;

        return 0;
    }
```

Fig. 15.3 *Declaration of Variables*

The only disadvantage of this style of declaration is that we cannot see all the variables used in a scope at a glance.

15.11 Dynamic Initialization of Variables

In C, a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. C++, however, permits initialization of the variables at run time. This is referred to as *dynamic initialization*. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

```
.....
.....
int n = strlen(string);
.....
float area = 3.14159 * rad * rad;
```

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the example of the previous section

```
float average;   // declare where it is necessary
average = sum/i;
```

can be combined into a single statement:

```
float average = sum/i;    // initialize dynamically at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

15.12 Reference Variables

C++ introduces a new kind of variable known as the *reference* variable. A reference variable provides an *alias* (alternative name) for a previously defined variable. For example, if we make the variable **sum** a reference to the variable **total**, then **sum** and **total** can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data-type & reference-name = variable-name
```

Example:

```
float total = 100;
float & sum = total;
```

total is a **float** type variable that has already been declared; **sum** is the alternative name declared to represent the variable **total**. Both the variables refer to the same data object in the memory. Now, the statements

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol **&**. Here, **&** is not an address operator. The notation **float &** means reference to **float**. Other examples are:

```
int n[10];
int & x = n[10];        // x is alias for n[10]
char & a = '\n';        // initialize reference to a literal
```

The variable **x** is an alternative to the array element **n[10]**. The variable **a** is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant **\n** is stored.

The following references are also allowed:

```
i. int x;
   int *p = &x;
   int & m = *p;

ii. int & n = 50;
```

The first set of declarations causes **m** to refer to **x** which is pointed to by the pointer **p** and the statement in (ii) creates an **int** object with value 50 and name **n**.

A major application of reference variables is in passing arguments to functions. Consider the following:

```
void f(int & x)         // uses reference
{
  x = x+10;             // x is incremented; so also m
}
int main()
{
  int m = 10;
  f(m);                 // function call
  .....
  .....
}
```

When the function call **f(m)** is executed, the following initialization occurs:

```
int & x = m;
```

Thus x becomes an alias of m after executing the statement

```
f(m);
```

Such function calls are known as *call by reference*. This implementation is illustrated in Fig. 15.4. Since the variables **x** and **m** are aliases, when the function increments **x, m** is also incremented. The value of **m** becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

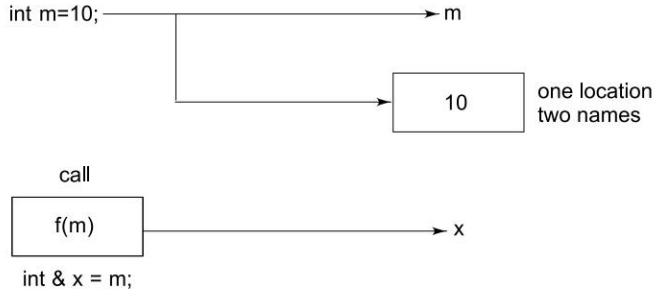


Fig. 15.4 *Call by reference mechanism*

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth. It is also important to note that references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

15.13 Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >>. Other new operators are:

```
::        Scope resolution operator
::*       Pointer-to-member declarator
->*       Pointer-to-member operator
.*        Pointer-to-member operator
delete    Memory release operator
endl      Line feed operator
new       Memory allocation operator
setw      Field width operator
```

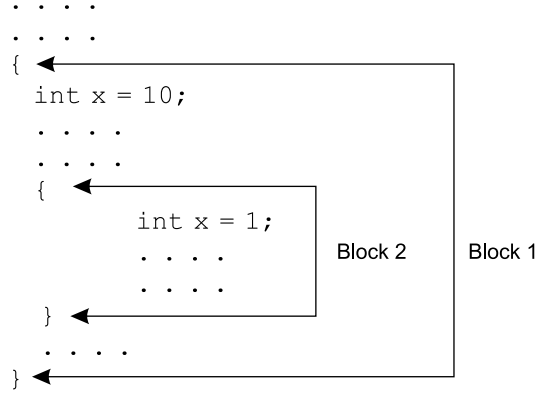
In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

15.14 Scope Resolution Operator

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program:

```
.....
.....
{
   int x = 10;
   .....
   .....
}
.....
.....
{
   int x = 1;
   .....
   .....
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common:



Block2 is contained in block1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of **x** causes it to refer to a different data object. Within the inner block, the variable **x** will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the *scope resolution operator*. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block). The program in Fig. 15.5 illustrates this feature.

Program

```
#include <iostream>
using namespace std;
int m = 10;          // global m

int main()
{
   int m = 20;  // m redeclared, local to main

   {
      int k = m;
      int m = 30;  // m declared again
                   // local to inner block

      cout << "we are in inner block \n";
      cout << "k = " << k << "\n";
      cout << "m = " << m << "\n";
      cout << "::m = " << ::m << "\n";
   }

   cout << "\nWe are in outer block \n";
   cout << "m = " << m << "\n";
   cout << "::m = " << ::m << "\n";

   return 0;
}
```

Output

```
We are in inner block
k = 20
m = 30
::m = 10
We are in outer block
m = 20
::m = 10
```

Fig. 15.5 *Program demonstrating scope resolution operator*

In the above program, the variable **m** is declared at three places, namely, outside the **main()** function, inside the **main()**, and inside the inner block.

Note *It is to be noted* ***::m*** *will always refer to the global* ***m****. In the inner block, ::m refers to the value 10 and not 20.*

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs. This will be dealt in detail later when the classes are introduced.

15.15 Member Dereferencing Operators

As you know, C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators. Table 15.3 shows these operators and their functions.

Table 15.3 ***Member dereferencing operators***

Operator	Function
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
–>*	To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, and therefore we defer the use of member dereferencing operators until then.

15.16 Memory Management Operators

C uses **malloc()** and **calloc()** functions to allocate memory dynamically at run time. Similarly, it uses the function **free()** to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as *free store* operators.

An object can be created by using **new**, and destroyed by using **delete**, as and when required. A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**. Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
pointer-variable = new data-type;
```

Here, *pointer-variable* is a pointer of type *data-type*. The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object. The *data-type* may be any valid data type. The *pointer-variable* holds the address of the memory space allocated. Examples:

```
p = new int;
q = new float;
```

where **p** is a pointer of type **int** and **q** is a pointer of type **float**. Here, **p** and **q** must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;
float *q = new float;
```

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

We can also initialize the memory using the **new** operator. This is done as follows:

```
pointer-variable = new data-type(value);
```

Here, value specifies the initial value. Examples:

```
int *p = new int(25);
float *q = new float(7.5);
```

As mentioned earlier, **new** can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

```
pointer-variable = new data-type[size];
```

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

When creating multi-dimensional arrays with **new**, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4];    // legal
array_ptr = new int[m][5][4];    // legal
array_ptr = new int[3][5][ ];    // illegal
array_ptr = new int[ ][5][4];    // illegal
```

The first dimension may be a variable whose value is supplied at runtime. All others must be constants.

The application of **new** to class objects will be discussed later in Chapter 17.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer-variable;
```

The *pointer-variable* is the pointer that points to a data object created with **new**. Examples:

```
delete p;
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of **delete**:

```
delete [size] pointer-variable;
```

The *size* specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ]p;
```

will delete the entire array pointed to by **p**.

What happens if sufficient memory is not available for allocation? In such cases, like **malloc(), new** returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows:

```
.....
.....
p = new int;
if(!p)
{
```

```
    cout << "allocation failed \n";
}
.....
.....
```

The **new** operator offers the following advantages over the function **malloc()**.

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.

15.17 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement

```
.....
.....
cout  << "m = " << m << endl
      << "n = " << n << endl
      << "p = " << p << endl;
.....
.....
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:

```
m =  2597
n =  14
p =  175
```

It is important to note that this form is not the ideal output. It should rather appear as under:

```
m = 2597
n =  14
p =  175
```

Here, the numbers are *right-justified*. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout << setw(5) << sum << endl;
```

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

		3	4	5

The program in Fig. 15.6 illustrates the use of **endl** and **setw**.

Program

```
#include <iostream>
#include <iomanip>  // for setw

using namespace std;

int main()
{
   int Basic = 950, Allowance = 95, Total = 1045;

   cout  << setw(10) << "Basic" << setw(10) << Basic << endl
         << setw(10) << "Allowance" << setw(10) << Allowance << endl
         << setw(10) << "Total" << setw(10) << Total << endl;

   return 0;
}
```

Output

```
     Basic       950
 Allowance        95
     Total      1045
```

Fig. 15.6 *Program illustrating use of manipulators*

Note *Character strings are also printed right-justified.*

We can also write our own manipulators as follows:

```
#include <iostream>
ostream & symbol(ostream & output)
{
   return output << "\tRs";
}
```

The **symbol** is the new manipulator which represents **Rs**. The identifier **symbol** can be used whenever we need to display the string **Rs.**

15.18 Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression   // C notation
type-name (expression)   // C++ notation
```

Examples:

```
average = sum/(float)i;   // C notation
average = sum/float(i);   // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.

```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;
p = int_pt(q);
```

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

15.19 Expressions and Their Types

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as *compound expressions*.

15.19.1 Constant Expressions

Constant Expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

15.19.2 Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m
m * n - 5
m * 'x'
5 + int(2.0)
```

where **m** and **n** are integer variables.

15.19.3 Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results. Examples:

```
x + y
x * y / 10
5 + float(10)
10.75
```

where **x** and **y** are floating-point variables.

15.19.4 Pointer Expressions

Pointer Expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

15.19.5 Relational Expressions

Relational Expressions yield results of type **bool** which takes a value **true** or **false**. Examples:

```
x <= y
a+b == c+d
m+n > 100
```

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as *Boolean expressions*.

15.19.6 Logical Expressions

Logical Expressions combine two or more relational expressions and produces **bool** type results. Examples:

```
a>b && x==10
x==10 || y==5
```

15.19.7 Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3        // Shift three bit position to left
y >> 1        // Shift one bit position to right
```

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what are termed as *operator keywords* that can be used as alternative representation for operator symbols.

15.20 Special Assignment Expressions

15.20.1 Chained Assignment

```
x = (y = 10);
   or
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a = b = 12.34;          // wrong
```

is illegal. This may be written as

```
float a=12.34, b=12.34        // correct
```

15.20.2 Embedded Assignment

```
x = (y = 50) + 10;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10 = 60 is assigned to x. This statement is identical to

```
y = 50;
x = y + 10;
```

15.20.3 Compound Assignment

Like C, C++ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

```
variable1 = variable1 op variable2;
```

15.21 Implicit Conversions

We can mix data types in expressions. For example,

```
m = 5+2.75;
```

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as *implicit* or *automatic conversion*.

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the "smaller" type is

converted to the "wider" type. For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int**. The "water-fall" model shown in **Fig. 15.7** illustrates this rule.

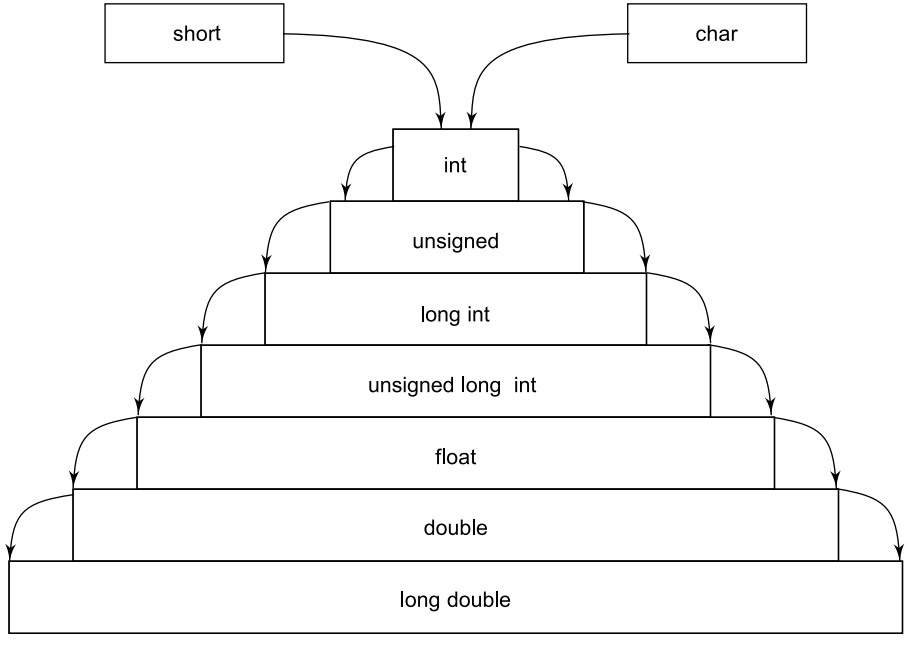


Fig. 15.7 *Water-fall model of type conversion*

Whenever a **char** or **short int** appears in an expression, it is converted to an **int**. This is called *integral widening conversion*. The implicit conversion is applied only after completing all integral widening conversions.

Table 15.4 ***Results of Mixed-mode Operations***

RHO / LHO	char	short	int	long	float	double	long double
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand
LHO – Left-hand operand

15.22 Operator Overloading

As stated earlier, overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. Actually, we have used the concept of overloading in C also. For example, the operator * when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers. The number and type of operands decide the nature of operation to follow.

The input/output operators << and >> are good examples of operator overloading. Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header file *iostream* where a number of overloading definitions for << are included. Thus, the statement

```
cout << 75.86;
```

invokes the definition for displaying a **double** type value, and

```
cout << "well done";
```

invokes the definition for displaying a **char** value. However, none of these definitions in *iostream* affect the built-in meaning of the operator.

Similarly, we can define additional meanings to other C++ operators. For example, we can define + operator to add two structures or objects. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (**.** and **.***), conditional operator (?:), scope resolution operator (::) and the size operator **(sizeof)**.

15.23 Operator Precedence

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. Table 15.5 gives the precedence and associativity of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels *prefix* and *postfix* distinguish the uses of ++ and --. Also, the symbols +, –, *, and & are used as both unary and binary operators.

Table 15.5 ***Operator precedence and associativity***

Operator	Associativity
::	left to right
–> . ( ) [ ] postfix ++ postfix – –	left to right
prefix ++ prefix – – ~ ! unary + unary –	
unary * unary & (type) sizeof new delete	right to left
–> * *	left to right
* / %	left to right
+ –	left to right
<< >>	left to right

Contd.

Table 15.5 *(Contd.)*

Operator	Associativity
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^= |=	left to right
, (comma)	

The unary operations assume higher precedence.

15.24 Control Structures

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple to comprehend, while others are not. Experience has also shown that the number of bugs that occur is related to the format of the program. The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures:

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition)

Figure 15.8 shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.

It is important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in Figs 15.9 (a), (b) and (c).

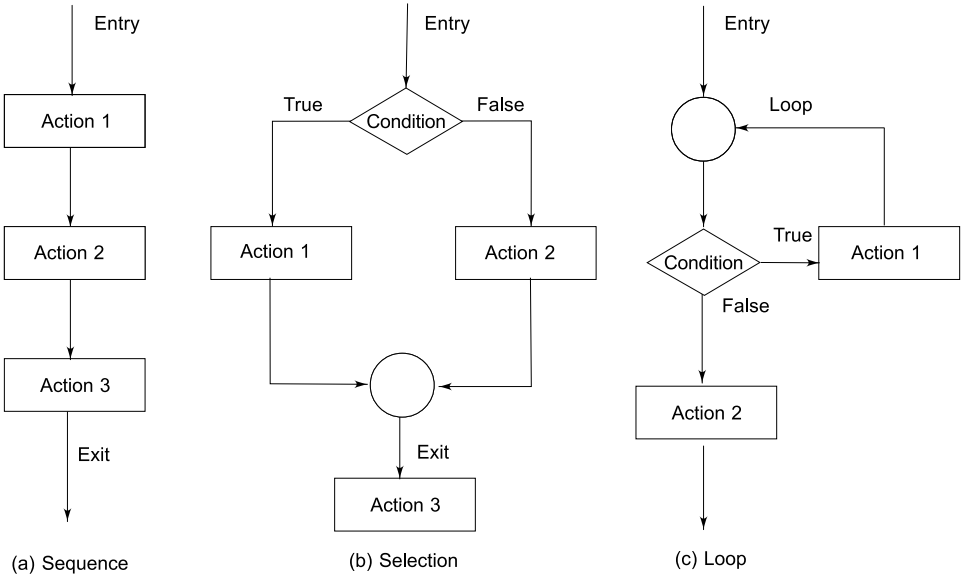


Fig. 15.8 *Basic control structures*

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in Fig. 15.10. This shows that C++ combines the power of structured programming with the object-oriented paradigm.

15.24.1 The if Statement

The **if** statement is implemented in two forms:

- Simple **if** statement
- **if...else** statement

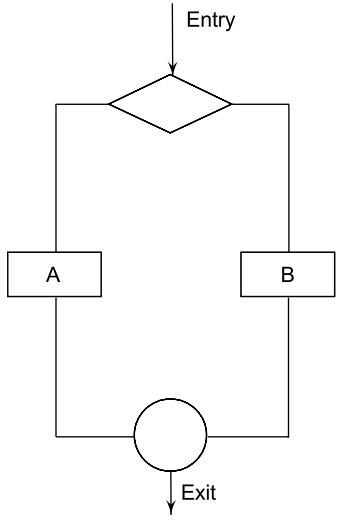
Examples:

Form 1

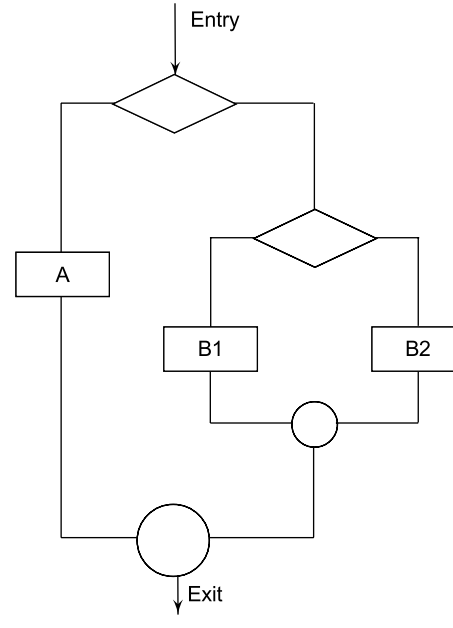
```
if(expression is true)
{
   action1;
}
action2;
action3;
```

Form 2

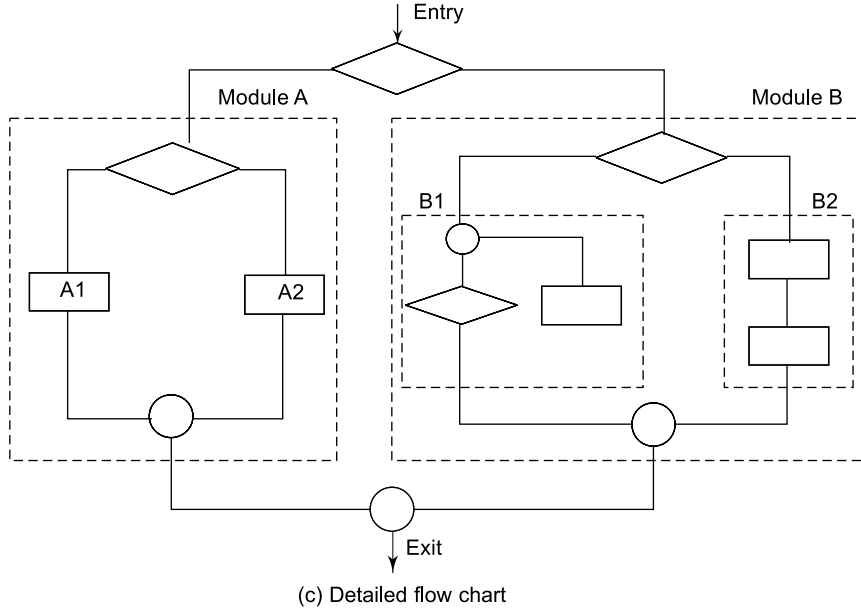
```
if(expression is true)
{
action1;
}
else
{
action2;
}
action3;
```



(a) First level of abstraction



(b) Second level of abstraction



(c) Detailed flow chart

Fig. 15.9 *Different levels of abstraction*

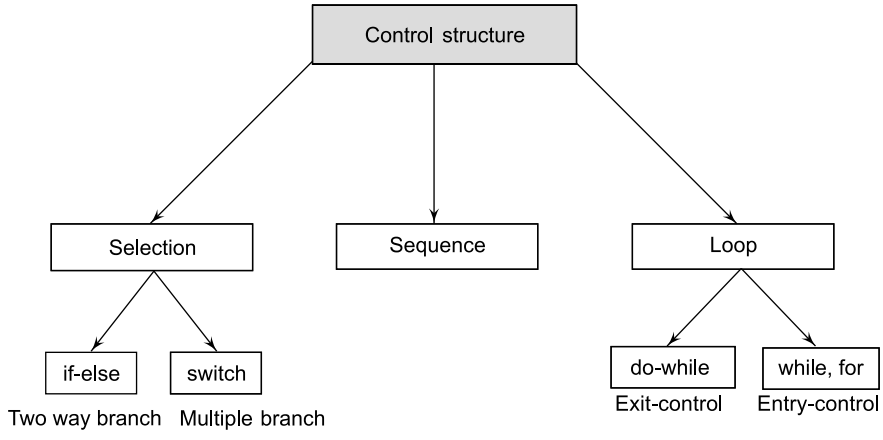


Fig. 15.10 *C++ Statements to implement in two forms*

15.24.2 The Switch Statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
{
   case1:
   {
   action1;
   }
   case2:
   {
   action2;
   }
   case3:
   {
   action3;
   }
   default:
   {
   action4;
   }
}
action5;
```

15.24.3 The do-while Statement

The **do-while** is an *exit-controlled* loop. Based on a condition,the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
   action1;
}
while(condition is true);
action2;
```

15.24.4 The while Statement

This is also a loop structure, but is an *entry-controlled* one. The syntax is as follows:

```
while(condition is true)
{
   action1;
}
action2;
```

15.24.5 The for Statement

The **for** is an *entry-entrolled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
   action1;
}
action2;
```

The syntax of the control statements in C++ is very much similar to that of C and therefore they are implemented as and when they are required.

Just Remember

- C++ provides various types of tokens that include keywords, identifiers, constants, strings, and operators.
- Identifiers refer to the names of variables, functions, arrays, classes, etc.
- C++ provides an additional use of **void**, for declaration of generic pointers.
- The enumerated data types differ slightly in C++. The tag names of the enumerated data types become new type names. That is, we can declare new variables using these tag names.
- In C++, the size of character array should be one larger than the number of characters in the string.
- C++ adds the concept of constant pointer and pointer to constant. In case of constant pointer we can not modify the address that the pointer is initialized to. In case of pointer to a constant, contents of what it points to cannot be changed.
- Pointers are widely used in C++ for memory management and to achieve polymorphism.
- C++ provides a qualifier called **const** to declare named constants which are just like variables except that their values can not be changed. A **const** modifier defaults to an **int**.
- C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is type casting.

- C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- A reference variable provides an alternative name for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value of the other variable.
- A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.
- A major application of the scope resolution (::) operator is in the classes to identify the class to which a member function belongs.
- In addition to **malloc(), calloc()** and **free()** functions, C++ also provides two unary operators, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.
- C++ also provides manipulators to format the data display. The most commonly used manipulators are **endl** and **setw**.
- C++ supports seven types of expressions. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.
- C++ also permits explicit type conversion of variables and expressions using the type cast operators.
- Like C, C++ also supports the three basic control structures namely, sequence, selection and loop, and implements them using various control statements such as, **if, if...else, switch, do.. while, while** and **for**.

Review Questions

15.1 Enumerate the rules of naming variables in C++. How do they differ from ANSI C rules?
15.2 An unsigned int can be twice as large as the signed int. Explain how?
15.3 Why does C++ have type modifiers?
15.4 What are the applications of void data type in C++?
15.5 Can we assign a void pointer to an int type pointer? If not, why? How can we achieve this?
15.6 Describe, with examples, the uses of enumeration data types.
15.7 Describe the differences in the implementation of enum data type in ANSI C and C++.
15.8 Why is an array called a derived data type?
15.9 The size of a char array that is declared to store a string should be one larger than the number of characters in the string. Why?
15.10 The const was taken from C++ and incorporated in ANSI C, although quite differently. Explain.
15.11 How does a constant defined by const differ from the constant defined by the preprocessor statement # define?
15.12 In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?
15.13 What do you mean by dynamic initialization of a variable? Give an example.
15.14 What is a reference variable? What is its major use?
15.15 List at least four new operators added by C++ which aid OOP.
15.16 What is the application of the scope resolution operator :: in C++?

15.17 What are the advantages of using new operator as compared to the function malloc()?
15.18 Illustrate with an example, how the setw manipulator works.
15.19 How do the following statements differ?
(a) char * const p;
(b) char const *p;

Programming Exercises

15.1 Write a function using reference variables as arguments to swap the values of a pair of integers.
15.2 Write a function that creates a vector of user-given size **M** using **new** operator.
15.3 Write a program to print the following output using for loops.

```
1
22
333
4444
55555
. . . . . .
```

15.4 Write a program to evaluate the following investment equation

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

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

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

r: 0.10, 0.11, 0.12, ..., 0.20

n: 1, 2, 3, ..., 10

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

$V = P(1 + r)$

$P = V$

In other words, the value of money at the end of the first year becomes the principal amount for the next year, and so on.

15.5 An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and **count** the votes cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot', and the program should also count the number of spoilt ballots.
15.6 A cricket team has the following table of batting figures for a series of test matches:

Player's name	Runs	Innings	Times not out
Sachin	8430	230	18
Saurav	4200	130	9
Rahul	3350	105	11
.	.	.	.
.	.	.	.

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

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

(a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$

(b) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \cdots \cdots$

(c) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$

15.8 Write a program to print a table of values of the function

$y = e^{-x}$

for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

Table For *Y* = *EXP* [–*X*]

X	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.0									
1.0									
.									
.									
.									
9.0									

15.9 Write a program to calculate the variance and standard deviation of N numbers.

$$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N}(x_1 - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_1$

15.10 An electricity board charges the following rates to domestic users to discourage large consumption of energy:

For the first 100 units - 60P per unit
For next 200 units - 80P per unit
Beyond 300 units - 90P per unit

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

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

Debugging Exercises

15.1 What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
        int i=0;
        i=400*400/400;
        cout << i;
}
```

15.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
      int num[]={1,2,3,4,5,6};
      num[1]==[1]num ?  cout<<"Success"  :   cout<<"Error";
}
```

15.3 Identify the errors in the following program.

```
#include <iostream.h>
void main()
{
      int i=5;
      while(i)
      {
         switch(i)
         {
         default:
         case 4:
         case 5:
         break;
         case 1:
         continue;
         case 2:
         case 3:
         break;
         }
         i–;
   }
}
```

15.4 Identify the error in the following program.

```
#include <iostream.h>
#define pi 3.14
int squareArea(int &);
int circleArea(int &);

void main()
{
```

```
        int a=10;
        cout << squareArea(a) << " ";
        cout << circleArea(a) << " ";
        cout << a << endl;
    }

int squareArea(int &a)
{
   return a *== a;
      }

int circleArea(int &r)
{
   return r = pi * r * r;
}
```

15.5 Identify the error in the following program.

```
#include <iostream.h>
#include <malloc.h>

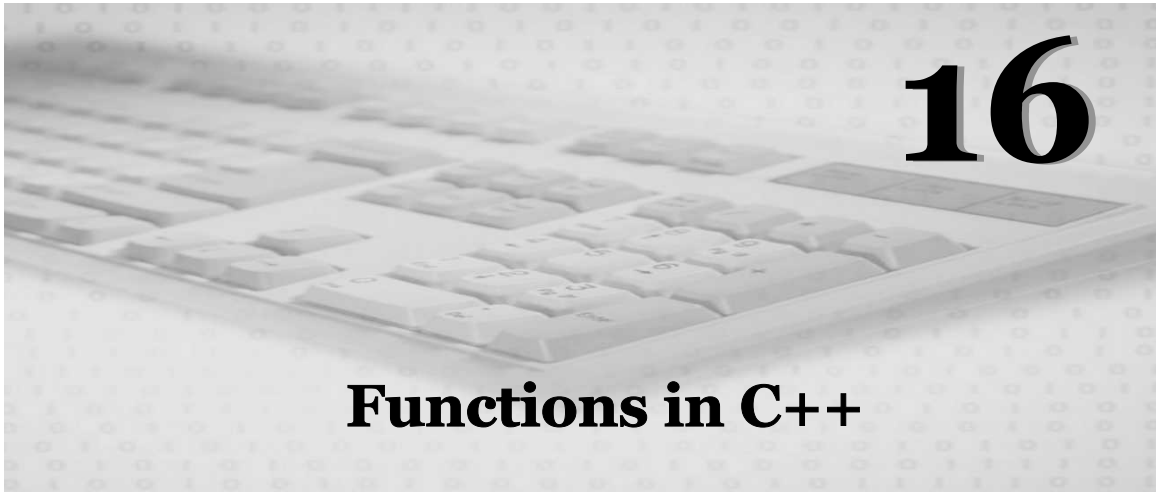
char* allocateMemory();

void main()
{
      char* str;
      str = allocateMemory();
      cout << str;
      delete str;
      str = "   ";
      cout << str;
}

char* allocateMemory()
{
         str = "Memory allocation test, ";
         return str;
}
```

15.6 Find errors, if any, in the following C++ statements.

(a) long float x;
(b) char *cp = vp; // vp is a void pointer
(c) int code = three; // three is an enumerator
(d) int *p = new; // allocate memory with new
(e) enum (green, yellow, red);
(f) int const *p = total;
(g) const int array_size;
(h) for (i=1; int i<10; i++) cout << i << "\n";
(i) int & number = 100;
(j) float *p = new int [10];
(k) int public = 1000;



16 Functions in C++

16.1 Introduction

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Recall that we have used a syntax similar to the following in developing C programs.

```
void show();    /* Function declaration */
main()
{
      .....
      show();   /* Function call */
      .....
}
void show()     /* Function definition */
{
      .....
      .....     /* Function body */
      .....
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different

tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

In this chapter, we shall briefly discuss the various new features that are added to C++ functions and their implementation.

16.2 The Main Function

C does not specify any return type for the **main()** function which is the starting point for the execution of a program. The definition of **main()** would look like this:

```
main()
{
        // main program statements
}
```

This is perfectly valid because the **main()** in C does not return any value.

In C++, the **main()** returns a value of type **int** to the operating system. C++, therefore, explicitly defines **main()** as matching one of the following prototypes:

```
int main();
int main(int argc, char * argv[]);
```

The functions that have a return value should use the **return** statement for termination. The **main()** function in C++ is, therefore, defined as follows:

```
int main()
{
      ......
      ......
      return 0;
}
```

Since the return type of functions is **int** by default, the keyword **int** in the **main()** header is optional. Most C++ compilers will generate an error or warning if there is no **return** statement. Turbo C++ issues the warning

```
Function should return a value
```

and then proceeds to compile the program. It is good programming practice to actually return a value from **main()**.

Many operating systems test the return value (called *exit value*) to determine if there is any problem. The normal convention is that an exit value of zero means the program ran successfully, while a nonzero value means there was a problem. The explicit use of a **return(0)** statement will indicate that the program was successfully executed.

16.3 Function Prototyping

Function *prototyping* is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ by Stroustrup and the success of this feature inspired the ANSI C committee to adopt it. However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a *declaration statement* in the calling program and is of the following form:

```
type function-name (argument-list);
```

The *argument-list* contains the types and names of arguments that must be passed to the function. Example:

```
float volume(int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float volume(int x, float y, z);
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. That is, the form

```
float volume(int, float, float);
```

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the *function call or function definition.*

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float c)
{
       float v = a*b*c;
       .....
       .....
}
```

The function **volume()** can be invoked in a program as follows:

```
float cube1 = volume(b1,w1,h1);  // Function call
```

The variable **b1, w1**, and **h1** are known as the actual parameters which specify the dimensions of **cube1**. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an *empty argument list*, as in the following example:

```
void display( );
```

In C++, this means that the function does not pass any parameters. It is identical to the statement

```
void display(void);
```

However, in C, an empty parentheses implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something(...);
```

16.4 Call by Reference

In traditional C, a function call passes arguments by value. The *called function* creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the *calling program*. For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for *bubble sort*, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by-value method is used.

Provision of the *reference variables* in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int &a,int &b) // a and b are reference variables
{
      int t = a;         // Dynamic initialization
      a = b;
      b = t;
}
```

Now, if **m** and **n** are two integer variables, then the function call

```
swap(m, n);
```

will exchange the values of **m** and **n** using their aliases (reference variables) **a** and **b**. Reference variables have been discussed in detail in Chapter 15. In traditional C, this is accomplished using *pointers* and *indirection* as follows:

```
void swap1(int *a, int *b)  /* Function definition */
{
       int t;
       t = *a;    /* assign the value at address a to t */
       *a = *b;   /* put the value at b into a  */
       *b = t;       /* put the value at t into b  */
}
```

This function can be called as follows:

```
swap1(&x, &y);      /* call by passing */
                    /* addresses of variables */
```

This approach is also acceptable in C++. Note that the call-by-reference method is neater in its approach.

16.5 Return by Reference

A function can also return a reference. Consider the following function:

```
int & max(int &x,int &y)
{
      if (x > y)
         return x;
      else
         return y;
}
```

Since the return type of **max()** is **int &**, the function returns reference to **x** or **y** (and not the values). Then a function call such as **max(a, b)** will yield a reference to either **a** or **b** depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a,b) = -1;
```

is legal and assigns -1 to **a** if it is larger, otherwise -1 to **b**.

16.6 Inline Functions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as *macros*. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function*. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion). The inline functions are defined as follow:

```
inline function-header
{
   function body
}
```

Example:

```
inline double cube(double a)
{
   return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+1.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword **inline** to the function definition. All inline functions must be defined before they are called.

We should exercise care before making a function **inline**. The speed benefits of **inline** functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of **inline** functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube(double a) {return(a*a*a);}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.

Note *Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So, a trade-off becomes necessary.*

Program

```
#include <iostream>

using namespace std;

   inline float mul(float x, float y)
   {
      return(x*y);
   }

   inline double div(double p, double q)
   {
      return(p/q);
   }

int main()
{
   float a = 12.345;
   float b = 9.82;

   cout << mul(a,b) << "\n";
   cout << div(a,b) << "\n";

   return 0;
}
```

output

```
121.228
1.25713
```

Fig. 16.1 *Program showing use of inline functions.*

16.7 Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default value* to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with default values:

```
float amount(float principal,int period,float rate=0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument **rate**. A subsequent function call like

```
value = amount(5000,7);      // one argument missing
```

passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for rate. The call

```
value = amount(5000,5,0.12);    // no missing argument
```

passes an explicit value of 0.12 to **rate**.

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i, int j=5, int k=10);       //    legal
int mul(int i=5, int j);                 //    illegal
int mul(int i=0, int j, int k=10);       //    illegal
int mul(int i=2, int j=5, int k=10);     //    legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. The program in Fig. 16.2 illustrates the use of default arguments.

Program

```
#include <iostream>

using namespace std;

int main()
{
   float amount;

   float value(float p, int n, float r=0.15);      // prototype
   void printline(char ch='*', int len=40);        // prototype

   printline();           // uses default values for arguments

   amount = value(5000.00,5);           // default for 3rd argument

   cout << "\n       Final Value = " << amount << "\n\n";

   printline('=');     // use default value for 2nd argument

   return 0;
}
/*-------------------------------------------------------*/
float value(float p, int n, float r)
{
```

Contd.

```
        int year = 1;
        float sum = p;

        while(year <= n)
        {
            sum = sum*(1+r);
            year = year+1;
        }
        return(sum);
    }

    void printline(char ch, int len)
    {
        for(int i=1; i<=len; i++) printf("%c",ch);
        printf("\n");
    }
Output
    **********************************

            Final Value = 10056.8

    ==================================
```

Fig.16.2 *Program showing use of default arguments*

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

16.8 const Arguments

In C++, an argument to a function can be declared as const as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

16.9 Function Overloading

As stated earlier, *overloading* refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking

the number and type of the arguments but not on the function type. For example, an overloaded **add()** function handles different types of data as shown below:

```
// Declarations
int add(int a, int b);              // prototype 1
int add(int a, int b, int c);       // prototype 2
double add(double x, double y);     // prototype 3
double add(int p, double q);        // prototype 4
double add(double p, int q);        // prototype 5

// Function calls
cout << add(5, 10);          // uses prototype 1
cout << add(15, 10.0);       // uses prototype 4
cout << add(12.5, 7.5);      // uses prototype 3
cout << add(5, 10, 15);      // uses prototype 2
cout << add(0.75, 5);        // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   ```
   char to int
   float to double
   ```

   to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   ```
   long square(long n)
   double square(double x)
   ```

   A function call such as

   ```
   square(10)
   ```

   will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program

```
// Function volume() is overloaded three times

#include <iostream>

using namespace std;

// Declarations (prototypes)
int volume(int);
double volume(double, int);
long volume(long, int, int);

int main()
{
      cout << volume(10) << "\n";
      cout << volume(2.5,8) << "\n";
      cout << volume(100L,75,15) << "\n";

      return 0;
}

// Function definitions

int volume(int s)  // cube
{
      return(s*s*s);
}

double volume(double r, int h)   // cylinder
{
      return(3.14519*r*r*h);
}

long volume(long l, int b, int h) // rectangular box
{
      return(l*b*h);
}
```

Output

```
1000
157.26
112500
```

Fig. 16.3 *Program showing use of function overloading.*

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks. Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects. They will be illustrated later when the classes are discussed in the next chapter.

16.10 Friend and Virtual Functions

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. Therefore, discussions on these functions have been reserved until after the class objects are discussed. The friend functions are discussed in Sec. 17.15 of the next chapter .

16.11 Math Library Functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized in Table 16.1.

Table 16.1 ***Commonly used math library functions***

Function	Purposes
ceil(x)	Rounds x to the smallest integer not less than x ceil(8.1) = 9.0 and ceil(-8.8) = –8.0
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x.
fabs(x)	Absolute value of x.
	If x>0 then abs(x) is x
	If x=0 then abs(x) is 0.0
	If x<0 then abs(x) is –x
floor(x)	Rounds x to the largest integer not greater than x floor(8.2) = 8.0 and floor(–8.8 = –9.0
log(x)	Natural logarithm of x(base e)
log10(x)	Logarithm of x(base 10)
pow(x,y)	x raised to power y(x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)

Note *The argument variables **x** and **y** are of type **double** and all the functions return the data type **double**.*

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

Just Remember

- It is possible to reduce the size of program by calling and using functions at different places in the program.

- In C++ the main() returns a value of type **int** to the operating system. Since the return type of functions is **int** by default, the keyword **int** in the main() header is optional. Most C++ compilers issue a warning, if there is no return statement.
- Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.
- Reference variables in C++ permit us to pass parameters to the functions by reference. A function can also return a reference to a variable.
- When a function is declared inline the compiler replaces the function call with the respective function code. Normally, a small size function is made as **inline**.
- The compiler may ignore the inline declaration if the function declaration is too long or too complicated and hence compile the function as a normal function.
- C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.
- In C++, an argument to a function can be declared as **const**, indicating that the function should not modify the argument.
- C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.
- C++ supports two new types of functions, namely **friend** functions and **virtual** functions.
- Many mathematical computations can be carried out using the library functions supported by the C++ standard library.

Review Questions

16.1 State whether the following statements are TRUE or FALSE.

(a) A function argument is a value returned by the function to the calling program.
(b) When arguments are passed by value, the function works with the original arguments in the calling program.
(c) When a function returns a value, the entire function call can be assigned to a variable.
(d) A function can return a value by reference.
(e) When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.
(f) It is not necessary to specify the variable name in the function prototype.

16.2 What are the advantages of function prototypes in C++?

16.3 Describe the different styles of writing prototypes.

16.4 Find errors, if any, in the following function prototypes.

(a) float average(x,y);
(b) int mul(int a,b);
(c) int display(...);

(d) void Vect(int? &V, int & size);
(e) void print(float data [], size = 20);

16.5 What is the main advantage of passing arguments by reference?
16.6 When will you make a function inline? Why?
16.7 How does an inline function differ from a preprocessor macro?
16.8 When do we need to use default arguments in a function?
16.9 What is the significance of an empty parenthesis in a function declaration?
16.10 What do you meant by overloading of a function? When do we use this concept?
16.11 Comment on the following function definitions:

(a)
```
int *f( )
{
    int m = 1;
    ....
    ....
    return(&m);
}
```

(b)
```
double f( )
{
    ....
    ....
    return(1);
}
```

(c)
```
int & f()
{
    int n = 10;
    ....
    ....
    return(n);
}
```

Programming Exercises

16.1 Write a function to read a matrix of size m x n from the keyboard.
16.2 Write a program to read a matrix of size m x n from the keyboard and display the same on the screen. Use the functions discudssed in Exercises 4.12
16.3 Rewrite the program of Exercise 16.2 to make the row parameter of the matrix as a default argument.
16.4 The effect of a default argument can be alternatively achieved by overloading. Discuss with an example.

16.5 Write a macro that obtains the largest of three numbers.

16.6 Redo Exercise 16.16 using inline function. Test the function using a **main** program.

16.7 Write a function **power()** to raise a number **m** to a power **n**. The function takes a **double** value for **m** and **int** value for **n**, and returns the result correctly. Use a default value of 2 for n to make the function to calculate squares when this argument is omitted. Write a **main** that gets the values of **m** and **n** from the user to test the function.

16.8 Write a function that performs the same operation as that of Exercise 16.18 but takes an int value for **m**. Both the functions should have the same name. Write a **main** that calls both the functions. Use the concept of function overloading.

Debugging Exercises

16.1 Identify the error in the following program:

```
#include <iostream.h>
int fun()
{
      return 1;
}
float fun()
{
      return 10.23;
void main()
{
         cout << (int)fun() << ' ';
         cout << (float)fun() << ' ';
}
```

16.2 Identify the error in the following program:

```
#include <iostream.h>
   void display(const int const1=5)
   {
      const int const2=5;
      int array1[const1];
      int array2[const2];
      for(int i=0; i<5; i++)
      {
         array1[i] = i;
         array2[i] = i*10;
         cout << array1[i] << ' ' << array2[i] << ' ' ;
      }
}

void main()
{
      display(5);
}
```

16.3 Identify the error in the following program:

```
#include <iostream.h>
int gValue=10;
void extra()
{
         cout << gValue << ’   ‘;
}
void main()
{
         extra();
         {
                  int gValue = 20;
                  cout << gValue << ‘ ‘;
                  cout << : gValue << ‘ ‘;
         }
 }
```

16.4 Find errors, if any, in the following function definition for displaying a matrix: void display(int A[ ] [ ], int m, int n)

```
{
      for(i=0; i<m; i++)
         for(j=0; j<n; j++)
            cout << “ “ << A[i][j];
         cout << “\n”;
   }
```



17 Classes and Objects

17.1 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.

17.2 C Structures Revisited

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
   char   name[20];
   int    roll_number;
   float total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* or *elements*. The identifier student, which is referred to as *structure name* or *structure tag*, can be used to create variables of type student. Example:

```
struct student A;  // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period operator* as follows:

```
strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks = 595.5;
Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

17.2.1 Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
      float x;
      float y;
};     struct complex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding*. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

17.2.2 Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

```
student A;    // C++ declaration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most

of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

Note *The only difference between a structure and a class in C++ is that, by default, the members of a class are private, while, by default, the members of a structure are public.*

17.3 Specifying a Class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
        private:
          variable declarations;
          function declarations;
        public:
          variable declarations;
          function declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are **private**. If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as *data members* and the functions are known as *member functions*. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in Fig. 17.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

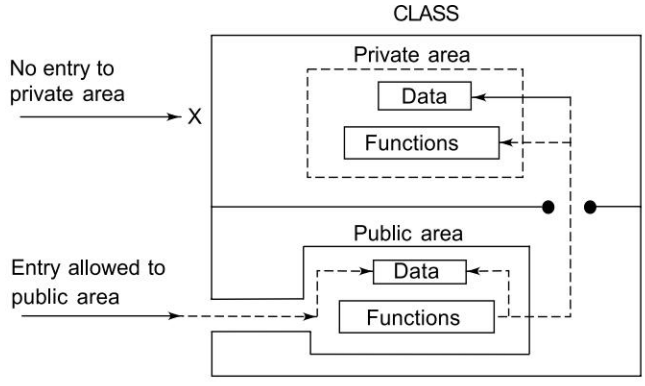


Fig. 17.1 *Data hiding in classes*

17.3.1 A Simple Class Example

A typical class declaration would look like:

```
class item
{
      int number;                    // variables declaration
      float cost;                    // private by default
  public:
      void getdata(int a, float b);  // functions declaration
      void putdata(void);            // using prototype
   };

      // ends with semicolon
```

We usually give a class some meaningful name, such as **item**. This name now becomes a new type identifier that can be used to declare *instances* of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables number and cost, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**. Figure 17.2 shows two different notations used by the OOP analysts to represent a class.

17.3.2 Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies *what* they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

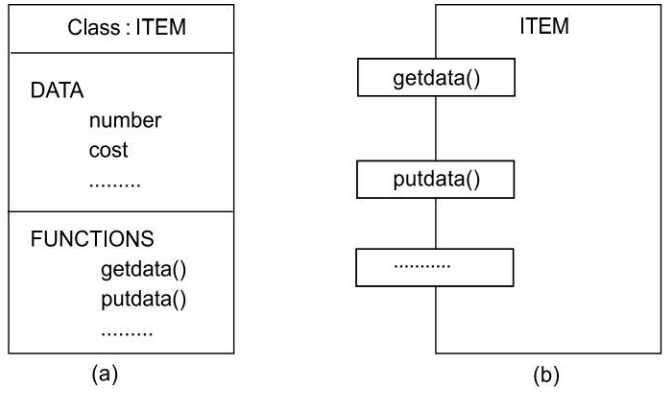


Fig. 17.2 *Representation of a class*

```
item x;          // memory for x is created
```

creates a variable **x** of type **item**. In C++, the class variables are known as *objects*. Therefore, **x** is called an object of type **item**. We may also declare more than one object in one statement. Example:

```
item x, y, z;
```

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a *template* and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class item
{
        .....
        .....
        .....
} x,y,z;
```

would create the objects **x, y** and **z** of type **item**. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

17.3.3 Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual function. Please refer Sec. 17.4 for further details.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```

sends a message to the object **x** requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
      int x;
      int y;
  public:
      int z;
};
      .....
      .....
xyz p;
p.x = 0;     // error, x is private
p.z = 10     // OK, z is public
      .....
      .....
```

Note *The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.*

17.4 Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

17.4.1 Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the *ANSI prototype* form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which **class** the function belongs to. The general form of a member function definition is:

```
return-type class-name :: function-name (argument declaration)
  {
      Function body
  }
```

The membership label class-name :: tells the compiler that the function *function-name* belongs to the class *class-name*. That is, the scope of the function is restricted to the *class-name* specified in the header line. The symbol :: is called the *scope resolution* operator.

For instance, consider the member functions **getdata()** and **putdata()** as discussed above. They may be coded as follows:

```
void item :: getdata(int a, float b)
{
     number = a;
     cost = b;
}
void item :: putdata(void)
{
     cout << "Number :" << number << "\n";
     cout << "Cost   :" << cost   << "\n";
}
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in the program development. These characteristics are :

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)
- A member function can call another member function directly, without using the dot operator.

17.4.2 Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
     int number;
     float cost;
  public:
     void getdata(int a, float b);   // declaration
          // inline function
     void putdata(void)              // definition inside the class
     {
          cout << number  <<  "\n";
          cout << cost    <<  "\n";
     }
 };
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an **inline** function are also applicable here. Normally, only small functions are defined inside the class definition.

17.5 A C++ Program with Class

All the details discussed so far are implemented in Fig. 17.3.

Program

```
#include <iostream>
using namespace std;
class item
{
   int number;     // private by default
   float cost;     // private by default
   public:
   void getdata(int a, float b);    // prototype declaration,
                                    // to be defined
   // Function defined inside class
   void putdata(void)
   {
      cout << "number :" << number << "\n";
      cout << "cost   :" << cost   << "\n";
   }
};
//............ Member Function Definition ..............
```

Contd.

```
void item :: getdata(int a, float b)   // use membership label
{
   number = a;     // private variables
   cost = b;       // directly used
}
//..................... Main Program .....................
int main()
{
   item x;  // create object x
   cout <<  "\nobject x "  <<  "\n";
   x.getdata(100, 299.95);                  // call member function
   x.putdata();                             // call member function

   item y;                                  // create another object
   cout <<  "\nobject y"  << "\n";
   y.getdata(200, 175.50);
   y.putdata();
   return 0;
}
```

Output

```
object x
number :100
cost   :299.95

object y
number :200
cost   :175.5
```

Fig.17.3 *Program showing class implementation*

This program features the class **item**. This class contains two private variables and two public functions. The member function **getdata()** which has been defined outside the class supplies values to both the variables. Note the use of statements such as

```
number = a;
```

in the function definition of **getdata()**. This shows that the member functions can have direct access to private data items.

The member function **putdata()** has been defined inside the class and therefore behaves like an **inline** function. This function displays the values of the private variables **number** and **cost**.

The program creates two objects, x and y in two different statements. This can be combined in one statement.

```
item x, y;   // creates a list of objects
```

For the sake of illustration we have shown one member function as **inline** and the other as an 'external' member function. Both can be defined as **inline** or external functions.

17.6 Making an Outside Function Inline

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** in the header line of function definition. Example:

```
class item
{
        .....
        .....
  public:
        void getdata(int a, float b);            // declaration
};
inline void item :: getdata(int a, float b)      // definition
{
        number = a;
        cost = b;
}
```

17.7 Nesting of Member Functions

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as *nesting of member functions*. Fig. 17.4 illustrates this feature.

Program

```
#include <iostream>
using namespace std;
class set
{
   int m, n;
  public:
   void input(void);
   void display(void);
   int largest(void);
};
int set :: largest(void)
{
   if(m >= n)
      return(m);
   else
      return(n);
}
   void set :: input(void)
```

Cont'd...

```
        {
            cout << "Input values of m and n" << "\n";
            cin >> m >> n;
    }
    void set :: display(void)
    {
        cout << "Largest value = "
             << largest() << "\n";      // calling member function
    }
    int main()
    {
        set A;
        A.input();
        A.display();
        return 0;
    }
```

Output

```
Input values of m and n
25 18
Largest value = 25
```

Fig. 17.4 *Program showing nesting of member functions*

17.8 Private Member Functions

Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
      int m;
      void read(void);       // private member function
  public:
      void update(void);
      void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();    // won't work; objects cannot access
              // private members
```

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
      read();   // simple call; no object used
}
```

17.9 Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;     // provides value for array size
class array
{
   int a[size];        // 'a' is int type array
  public:
      void setval(void);
      void display(void);
};
```

The array variable **a[ ]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[ ]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Figure 17.5 shows how these operations are implemented using a class with arrays as data members.

Program

```
#include <iostream>
using namespace std;
const m=50;
class ITEMS
{
     int itemCode[m];
     float itemPrice[m];
     int count;
  public:
     void CNT(void){count = 0;}      // initializes count to 0
     void getitem(void);
     void displaySum(void);
     void remove(void);
     void displayItems(void);
```

Contd

```
};
//=========================================================
void ITEMS :: getitem(void)       // assign values to data
                                  // members of item
{
     cout << "Enter item code :";
     cin >> itemCode[count];
     cout << "Enter item cost :";
     cin >> itemPrice[count];
     count++;
}
void ITEMS :: displaySum(void)       // display total value of
                                     // all items
{
     float sum = 0;
for(int i=0; i<count; i++)
        sum = sum + itemPrice[i];
     cout << "\nTotal value :" << sum << "\n";
}
void ITEMS :: remove(void)           // delete a specified item
{
     int a;
     cout << "Enter item code :";
     cin >> a;
     for(int i=0; i<count; i++)
        if(itemCode[i] == a)
           itemPrice[i] = 0;
}
void ITEMS :: displayItems(void)     // displaying items
{
     cout << "\nCode  Price\n";

     for(int i=0; i<count; i++)
     {
        cout <<"\n" << itemCode[i];
        cout <<"   " << itemPrice[i];
     }
     cout << "\n";
}
//=========================================================
int main()
{
     ITEMS order;
     order.CNT();
     int x;
     do       // do....while loop
```

```
      {
         cout << "\nYou can do the following;"
              << "Enter appropriate number \n";
         cout << "\n1 : Add an item ";
         cout << "\n2 : Display total value";
         cout << "\n3 : Delete an item";
         cout << "\n4 : Display all items";
         cout << "\n5 : Quit";
         cout << "\n\nWhat is your option?";
         cin >> x;
         switch(x)
         {
            case 1 : order.getitem(); break;
            case 2 : order.displaySum(); break;
            case 3 : order.remove(); break;
            case 4 : order.displayItems(); break;
            case 5 : break;
            default : cout << "Error in input; try again\n";
         }
      }  while(x != 5);           // do...while ends
      return 0;
 }
```

Output

```
You can do the following; Enter appropriate number
   1 : Add an item
   2 : Display total value
   3 : Delete an item
   4 : Display all items
   5 : Quit

   What is your option?1
   Enter item code :111
   Enter item cost :100

   You can do the following; Enter appropriate number
   1 : Add an item
   2 : Display total value
   3 : Delete an item
   4 : Display all items
   5 : Quit

   What is your option?1
   Enter item code :222
   Enter item cost :200
   You can do the following; Enter appropriate number
   1 : Add an item
   2 : Display total value
```

Contd.

```
  3 : Delete an item
  4 : Display all items
  5 : Quit

  What is your option?1
  Enter item code :333
  Enter item cost :300

  You can do the following; Enter appropriate number
  1 : Add an item
  2 : Display total value
  3 : Delete an item
  4 : Display all items
  5 : Quit

  What is your option?2
  Total value :600

  You can do the following; Enter appropriate number
  1 : Add an item
  2 : Display total value
  3 : Delete an item
  4 : Display all items
  5 : Quit

  What is your option?3
  Enter item code :222

  You can do the following; Enter appropriate number
  1 : Add an item
  2 : Display total value
  3 : Delete an item
  4 : Display all items
  5 : Quit

  What is your option?4
  Code      Price
  111       100
  222       0
  333       300

  You can do the following; Enter appropriate number
  1 : Add an item
  2 : Display total value
  3 : Delete an item
  4 : Display all items
  5 : Quit
What is your option?5
```

Fig. 17.5 *Program to process shopping list*

Note *The program uses two arrays, namely* ***itemCode[ ]*** *to hold the code number of items and* ***itemPrice[ ]*** *to hold the prices. A third data member* ***count*** *is used to keep a record of items in the list. The program uses a total of four functions to implement the operations to be performed on the list. The statement*

```
const int m = 50;
```

defines the size of the array members.

The first function **CNT**( ) simply sets the variable **count** to zero. The second function **getitem**() gets the item code and the item price interactively and assigns them to the array members **itemCode[count]** and **itemPrice[count]**. Note that inside this function **count** is incremented after the assignment operation is over. The function **displaySum**( ) first evaluates the total value of the order and then prints the value. The fourth function **remove**() deletes a given item from the list. It uses the item code to locate it in the list and sets the price to zero indicating that the item is not 'active' in the list. Lastly, the function **displayItems**( ) displays all the items in the list.

The program implements all the tasks using a menu-based user interface.

17.10 Memory Allocation for Objects

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in Fig. 17.6.

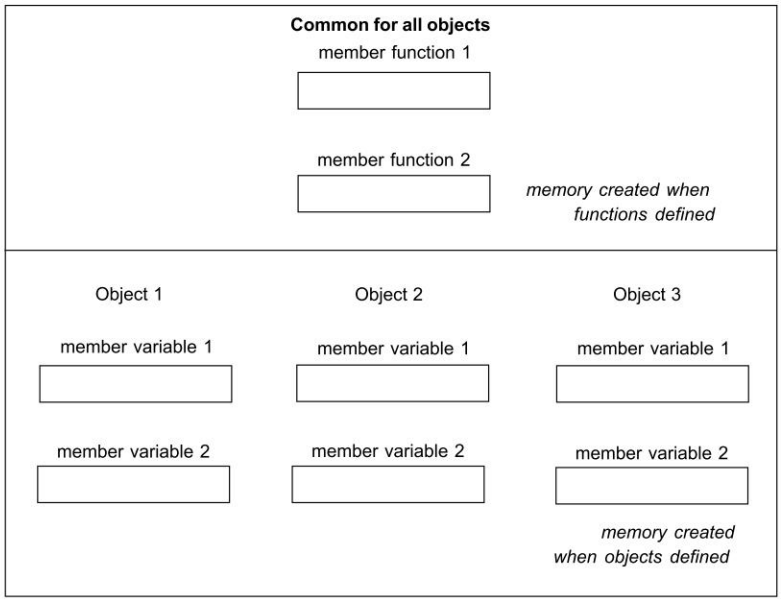


Fig. 17.6 *Object of memory*

17.11 Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. The program in Fig. 17.7 illustrates the use of a static data member.

Program

```
#include <iostream>
using namespace std;
class item
{
      static int count;
      int number;
  public:
      void getdata(int a)
      {
         number = a;
         count ++;
      }
      void getcount(void)
      {
         cout << "count: ";
         cout << count  << "\n";
      }
};
int item :: count;
int main()
{
      item a, b, c;          // count is initialized to zero
      a.getcount();          // display count
      b.getcount();
      c.getcount();
      a.getdata(100);        // getting data into object a
      b.getdata(200);        // getting data into object b
      c.getdata(300);        // getting data into object c
      cout << "After reading data" << "\n";
      a.getcount();             // display count
```

```
        b.getcount();
        c.getcount();
        return 0;
}
Output
  count: 0
  count: 0
  count: 0
  After reading data
  count: 3
  count: 3
  count: 3
```

Fig. 17.7 *Program illustrating static class member*

Note *Notice the following statement in the program:*

```
int item :: count;    // definition of static data member
```

Note that the type and scope of each **static** member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables*.

The **static** variable **count** is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed. Figure 17.8 shows how a static variable is used by the objects.

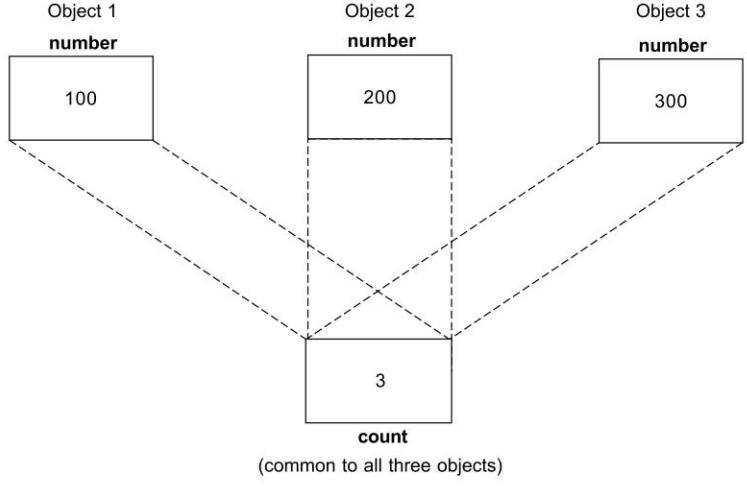


Fig. 17.8 *Sharing of a static data member*

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item :: count = 10;
```

17.12 Static Member Functions

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name ::  function-name;
```

Figure 17.9 illustrates the implementation of these characteristics. The **static** function **showcount()** displays the number of objects created till that moment. A count of number of objects created is maintained by the **static** variable count.

The function **showcode()** displays the code number of each object.

Program

```
#include <iostream>
using namespace std;
class test
{
     int code;
     static int count;         // static member variable
public:
     void setcode(void)
     {
     code = ++count;
     }
     void showcode(void)
     {
     cout << "object number: " << code << "\n";
     }
     static void showcount(void)    // static member function
     {
     cout << "count: " << count << "\n";
     }
};
int test :: count;
int main()
{
     test t1, t2;
```

Contd

```
    t1.setcode();
    t2.setcode();

    test :: showcount();    // accessing static function

    test t3;
    t3.setcode();

    test :: showcount();

    t1.showcode();
    t2.showcode();
    t3.showcode();

    return 0;
}
Output
    count: 2
    count: 3
    object number: 1
    object number: 2
    object number: 3
```

Fig. 17.9 *Program demonstrating static member function*

Note *Note that the statement*
`code = ++count;`
is executed whenever ***setcode( )*** *function is invoked and the current value of count is assigned to* ***code.*** *Since each object has its own copy of* ***code****, the value contained in* ***code*** *represents a unique number of its object.*

Remember, the following function definition will not work:

```
static void showcount()
   {
         cout << code;   // code is not static
   }
```

17.13 Arrays of Objects

We know that an array can be of any data type including **struct**. Similarly, we can also have arrays of variables that are of the type **class**. Such variables are called *arrays of objects*. Consider the following class definition:

```
class employee
{
      char name[30];
      float age;
public:
      void getdata(void);
      void putdata(void);
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];      // array of manager
employee foreman[15];     // array of foreman
employee worker[75];      // array of worker
```

The array **manager** contains three objects(managers), namely, **manager[0], manager[1]** and **manager[2]**, of type **employee** class. Similarly, the **foreman** array contains 15 objects (foremen) and the **worker** array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array-accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

will display the data of the ith element of the array **manager**. That is, this statement requests the object **manager[i]** to invoke the member function **putdata()**.

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array manager is represented in Fig. 17.10. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

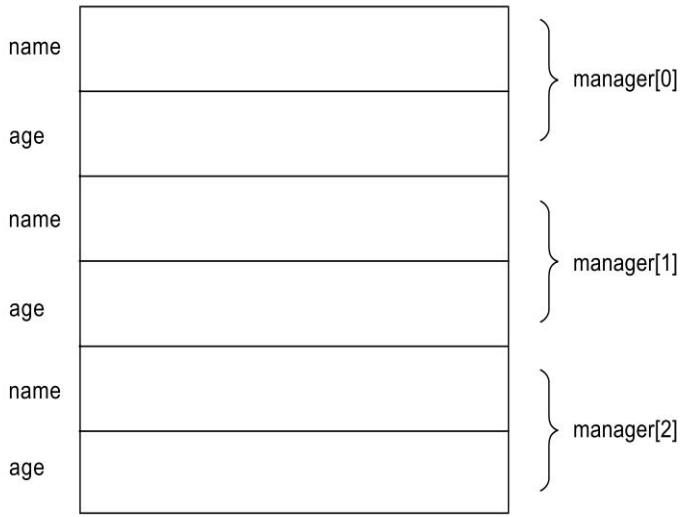


Fig. 17.10 *Storage of data items of an object array*

Figure 17.11 illustrates the use of object arrays.

Program

```
#include <iostream>
using namespace std;
class employee
{
```

Cont'd

```
        char  name[30];    // string as class member
        float age;
    public:
        void getdata(void);
        void putdata(void);
};
void employee :: getdata(void)
{
      cout << "Enter name: ";
      cin  >> name;
      cout << "Enter age:  ";
      cin  >> age;
}
void employee :: putdata(void)
{
      cout << "Name: " << name << "\n";
      cout << "Age:  " << age  << "\n";
}
const int size=3;
int main()
{
      employee manager[size];
      for(int i=0; i<size; i++)
      {
         cout << "\nDetails of manager" << i+1 << "\n";
         manager[i].getdata();
      }
      cout <<  "\n";
      for(i=0; i<size; i++)
      {
         cout << "\nManager" << i+1 << "\n";
         manager[i].putdata();
      }
      return 0;
 }
```

This being an interactive program, the input data and the program output are shown below:

Interactive input

```
    Details of manager1
    Enter name: xxx
    Enter age:  45

    Details of manager2
    Enter name: yyy
    Enter age:  37
    Details of manager3
```

Contd.

```
     Enter name: zzz
     Enter age:  50
Output
     Manager1
     Name: xxx
     Age:  45

     Manager2
     Name: yyy
     Age:  37

     Manager3
     Name: zzz
     Age:  50
```

Fig. 17.11 *Program illustrating use of arrays of objects*

17.14 Objects as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Figure 17.12 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

```
Program
  #include <iostream>
  using namespace std;
  class time
  {
        int  hours;
        int  minutes;
    public:
        void gettime(int h, int m)
        {  hours = h; minutes = m; }
```

Contd.

```
        void puttime(void)
        {
           cout << hours << " hours and ";
           cout << minutes << " minutes " << "\n";
        }
        void sum(time, time);  // declaration with objects as arguments
  };
  void time :: sum(time t1, time t2)       // t1, t2 are objects
  {
        minutes = t1.minutes + t2.minutes;
        hours = minutes/60;
        minutes = minutes%60;
        hours = hours + t1.hours + t2.hours;
  }
int main()
  {
        time T1, T2, T3;

        T1.gettime(2,45);     // get T1
        T2.gettime(3,30);     // get T2

        T3.sum(T1,T2);        // T3=T1+T2
        cout << "T1 = "; T1.puttime();       // display T1
        cout << "T2 = "; T2.puttime();       // display T2
        cout << "T3 = "; T3.puttime();       // display T3
        return 0;
  }
Output
  T1 = 2 hours and 45 minutes
  T2 = 3 hours and 30 minutes
  T3 = 6 hours and 15 minutes
```

Fig. 17.12 *Program illustrating use of objects as arguments*

Note *Since the member function* ***sum( )*** *is invoked by the object* ***T3****, with the objects* ***T1*** *and* ***T2*** *as arguments, it can directly access the hours and minutes variables of* ***T3****. But, the members of* ***T1*** *and* ***T2*** *can be accessed only by using the dot operator (like* ***T1.hours*** *and* ***T1.minutes****). Therefore, inside the function sum(), the variables* ***hours*** *and* ***minute srefer*** *to* ***T3****,* ***T1.hours*** *and* ***T1.minutes*** *refer to* ***T1****, and* ***T2.hours*** *and* ***T2.minutes*** *refer to* ***T2****.*

Figure 17.13 illustrates how the members are accessed inside the function **sum()**.

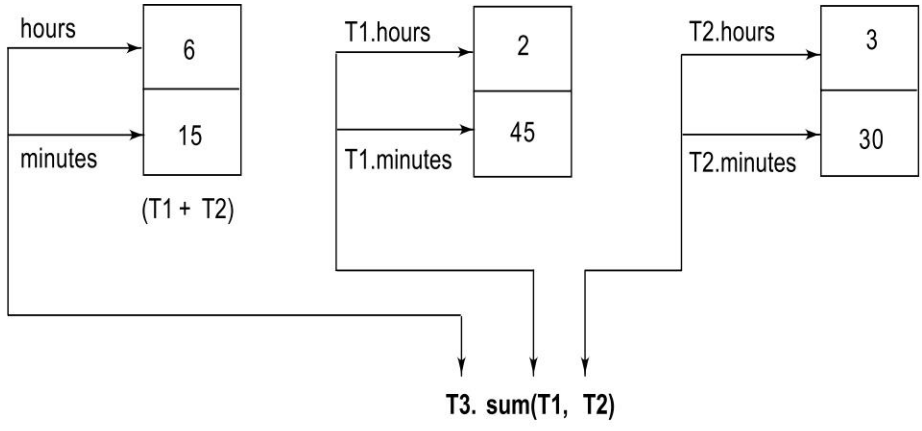


Fig. 17.13 *Accessing members of objects within a called function*

An object can also be passed as an argument to a non-member function. However, such functions can have access to the **public member** functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

17.15 Friendly Functions

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, **manager** and **scientist**, have been defined. We would like to use a function **income_tax()** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
class ABC
{
        .....
        .....
   public:
        .....
        .....
    friend void xyz(void);   // declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope operator ::. The functions that are declared with the keyword friend are

known as friend functions. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

The friend functions are often used in operator overloading which will be discussed later.

Figure 17.14 illustrates the use of a friend function.

Program

```
#include <iostream>
using namespace std;
class sample
{
   int a;
   int b;
  public:
   void setvalue() {a=25; b=40; }
   friend float mean(sample s);
};
float mean(sample s)
{
   return float(s.a + s.b)/2.0;
}
int main()
{
   sample X;    // object X
   X.setvalue();
   cout << "Mean value = " << mean(X) << "\n";

   return 0;
}
```

Output

```
Mean value = 32.5
```

Fig. 17.14 *Program illustrating use of friend function*

Note *The friend function accesses the class variables a and b by using the dot operator and the object passed to it. The function call* ***mean(X)*** *passes the object* ***X*** *by value to the friend function.*

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
      .....
      .....
      int fun1();       // member function of X
      .....
};
class Y
{
       .....
       .....
       friend int X :: fun1();     // fun1() of X
                                   // is friend of Y
       .....
};
```

The function **fun1()** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
      .....
      friend class X; // all member functions of X are
                      // friends to Z
};
```

Figure 17.5 demonstrates (how friend functions work as a bridge between the classes).

Program

```
#include <iostream>
using namespace std;
class ABC;      // Forward declaration
//-----------------------------------------------------//
class XYZ
{
      int x;
  public:
      void setvalue(int i) {x = i;}
      friend void max(XYZ, ABC);
};
//-----------------------------------------------------//
class ABC
{
```

Contd.

```
            int a;
        public:
            void setvalue(int i) {a = i;}
            friend void max(XYZ, ABC);
    };
    //-------------------------------------------------------//
    void max(XYZ m, ABC n)      // Definition of friend
    {
            if(m.x >= n.a)
                    cout << m.x;
            else
                    cout << n.a;
    }
    //-------------------------------------------------------//
    int main()
    {
            ABC abc;
            abc.setvalue(10);
            XYZ xyz;
            xyz.setvalue(20);
            max(xyz, abc);
            return 0;
    }
Output
 20
```

Fig.17.15 *Program demonstrating a friendly function to two classes*

Note *The function max() has arguments from both **XYZ** and **ABC**. When the function max() is declared as a friend in XYZ for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as*

```
class ABC;
```

This is known as 'forward' declaration.

As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private members of a class. Remember, altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

Figure 17.16 shows (how to use a common friend function to exchange the private values of two classes.) The function is called by reference.

Program

```
#include <iostream>
using namespace std;
class class_2;
class class_1
   {
   int value1;
  public:
          void indata(int a) {value1 = a;}
          void display(void) {cout << value1 << "\n";}
          friend void exchange(class_1 &, class_2 &);
};
class class_2
{
      int value2;
  public:
          void indata(int a) {value2 = a;}
          void display(void) {cout << value2 << "\n";}
          friend void exchange(class_1 &, class_2 &);
};
void exchange(class_1 & x, class_2 & y)
{
          int temp = x.value1;
          x.value1 = y.value2;
          y.value2 = temp;
}
int main()
{
   class_1 C1;
   class_2 C2;
   C1.indata(100);
   C2.indata(200);

   cout << "Values before exchange" << "\n";
   C1.display();
   C2.display();
   exchange(C1, C2);       // swapping

   cout << "Values after exchange " << "\n";
   C1.display();
   C2.display();
   return 0;
}
```

The objects x and y are aliases of **C1** and **C2** respectively. The statements

```
int temp = x.value1
```

Cont'd

```
      x.value1 = y.value2;
      y.value2 = temp;
```

directly modify the values of **value1** and **value2** declared in **class_1** and **class_2**.

Output

```
Values before exchange
100
200
Values after exchange
200
100
```

Fig. 17.16 *Program to swap private data of classes*

17.16 Returning Objects

A function cannot only receive objects as arguments but also can return them. The example in Fig. 17.17 illustrates how an object can be created (within a function) and returned to another function

Program

```
#include <iostream>
using namespace std;
class complex         // x + iy form
{
      float x;               // real part
      float y;               // imaginary part
  public:
      void input(float real, float imag)
      { x = real; y = imag; }
      friend complex sum(complex, complex);
      void show(complex);
};
complex sum(complex c1, complex c2)
{
      complex c3;            // objects c3 is created
      c3.x = c1.x + c2.x;
      c3.y = c1.y + c2.y;
      return(c3);            // returns object c3
}
void complex :: show(complex c)
{
      cout << c.x << " + j" << c.y << "\n";
}
int main()
{
```

Contd

```
            complex A, B, C;

            A.input(3.1, 5.65);
            B.input(2.75, 1.2);

         C = sum(A, B);      // C = A + B

         cout << "A = "; A.show(A);
         cout << "B = "; B.show(B);
         cout << "C = "; C.show(C);
         return 0;
      }
Output
      A = 3.1 + j5.65
      B = 2.75 + j1.2
      C = 5.85 + j6.85
```

Fig. 17.17

The program adds two complex numbers **A** and **B** to produce a third complex number **C** and display all the three numbers.

17.17 const Member Functions

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows:

```
void mul(int, int) const;
double get_balance() const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

17.18 Pointers to Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name. A class member pointer can be declared using the operator ::* with the class name. For example, given the class

```
class A
{
  private:
       int m;
  public:
       void show();
};
```

We can define a pointer to the member m as follows:

```
int A::* ip = &A :: m;
```

The **ip** pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase **A::*** means "pointer-to-member of **A** class". The phrase **&A::m** means the "address of the m member of A class".

Remember, the following statement is not valid:

```
int *ip = &m;   // won't work
```

This is because **m** is not simply an **int** type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function. We can access **m** using the pointer **ip** as follows:

```
cout << a.*ip;  // display
cout << a.m;    // same as above
```

Now, look at the following code:

```
ap = &a;           // ap is pointer to object a
cout << ap -> *ip; // display m
cout << ap -> m; // same as above
```

The *dereferencing operator* ->* is used to access a member when we use pointers to both the object and the member. The *dereferencing* operator.* is used when the object itself is used with the member pointer. Note that ***ip** is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the **main** as shown below :

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of () is higher than that of .* and ->*, so the parentheses are necessary.

Figure 17.18 illustrates the use of dereferencing operators to access the class members.

Program

```
#include <iostream>
using namespace std;
class M
{
      int x;
      int y;
  public:
      void set_xy(int a, int b)
      {
            x = a;
            y = b;
      }
      friend int sum(M m);
};
int sum(M m)
```

Contd

```
{
        int M ::* px = &M :: x;
        int M ::* py = &M :: y;
        M *pm = &m;
        int S = m.*px + pm->*py;
        return S;
}
int main()
{
        M n;
        void (M :: *pf)(int,int) = &M :: set_xy;
        (n.*pf)(10,20);
        cout << "SUM = " << sum(n) << "\n";

        M *op = &n;
        (op->*pf)(30,40);
        cout << "SUM = " << sum(n) << "\n";

        return 0;
   }
```

Output

```
sum = 30
sum = 70
```

Fig.17.18 *Program to illustrate derefencing operators*

17.19 Local Classes

Classes can be defined and used inside a function or a block. Such classes are called local classes. Examples:

```
void test(int a)      // function
{
      .....
      .....
      class student   // local class
      {
            .....
            .....       // class definition
            .....
       };
       .....
       .....
       student s1(a);    // create student object
       .....             // use student object
}
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

Just Remember

- A class is an extension to the structure data type. A class can have both variables and functions as members.
- By default, members of the class are private whereas that of structure are public.
- Only the member functions can have access to the private data members and private functions. However the public members can be accessed from outside the class.
- In C++, the class variables are called objects. With objects we can access the public members of a class using a dot operator.
- We can define the member functions inside or outside the class. The difference between a member function and a normal function is that a member function uses a membership 'identity' label in the header to indicate the class to which it belongs.
- The memory space for the objects is allocated when they are declared. Space for member variables is allocated separately for each object, but no separate space is allocated for member functions.
- A data member of a class can be declared as a **static** and is normally used to maintain values common to the entire class.
- The static member variables must be defined outside the class.
- A static member function can have access to the static members declared in the same class and can be called using the class name.
- C++ allows us to have arrays of objects.
- We may use objects as function arguments.
- A function declared as a **friend** is not in the scope of the class to which it has been declared as friend. It has full access to the private members of the class.
- A function can also return an object.
- If a member function does not alter any data in the class, then we may declare it as a **const** member function. The keyword **const** is appended to the function prototype.
- It is also possible to define and use a class inside a function. Such a class is called a local class.

Review Questions

17.1 How do structures in C and C++ differ?
17.2 What is a class? How does it accomplish data hiding?
17.3 How does a C++ structure differ from a C++ class?
17.4 What are objects? How are they created?
17.5 How is a member function of a class defined?

17.6 Can we use the same function name for a member function of a class and an outside function in the same program file? If yes, how are they distinguished? If no, give reasons.

17.7 Describe the mechanism of accessing data members and member functions in the following cases:

(a) Inside the main program.
(b) Inside a member function of the same class.
(c) Inside a member function of another class.

17.8 When do we declare a member of a class static?

17.9 What is a friend function? What are the merits and demerits of using friend functions?

17.10 State whether the following statements are TRUE or FALSE.

(a) Data items in a class must always be private.
(b) A function designed as private is accessible only to member functions of that class.
(c) A function designed as public can be accessed like any other ordinary functions.
(d) Member functions defined inside a class specifier become inline functions by default.
(e) Classes can bring together all aspects of an entity in one place.
(f) Class members are public by default.
(g) Friend functions have access to only public members of a class.
(h) An entire class can be made a friend of another class.
(i) Functions cannot return class objects.
(j) Data members can be initialized inside class specifier.

Programming Exercises

17.1 Define a class to represent a bank account. Include the following members:

Data members

1. Name of the depositor
2. Account number
3. Type of account
4. Balance amount in the account

Member functions

1. To assign initial values
2. To deposit an amount
3. To withdraw an amount after checking the balance
4. To display name and balance

Write a main program to test the program.

17.2 Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks:

(a) To create the vector
(b) To modify the value of a given element
(c) To multiply by a scalar value
(d) To display the vector in the form (10, 20, 30, ...)

Write a program to test your class.

17.3 Modify the class and the program of Exercise 17.11 for handling 10 customers.

17.4 Modify the class and program of Exercise 17.12 such that the program would be able to add two vectors and display the resultant vector. (Note that we can pass objects as function arguments.)

17.5 Create two classes DM and DB which store the value of distances. DM stores distances in metres and centimetres and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB.
Use a friend function to carry out the addition operation. The object that stores the results may be a DM object or DB object, depending on the units in which the results are required.
The display should be in the format of feet and inches or metres and centimetres depending on the object on display.

Debugging Exercises

17.1 Identify the error in the following program:

```
#include <iostream.h>
struct Room
{
       int width;
       int length;
          void setValue(int w, int l)
          {
             width = w;
             length = l;
          }
   };
   void main()
   {
          Room objRoom;
          objRoom.setValue(12, 1,4);
   }
```

17.2 Identify the error in the following program:

```
#include <iostream.h>
class Room
{
       int width, height;
       void setValue(int w, int h)
       {
          width = w;
          height = h;
       }
};
void main()
{
       Room objRoom;
```

```
        objRoom.width = 12;
}
```

17.3 Identify the error in the following program:

```
#include <iostream.h>
class Item
{
private:
        static int count;
public:
        Item()
        {
           count++;
        }
        int getCount()
        {
           return count;
        }
        int* getCountAddress()
        {
             return count;
           }
        };
        int Item::count = 0;

        void main()
        {
             Item objItem1;
             Item objItem2;

             cout << objItem1.getCount() << ' ';
             cout << objItem2.getCount() << ' ';

             cout << objItem1.getCountAddress() << ' ';
             cout << objItem2.getCountAddress() << ' ';
    }
```

17.4 Identify the error in the following program:

```
#include <iostream.h>
class staticFunction
{
        static int count;
public:
        static void setCount()
        {
           count++;
        }
```

```
        void displayCount()
        {
            cout << count;
        }
};
int staticFunction::count = 10;
void main()
{
        staticFunction obj1;
        obj1.setCount(5);
        staticFunction::setCount();
        obj1.displayCount();
}
```

17.5 Identify the error in the following program:

```
#include <iostream.h>
class Length
{
        int feet;
        float inches;
public:
        Length()
        {
            feet = 5;
            inches = 6.0;
        }
        Length(int f, float in)
        {
            feet = f;
            inches=in;
        }
        Length addLength(Length l)
        {
            l.inches += this->inches;
            l.feet += this->feet;
if(l.inches>12)
{
   l.inches-=12;
   l.feet++;
}
   return l;
        }
        int getFeet()
        {
            return feet;
        }
        float getInches()
        {
```

```
            return inches;
        }
};
void main()
{
        Length objLength1;
        Length objLength1(5, 6.5);
        objLength1 = objLength1.addLength(objLength2);
        cout << objLength1.getFeet() << ' ';
        cout << objLength1.getInches() << ' ';
}
```

17.6 Identify the error in the following program:

```
#include <iostream.h>
class Room;
void Area()
{
        int width, height;
        class Room
        {
            int width, height;
            public:
            void setValue(int w, int h)
            {
                width = w;
                height = h;
            }
            void displayValues()
            {
                cout << (float)width << ' ' <<  (float)height;
            }
        };
        Room objRoom1;
        objRoom1.setValue(12, 8);
        objRoom1.displayValues();
}

void main()
{
        Area();
        Room objRoom2;
}
```



18 Constructors and Destructors

18.1 Introduction

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata()** and **setvalue()** to provide initial values to the private member variables. For example, the following statement

```
A.input();
```

invokes the member function **input()**, which assigns the initial values to the data items of object **A**. Similarly, the statement

```
x.getdata(100,299.95);
```

passes the initial values as arguments to the function **getdata()**, where these values are assigned to the private variables of object **x**. All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class**, that behave very similar to the built-in types. This means that we should be able to initialize a **class** type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some

more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

18.2 Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
       int m, n;
   public:
       integer(void);           // constructor declared
       .....
       .....
};
integer :: integer(void)     // constructor defined
{
      m = 0; n = 0;
0}
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1;      // object int1 created
```

not only creates the object int1 of type integer but also initializes its data members m and n to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the *default constructor*. The default constructor for **class A is A::A()**. If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

invokes the default constructor of the compiler to create the object **a**.

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Note *Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.*

18.3 Parameterized Constructors

The constructor integer(), defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors.
The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
        int m, n;
   public:
        integer(int x, int y);   // parameterized constructor
        .....
        .....
};
integer :: integer(int x, int y)
{
       m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);         // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Note *Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. The program in Fig. 18.1 demonstrates the passing of arguments to the constructor functions.*

Program

```
#include <iostream>

using namespace std;

class integer
{
       int m, n;
  public:
       integer(int, int);   // constructor declared

       void display(void)
       {
        cout << " m = " << m  << "\n";
        cout << " n = " << n  << "\n";
       }
};

integer :: integer(int x, int y)   // constructor defined
{
      m = x;  n = y;
}

int main()
{
     integer int1(0,100);          // constructor called implicitly

     integer int2 = integer(25, 75); // constructor called explicitly

     cout << "\nOBJECT1" << "\n";
     int1.display();

     cout << "\nOBJECT2" << "\n";
     int2.display();
```

Contd.

```
        return 0;
    }
Output
    OBJECT1
    m = 0
    n = 100
    OBJECT2
    m = 25
    n = 75
```

Fig.18.1 *Program showing a class with constructors*

The constructor functions can also be defined as **inline** functions. Example:

```
class integer
{
     int m, n;
  public:
     integer(int x, int y)   // Inline constructor
     {
        m = x; y = n;
     }
      .....
      .....
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
       .....
       .....
       public:
       A(A);
};
```

is illegal.

However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

```
Class A
{
      .....
         .....
  public:
        A(A&);
};
```

is valid. In such cases, the constructor is called the *copy constructor*.

18.4 Multiple Constructors in a Class

So far we have used two kinds of constructors. They are:

```
integer();                  // No arguments
integer(int, int);     // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from **main()**. C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```
class integer
  {
        int m, n;
     public:
        integer(){m=0; n=0;}     // constructor 1
        integer(int a, int b)
        {m = a; n = b;}          // constructor 2
        integer(integer & i)
        {m = i.m;  n = i.n;}        // constructor 3
  };
```

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second receives two **integer** arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both **m** and **n** of I1 to zero. The statement

```
integer I2(20,40);
```

would call the second constructor which will initialize the data members **m** and **n** of I2 to 20 and 40 respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of **I2** into **I3**. In other words, it sets the value of every data element of **I3** to the value of the corresponding data element of **I2**. As mentioned earlier, such a constructor is called the *copy constructor*. We learned in Chapter 16 that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.
Figure 18.2 shows the use of overloaded constructors.

Program

```
#include <iostream>
using namespace std;
class complex
{
      float x, y;
```

```
  public:
       complex(){ }                    // constructor no arg
       complex(float a) {x = y = a;}   // constructor-one arg
       complex(float real, float imag)      // constructor-two args
       {x = real; y = imag;}

       friend complex sum(complex, complex);
       friend void show(complex);
};
complex sum(complex c1, complex c2)   // friend
{
       complex c3;
       c3.x = c1.x + c2.x;
       c3.y = c1.y + c2.y;
       return(c3);
}
void show(complex c)     // friend
{
       cout << c.x << " + j" << c.y << "\n";
}
int main()
{
       complex A(2.7, 3.5);         // define & initialize
       complex B(1.6);              // define & initialize
       complex C;                   // define

       C = sum(A, B);            // sum() is a friend
       cout << "A = "; show(A); // show() is also friend
       cout << "B = "; show(B);
       cout << "C = "; show(C);
// Another way to give initial values (second method)
       complex P,Q,R;            // define P, Q and R
          P = complex(2.5,3.9);     // initialize P
       Q = complex(1.6,2.5);     // initialize Q
          R = sum(P,Q);

        cout << "\n";
        cout << "P = "; show(P);
        cout << "Q = "; show(Q);
        cout << "R = "; show(R);

        return 0;
}
```

Output

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1
```

Contd.

```
P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j18.4
```

Fig. 18.2 *Program showing overloaded constructors*

Note *There are three constructors in the class complex. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.*

Let us look at the first constructor again.

```
complex(){ }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now?. As pointed out earlier, C++ compiler has an *implicit constructor* which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

18.5 Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument **imag** is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real** variable and 0.0 to **imag** (by default). However, the statement

```
complex C(2.0,3.0);
```

assigns 2.0 to **real** and 3.0 to **imag.** The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A()** and the default argument constructor **A::A(int = 0)**. The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' **A::A() or A::A(int = 0).**

18.6 Dynamic Initialization of Objects

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Figure 18.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

Program

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
      long int P_amount;       // Principal amount
      int      Years;          // Period of investment
      float    Rate;           // Interest rate
      float    R_value;        // Return value of amount
  public:
      Fixed_deposit(){ }
      Fixed_deposit(long int p, int y, float r=0.12);
      Fixed_deposit(long int p, int y, int r);
      void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
      P_amount = p;
      Years = y;
      Rate = r;
      R_value = P_amount;
      for(int i = 1; i <= y; i++)
         R_value = R_value * (1.0 + r);
}

Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
      P_amount = p;
      Years = y;
      Rate = r;
      R_value = P_amount;

      for(int i=1; i<=y; i++)
         R_value = R_value*(1.0+float(r)/100);
}
```

```
void Fixed_deposit :: display(void)
{
      cout << "\n"
            << "Principal Amount = " << P_amount << "\n"
            << "Return Value     = " << R_value  << "\n";
}
int main()
{
      Fixed_deposit FD1, FD2, FD3;  // deposits created

      long int p;                   // principal amount

   int     y;                   // investment period, years
   float   r;                   // interest rate, decimal form
   int     R;                   // interest rate, percent form
   cout << "Enter amount,period,interest rate(in percent)"<<"\n";
   cin >> p >> y >> R;
   FD1 = Fixed_deposit(p,y,R);

   cout << "Enter amount,period,interest rate(decimal form)" << "\n";
   cin >> p >> y >> r;
   FD2 = Fixed_deposit(p,y,r);

   cout << "Enter amount and period" << "\n";
   cin >> p >> y;
   FD3 = Fixed_deposit(p,y);

   cout << "\nDeposit 1";
   FD1.display();

   cout << "\nDeposit 2";
   FD2.display();

   cout << "\nDeposit 3";
   FD3.display();

   return 0;
}
```

Output

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest rate(in decimal form)
10000 3 0.18
Enter amount and period
10000 3
```

Contd.

```
Deposit 1
Principal Amount = 10000
Return Value     = 16430.3

Deposit 2
Principal Amount = 10000
Return Value     = 16430.3

Deposit 3
Principal Amount = 10000
Return Value      = 14049.3
```

Fig. 18.3 *Program to demonstrate dynamic initialization of constructors*

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

Note *Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for* ***r****.*

18.7 Copy Constructor

We briefly mentioned about the copy constructor in Sec. 18.3. We used the copy constructor

```
integer(integer &i);
```

in Sec. 18.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
integer I2 = I1;
```

The process of initializing through a copy constructor is known as copy initialization. Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1** and **I2** are objects, this statement is legal and simply assigns the values of **I1** to **I2**, member-by-member. This is the task of the overloaded assignment operator(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Fig 18.4.

Program

```
#include <iostream>
using namespace std;

class code
{
   int id;
  public:
   code(){ }                    // constructor
   code(int a) { id = a;}          // constructor again
   code(code & x)               // copy constructor

   {
      id = x.id;                   // copy in the value
   }
   void display(void)
   {
      cout << id;
   }
};
int main()
{
   code A(100);    // object A is created and initialized
   code B(A);      // copy constructor called
   code C = A;     // copy constructor called again

   code D; // D is created, not initialized
   D = A;             // copy constructor not called
   cout << "\n id of A: "; A.display();
   cout << "\n id of B: "; B.display();
   cout << "\n id of C: "; C.display();
   cout << "\n id of D: "; D.display();
   return 0;
}
```

Output

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

Fig. 18.4 *Program demonstrating copy constructor*

Note *A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.*

When no copy constructor is defined, the compiler supplies its own copy constructor.

18.8 Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. Figure 18.5 shows the use of new, in constructors that are used to construct strings in objects.

Program

```
#include <iostream>
#include <string>

using namespace std;

class String
{
      char *name;
      int  length;
  public:
      String()                        // constructor-1
      {
         length = 0;
         name = new char[length + 1];
      }

      String(char *s)              // constructor-2
      {
         length = strlen(s);
         name = new char[length + 1];    // one additional
                                         // character for \0
         strcpy(name, s);
         }

         void display(void)
         {cout << name << "\n";}
         void join(String &a, String &b);
      };

      void String :: join(String &a, String &b)
      {
         length = a.length + b.length;
         delete name;
         name = new char[length+1];           // dynamic allocation

         strcpy(name, a.name);
         strcat(name, b.name);
```

Cont'd

```
};

int main()
{
   char *first = "Joseph ";
   String name1(first), name2("Louis "),name3("Lagrange"),s1,s2;

   s1.join(name1, name2);
   s2.join(s1, name3);
   name1.display();
   name2.display();
   name3.display();
   s1.display();
   s2.display();

   return 0;
}
```

Output

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

Fig. 18.5 *Program illustrating constructors with* ***new***

Note *This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the* ***length*** *of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '\0'.*

The member function **join( )** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy( )** and **strcat( )**. Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. The **main( )** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

18.9 Constructing Two-dimensional Arrays

We can construct matrix variables using the class type objects. The example in Figure 18.6 illustrates how to construct a matrix of size m x n.

Program

```
#include <iostream>

using namespace std;

class matrix
{
   int **p;    // pointer to matrix
   int d1,d2;   // dimensions
public:
   matrix(int x, int y);
   void get_element(int i, int j, int value)
   {p[i][j]=value;}
   int & put_element(int i, int j)
   {return p[i][j];}
};
matrix :: matrix(int x, int y)
{
   d1 = x;
   d2 = y;
   p = new int *[d1];          // creates an array pointer
   for(int i = 0; i < d1; i++)
      p[i] = new int[d2]; // creates space for each row
}

int main()
{
   int m, n;

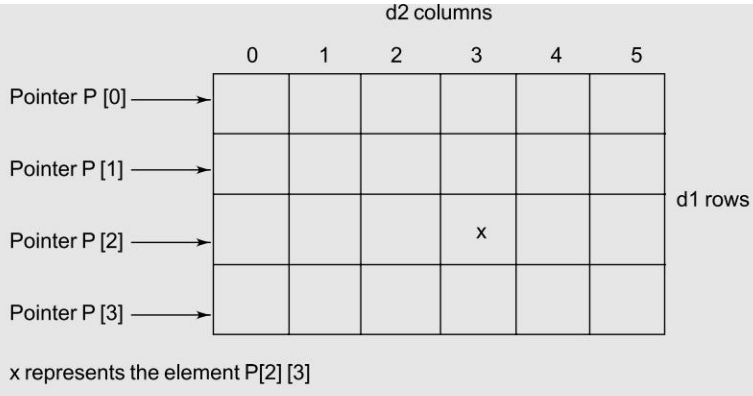
   cout << "Enter size of matrix: ";
   cin  >> m >> n;
   matrix A(m,n); // matrix object A constructed

   cout << "Enter matrix elements row by row \n";
   int i, j, value;

   for(i = 0; i < m; i++)
      for(j = 0; j < n; j++)
      {
         cin >> value;
         A.get_element(i,j,value);
      }
   cout << "\n";
   cout << A.put_element(1,2);

   return 0;
};
```

Contd



Output

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22
17
```

Fig. 18.6 *Program to construct matrix objects*

17 is the value of the element (1,2).

The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i]**.

Thus, space for the elements of a d1 × d2 matrix is allocated from free store as shown above.

18.10 const Objects

We may create and use constant objects using **const** keyword before object declaration. For example, we may create X as a constant object of the class **matrix** as follows:

```
const matrix X(m,n);  // object X is constant
```

Any attempt to modify the values of **m** and **n** will generate compile-time error. Further, a constant object can call only **const** member functions. As we know, a **const** member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever **const** objects try to invoke non-**const** member functions, the compiler generates errors.

18.11 Destructors

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer(){ }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory. For example, the destructor for the **matrix** class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
     for(int i=0; i<d1; i++)
     delete p[i];
     delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

Figure 18.7 illustrates that the destructor has been invoked implicitly by the compiler.

Program

```
#include <iostream>

using namespace std;

int count = 0;

class alpha
{
  public:
   alpha()
   {
      count++;
      cout << "\nNo.of object created " << count;
   }

   ~alpha()
   {
      cout << "\nNo.of object destroyed " << count;
      count--;
   }
};

int main()
{
   cout << "\n\nENTER MAIN\n";

   alpha A1, A2, A3, A4;
```

Cont'd

```
        {
           cout << "\n\nENTER BLOCK1\n";
           alpha A5;
        }

        {
           cout << "\n\nENTER BLOCK2\n";
           alpha  A6;
        }
        cout << "\n\nRE-ENTER MAIN\n";

        return 0;
     }
```

Output

```
     ENTER MAIN

     No.of object created 1
     No.of object created 2
     No.of object created 3
     No.of object created 4

     ENTER BLOCK1

     No.of object created 5
     No.of object destroyed 5

     ENTER BLOCK2

     No.of object created 5
     No.of object destroyed 5

     RE-ENTER MAIN

     No.of object destroyed 4
     No.of object destroyed 3
     No.of object destroyed 2
     No.of object destroyed 1
```

Fig. 18.7 *Program demonstrating implementation of destructors*

Note *As the objects are created and destroyed, they increase and decrease the count. Notice that after the first group of objects is created,* ***A5*** *is created, and then destroyed,* ***A6*** *is created, and then destroyed. Finally, the rest of the objects are also destroyed. When the closing brace of a scope is encountered, the destructors for each object in the scope are called. Note that the objects are destroyed in the reverse order of creation.*

Just Remember

- C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects.
- A constructor has the same name as that of a class.
- Constructors are normally used to initialize variables and to allocate memory.
- Similar to normal functions, constructors may be overloaded.
- When an object is created and initialized at the same time, a copy constructor gets called.
- We may make an object **const** if it does not modify any of its data values.
- C++ also provides another member function called the destructor that destroys the objects when they are no longer required.

Review Questions

18.1 What is a constructor? Is it mandatory to use constructors in a class?

18.2 How do we invoke a constructor function?

18.3 List some of the special properties of the constructor functions.

18.4 What is a parameterized constructor?

18.5 Can we have more than one constructors in a class? If yes, explain the need for such a situation.

18.6 What do you mean by dynamic initialization of objects? Why do we need to do this?

18.7 How is dynamic initialization of objects achieved?

18.8 Distinguish between the following two statements:

```
time T2(T1);
time T2 = T1;
```

T1 and T2 are objects of time class.

18.9 Describe the importance of destructors.

18.10 State whether the following statements are TRUE or FALSE.

(a) Constructors, like other member functions, can be declared anywhere in the class.

(b) Constructors do not return any values.

(c) A constructor that accepts no parameter is known as the default constructor.

(d) A class should have at least one constructor.

(e) Destructors never take any argument.

Programming Exercises

18.1 Design constructors for the classes designed in Programming Exercises 17.1 through 17.5 of Chapter 17.

18.2 Define a class String that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string

```
String s1; // string with length 0
```

and also to initialize an object with a string constant at the time of creation like

```
String s2("Well done!");
```

Include a function that adds two strings to make a third string. Note that the statement

```
s2 = s1;
```

will be perfectly reasonable expression to copy one string to another.
Write a complete program to test your class to see that it does the following tasks:

(a) Creates uninitialized string objects.
(b) Creates objects with string constants.
(c) Concatenates two strings properly.
(d) Displays a desired string object.

18.3 A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed.
Design a system using a class called books with suitable member functions and constructors. Use new operator in constructors to allocate memory space required.

18.4 Improve the system design in Exercise 18.3 to incorporate the following features:

(a) The price of the books should be updated as and when required. Use a private member function to implement this.
(b) The stock value of each book should be automatically updated as soon as a transaction is completed.
(c) The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use static data members to keep count of transactions.

18.5 Modify the program of Exercise 18.4 to demonstrate the use of pointers to access the members.

Debugging Exercises

18.1 Identify the error in the following program.

```
        #include <iostream.h>
class Room
{
        int length;
        int width;
```

```
public:
      Room(int l, int w=0):
         width(w),
         length(l)
      {
      }
};
void main()
{
      Room objRoom1;
      Room objRoom2(12, 8);
      }
```

18.2 Identify the error in the following program.

```
      #include <iostream.h>
class Room
{
      int length;
      int width;
public:
      Room()
      {
         length = 0;
         width = 0;
      }
      Room(int value=8)
      {
         length = width = 8;
      }
      void display()
      {
         cout << length << ' ' << width;
      }
};

void main()
{
      Room objRoom1;
      objRoom1.display();
      }
```

18.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
      int width;
      int height;
      static int copyConsCount;
```

```
public:
       void Room()
       {
          width = 12;
          height = 8;
       }

       Room(Room& r)
       {
          width = r.width;
          height = r.height;
          copyConsCount++;
       }

       void dispCopyConsCount()
       {
          cout << copyConsCount;
       }
};

int Room::copyConsCount = 0;

void main()
{
       Room objRoom1;
       Room objRoom2(objRoom1);
       Room objRoom3 = objRoom1;
       Room objRoom4;
       objRoom4 = objRoom3;

       objRoom4.dispCopyConsCount();
}
```

18.4 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
       int width;
       int height;
       static int copyConsCount;
public:
       Room()
       {
          width = 12;
          height = 8;
       }

       Room(Room& r)
```

```
        {
            width = r.width;
            height = r.height;
            copyConsCount++;
        }

        void disCopyConsCount()
        {
            cout << copyConsCount;
        }
};

int Room::copyConsCount = 0;

void main()
{
        Room objRoom1;
        Room objRoom2 (objRoom1);
        Room objRoom3 = objRoom1;
        Room objRoom4;
        objRoom4 = objRoom3;

        objRoom4.dispCopyConsCount();
}
```

Gujarat Technological University

B.E Sem-II Examination June-2010

Subject code: 110003

Subject Name: Computer Programming and Utillization

Date: 17/06/2010 **Time: 02.30 pm – 05.00 pm**

Total Marks: 70

Instructions

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**

Q.1 (a) Answer the following questions as directed: **12**

1. What is infinite loop in C? How can you create it? Give an example of situation where you need to use infinite loop?
2. Give the meaning and output of following C statements.
   a. x = 1, 2, 3;
   b. a = (b = 4, c = 5, b + c);
   c. if (x = 5)
      printf ("x is even\");
      else
      printf (" x is odd\n");
3. Provide two major differences between
   a. Software and Hardware
   b. Flowchart and Algorithm
4. Tell whether following words represent class or object. For those words representing class, give examples of object instances and for those words representing object, name its possible class.
   a. God
   b. India
   c. Ganga

(b) Name the major components of computer and give their function in brief. **02**

Q.1 (a) (1)

Infinite Loop

An infinite loop is the one that never terminates and runs infinite number of times. The test condition of an infinite loop always holds true. The only way of coming out of an infinite loop is through a jump statement.

Creation of Infinite Loop

An infinite loop can be created by using a test condition that never becomes false. The following examples depict the creation of infinite loops:

```
i=1;
while (i>o)
{
 .
 .
 i++;
}
```

```
while (1)
{
 .
 .
}
```

Use of Infinite Loop

An infinite loop may be used in a situation where a set of instructions are required to be executed continuously, for instance, while displaying menu options to the user. The user is always prompted a set of menu options to choose from. Even after a user has selected a particular menu option and performed the associated task, the menu options appear again (through infinite loop) for the user to perform another task. This cycle breaks only when the user chooses the exit option from the menu for which a corresponding jump statement is already included within the statement block of the looping construct. The following code depicts this scenario:

```
/*Creating an interactive interface for performing stack operations*/

while(1)
 {
  clrscr();
  printf("Select a choice from the following:");
  printf("\n[1] Push an element into the stack");
  printf("\n[2] Pop out an element from the stack");
  printf("\n[3] Display the stack elements");
  printf("\n[4] Exit\n");
  printf("\n\tYour choice: ");
  scanf("%d",&choice);

  switch(choice)
  {
   case 1:
```

Contd.

```
        {
          .
          .
         break;
        }

        case 2:
        {
          .
          .
         break;
        }
        case 3:
        {
          .
          .
         break;
        }
        case 4:
        {
         exit(1); //this statement takes the control out of the infinite while
    loop
        break;
        }

        default:
         printf("\nInvalid choice!\n");
         break;
        }
       }
```

Q.1 (a) (2) (a)

x = 1,2,3;

This statement will result in syntax error as only one value can be allocated to a variable at one time.

Q.1 (a) (2) (b)

a=(b=4,c=5,b+c);

The above statement will result in the following:

- Allocation of b to 4
- Allocation of c to 5

- Computation of sum of b and c i.e. 9
- Allocation of a to 9

Q.1 (a) (2) (c)

if(x=5)
printf("x is even\n");
else
printf("x is odd\n");

The test condition in the above if block uses an assignment operator contrary to the expected logical operator ==. Thus, x is allocated the value 5 and the test statement of the if block holds true resulting in display of message "x is even".

Q.1 (a) (3) (a)

Software	Hardware
Software is a computer program i.e. a set of logical instructions that tell how a computer/hardware should function.	Hardware comprises of the physical computer components such as monitor, printer, etc. that function as per the instructions provided by the computer software.
Software acts as an interface between computer users and the hardware.	Hardware works at the lower level and is actually responsible for executing the user initiated actions.

Q.1 (a) (3) (b)

Flowchart	Algorithm
A flowchart is the graphical representation of the flow of control and logic in the solution of a problem.	An algorithm is a sequence of steps written in the form of English phrases that specify the tasks that are performed while solving a problem.
Flowcharts help to understand the flow of program control in a better way.	Algorithms lack visual representation of programming constructs, thus understanding the logic becomes relatively difficult.

Q.1 (a) (4) (a)

God is a class.
Possible objects: Lord Ram, Lord Krishna, Jesus Christ, etc.

Q.1 (a) (4) (b)

India is an object.
Possible class: Country

Q.1 (a) (4) (c)

Ganga is an object.
Possible class: River

Q.1 (b)

Major Components of Computer

Refer Section 1.2, "Basic Anatomy of Computer"

Q.2 **(a)** Writ an algorithm and draw the flowchart to accept N number and count how many of them where odd and also compute sum of all these odd. **07**

(b) Write a C program to display the following triangle for N lines. **07**

```
        1
      A   B
    1   2   3
  A   B   C   D
1   2   3   4   5
```

OR

(b) Write a C program to display the following triangle for N lines. **07**

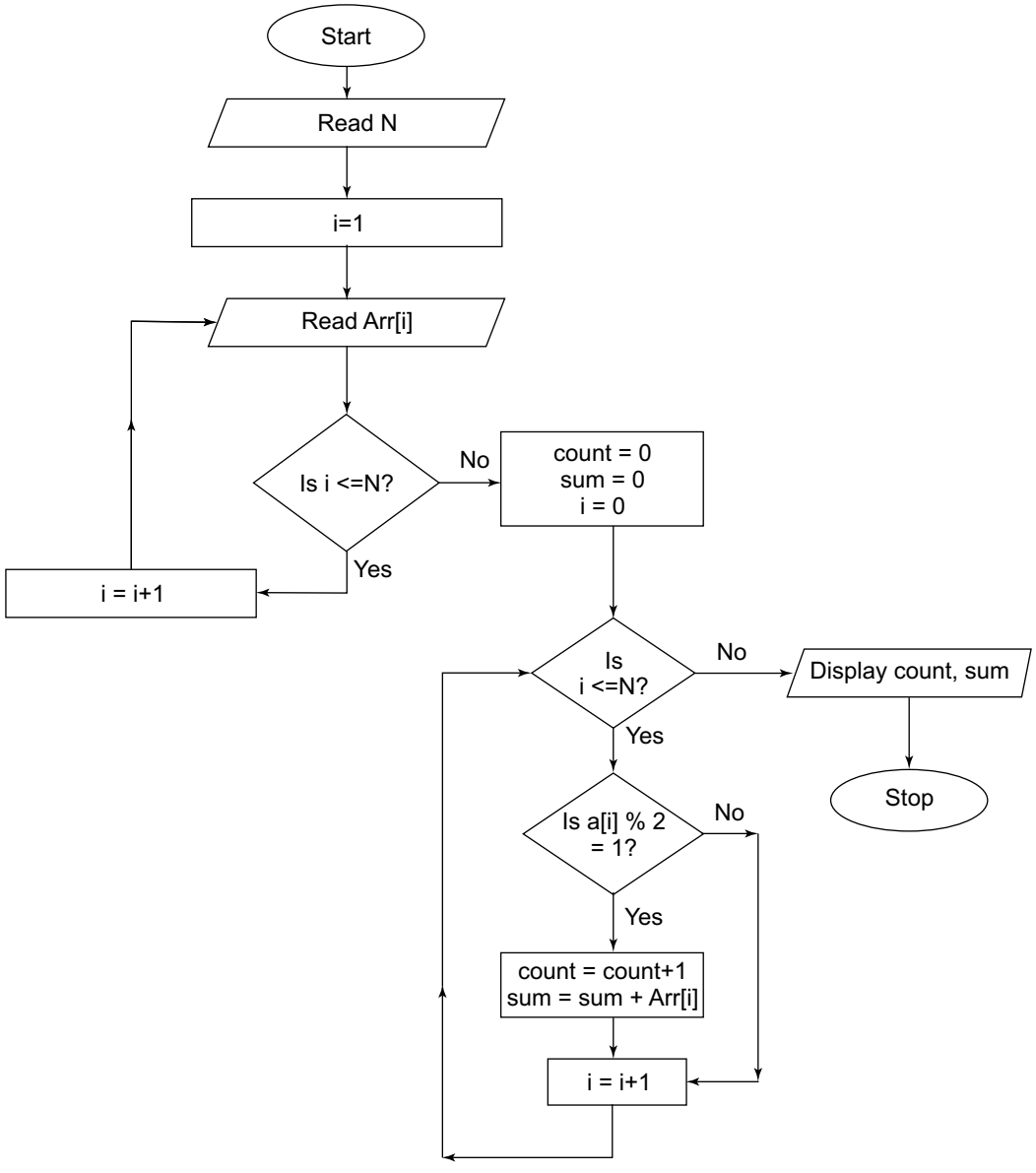
```
        1
      A   B
    2   3   4
  C   D   E   F
5   6   7   8   9
```

Q.2 (a)

Algorithm

```
Step 1 - Start
Step 2 - Accept the value of N
Step 3 - Initialize looping counter i to 1
Step 4 - Repeat Steps 5-6 while i <= N
Step 5 - Read the ith value and store it in Arr[i]
Step 6 - Increment the value of i by 1
Step 7 - Initialize variables count = 0, sum = 0
Step 8 - Initialize looping counter i to 1
Step 9 - Repeat Steps 10-12 while i <= N
Step 10- If the remainder of Arr[i] divided by 2 (Arr[i]%2) is equal to
1then goto Step 11 else goto Step 12
Step 11- count = count + 1; sum = sum + Arr[i]
Step 12- Increment the value of i by 1
Step 13- Display count as the number of odd values entered
Step 14- Display sum as the sum of the odd values
Step 15- Stop
```

Flowchart



Q.2 (b)

Program

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

Contd.

```
void main ()
{
 int n,i,j;
 clrscr();

 printf("Enter the value of n: ");
 scanf("%d",&n);
 printf("\n");
 for(i=1;i<=n;i++)
 {

 if(i%2==1)
 {
  for(j=0;j<=(n-i);j++)
   printf(" ");

  for(j=1;j<=i;j++)
   printf("%d ",j);

  printf("\n");
 }

 else
 {
  for(j=0;j<=(n-i);j++)
   printf(" ");

  for(j=1;j<=i;j++)
   printf("%c ",j-1+65);

  printf("\n");
 }
 }
 getch();
}
```

Output

```
Enter the value of n: 5

    1
   A B
  1 2 3
 A B C D
1 2 3 4 5
```

OR

Q.2 (b)

Program

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

void main ()
{
 int n,i,j,flag1,flag2;
 clrscr();

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

flag1=0;
flag2=0;

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

 if(i%2==1)
 {
  for(j=0;j<=(n-i);j++)
   printf(" ");

  for(j=1+flag1;j<=i+flag1;j++)
   printf("%d ",j);

  printf("\n");
  flag1=i+flag1;
 }

 else
 {
  for(j=0;j<=(n-i);j++)
   printf(" ");

  for(j=1+flag2;j<=i+flag2;j++)
   printf("%c ",j-1+65);

  printf("\n");
  flag2=i+flag2;
 }
    }

    getch();
}
```

Output

```
Enter the value of n: 5

     1
    A B
   2 3 4
  C D E F
 5 6 7 8 9
```

Q.3 **(a)** Explain the following with proper syntax and examples. **08**

1. Structure in C
2. Recursion

(b) Write a C program to reverse a string given from keyboard. Don't use library function to reverse it. **06**

OR

Q.3 **(a)** Explain the following with proper syntax and examples. **08**

1. Pointers in C
2. Call by value Vs. Call by reference

(b) Write a C program to accept array of n integers and find the largest odd number as well as largest even number and display them. **06**

Q.3 (a) (1)

Structure in C

Refer Section 12.2 – 12.5

Q.3 (a) (2)

Recursion

Refer Section "10.16 Recursion"

Q.3 (b)

Program

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

void main()
{
 char str[50],revstr[40];
 int i,len;
 clrscr();
 printf("\nEnter a string: ");
 gets(str);

 len=0;
 i=0;
 while(str[i]!='\0')
 {
  len=len+1;
  i=i+1;
```

Contd.

```
      }

      for(i=0;i<len;i++)
       revstr[len-i-1]=str[i];
      revstr[len]='\0';

      printf("\n\nThe reverse of string %s is %s",str,revstr);

      getch();
     }
```

Output

```
     Enter a string: Taj Mahal


     The reverse of string Taj Mahal is lahaM jaT
```

OR

Q.3 (a) (1)

Pointers in C

Refer Section 11.2 – 11.6

Q.3 (a) (2)

Call by value Vs. Call by reference

Refer Section 11.13

Q.3 (b)

Program

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

     void main()
     {
      int a[50],i,N,olarge,elarge;
      clrscr();
      printf("Enter the value of N: ");
      scanf("%d",&N);
      printf("\n");

      for(i=0;i<N;i++)
      {
       printf("Enter the value of number %d: ",i+1);
       scanf("%d",&a[i]);
      }

      olarge=1;
      elarge=2;
```

Contd.

```
for(i=0;i<N;i++)
{
 if(a[i]%2==1)
  if(a[i]>olarge)
   olarge=a[i];

 if(a[i]%2==0)
  if(a[i]>elarge)
   elarge=a[i];
}

 printf("\nThe largest odd value is %d \nThe largest even value is
%d",olarge,elarge);
 getch();
}
```

Q.4. **(a)** What is inline function in C++? Give their merits and explain them with examples. **06**

(b) Give one example for each of the following and explain the meaning. **06**

1. Array of structure
2. Static data and member function

(c) Explain the turnery operator in C. **02**

OR

Q.4. **(a)** Create a C++ class for representing the point with x and y coordinates. Provide member functions to read and print the point object, adding two point objects and necessary constructors. **06**

(b) Explain the following with proper examples. **08**

1. Copy constructor
2. Initialization of one and two dimensional array in C

Q.4 (a)

Inline Function

Refer Section "16.6 Inline Functions"

Q.4 (b) (1)

Array of Structure

Refer Section "12.8 Arrays of Structures"

Q.4 (b) (2)

Static data and member function

Refer Section "17.11 Static Data Members" and "17.12 Static Member Functions"

Q.4 (c)

Ternary Operator

Refer Section "4.7 Conditional Operators"

OR

Q.4 (a)

Program

```
#include <iostream>
#include <conio.h>
using namespace std;


class point
{
 private:
  int x,y;

 public:
  point()
  {
   x=0;
   y=0;
  }

  point(int a,int b)
  {
   x=a;
   y=b;
  }

  void set(int a,int b)
  {
   x=a;
   y=b;
  }

  void print(void)
  {
   cout<< "x= " << x <<" y = "<<y <<"\n";
  }

  point operator+(point z)
  {
   point temp;
   temp.x=x+z.x;
   temp.y=y+z.y;
   return(temp);
  }
};

int main()
{
```

Contd.

```
      point a;
      point b (10,10);
      point c;

      a.set(8,-2);

      c=a+b;

      cout <<"a: "; a.print();
      cout <<"b: "; b.print();
      cout <<"c: "; c.print();

      getch();
      return 0;
     }
```

Output

```
    a: x= 8 y = -2
    b: x= 10 y = 10
    c: x= 18 y = 8
```

Q.4 (b) (1)

Copy Constructor

Refer Section "18.7 Copy Constructor"

Q.4 (b) (2)

Initialization of one and two dimensional arrays in C

Refer Section "8.4 Initialization of One Dimensional Arrays" and "8.6 Initializing Two Dimensional Arrays"

Q.5 (a) Explain the various I/O function with example in C. **06**

(b) Write a C++ program representing the circle as class to find its area, where area = PIE* radius* radius. Use it in main to create array of five circle objects, get radius for them and print the circle object with radius and area whose area is largest. **08**

OR

Q.5. (a) What is a string? How string is stored in C? Give the various functions with their use for string processing. **06**

(b) Write a C++ program with a class having three integer data members. Provide the member function along with other functions and constructors, to test whether given three number forms an Pythagorean triplet i.e. $x^2 + y^2 = z^2$. **08**

Q.5 (a)

I/O Functions

Refer Section 5.2 and 5.3

Q.5 (b)

Program

```
#include <iostream>
#include <conio.h>
using namespace std;

class circle
{
 private:
  float radius;

 public:

  void set(float a)
  {
   radius=a;
  }

  void print(void)
  {
   cout<< "radius= " << radius <<" Area = "<<3.14*radius*radius <<"\n";
  }

};

int main()
{
 circle c[5];
 int i,flag;
 float temp, large;
 large=0;
 flag=0;

 for(i=0;i<5;i++)
 {
  cout << "Enter the radius of circle "<< i+1 <<" ";
  cin >> temp;
  c[i].set(temp);
  if(temp>large)
  {
   large=temp;
    flag=i;
  }
 }

 cout << "\nCircle with largest area is \n\n";
 c[flag].print();

 getch();
```

Contd.

```
      return 0;
     }
```

Output

```
     Enter the radius of circle 1 .1
     Enter the radius of circle 2 .5
     Enter the radius of circle 3 .6
     Enter the radius of circle 4 .2
     Enter the radius of circle 5 .4
     Circle with largest area is

     radius= 0.6 Area = 1.1304
```

OR

Q.5 (a)

Strings

Refer Section 9.1, 9.2 and 9.8

Q.5 (b)

Program

```
     #include <iostream>
     #include <conio.h>
     using namespace std;


     class data
     {
      private:
       int x,y,z;

      public:
       data()
       {
        x=0;
        y=0;
        z=0;
       }

       data(int a,int b, int c)
       {
        x=a;
        y=b;
        z=c;
       }

       void set(int a,int b, int c)
       {
        x=a;
        y=b;
```

Contd.

```
        z=c;
       }

       void test(void)
       {
        if((x*x+y*y)==(z*z))
          cout<< "Numbers x= " << x <<" y = "<<y <<" z = "<<z <<" form a
    Pythagorean Triplet\n";
        else
         cout<< "Numbers x= " << x <<" y = "<<y <<" z = "<<z <<" do not form a
    Pythagorean Triplet\n";
       }

    };

    int main()
    {
     data a;
     data b(10,5,10);

     a.set(3,4,5);

     a.test();
     b.test();

     getch();
     return 0;
    }
```

Output

```
Numbers x= 3 y = 4 z = 5 form a Pythagorean Triplet
Numbers x= 10 y = 5 z = 10 do not form a Pythagorean Triplet
```

Gujarat Technological University

B.E. Sem-I & II Remedial Examination Nov/Dec. 2010

Subject code: 110003

Subject Name: Computer Programming and Utillization

Date: 30/11/2010 **Time: 10.30 am – 01.00 pm**

Total Marks: 70

Instructions

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**

Q.1 **(a)** Answer the following questions. **12**

(i) Explain in brief the features of C language.

(ii) Explain logical operator and relational operator used in C language.

(iii) Draw a flowchart to find the smallest number out of three given numbers.

(b) What is "structure"? How do we declare and access structure variables? **02**

1. (a)

(i) Features of C Language

Some of the important features of the C language are the following:

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

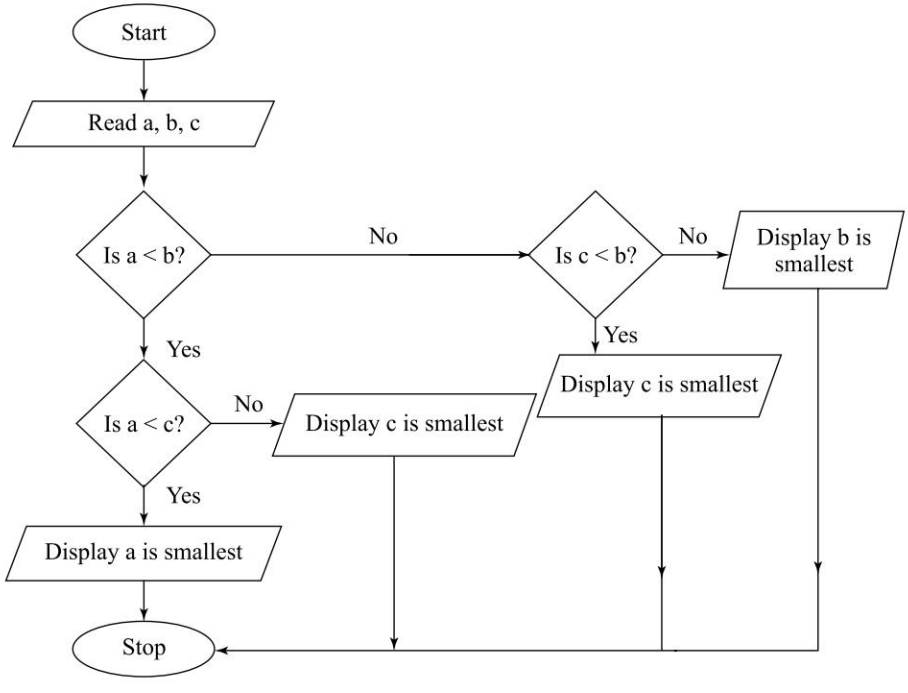
(ii) Logical Operator

Refer Section 4.4.

Relational Operator

Refer Section 4.3.

(iii) Flowchart to find the smallest number out of three given numbers:



1. (b) Structure

Refer Section 12.1.

Declaring a Structure

Refer Section 12.3.

Accessing Structure Variables

Refer Section 12.4.

Q.2 (a) Answer the following Questions. **07**

(i) Explain **"if …else…if"** ladder of C with a neat diagram and a brief program code.

(ii) Explain the concept of class and object used in C++ giving suitable example.

(b) Answer the following Questions. **(Any TWO)** **07**

(i) Compare **for loop** and **while loop** with illustration.

(ii) What are the characteristics of a constructor in C++? Explain its use with an example.

(iii) What do you mean by recursive function? Explain with a small example.

2. (a)

(i) if…else…if Ladder

Refer Section 6.6.

(ii) Classes and Objects

Class A class is a way of binding data and its associated functions together. A class definition creates an abstract data type that is similar to the other built-in data types. A typical class specification has two parts:

- *Class declaration*: Describes the type and scope of class members.
- *Class function definition*: Describes how class functions are implemented.

The following is a sample class declaration:

```
class item
   {
      int number;
      float cost;
    public:
      void getdata(int a, float b); //function prototype
      void putdata(void);
   }
```

Objects An object is an instance of a class. Just like we create instances (variables) of different built-in data types, objects are also instances of the abstract data types created using classes. The following is an example of object creation:

```
item x;
item x, y, z;
```

Here, *x, y* and *z* are objects of the item class.

2. (b)

(i) for Loop vs. while Loop

for	while
It is a concise loop-control structure.	It is less concise as compared to `for` loop.
The initialization of control variables, specification of test condition and incrementing/decrementing of control variables are all specified in a single statement.	The initialization of control variables, specification of test condition and incrementing/decrementing of control variables are all specified separately.

Example:
`for(i=0;i<10;i++)`
`{`
`.`
`.`
`statements`
`.`
`.`
`}`
	Example:
	`i=0;`
	`while(i<10)`
	`{`
	`.`
	`.`
	`statements`
	`.`
	`i = i+1;`
	`}`

(ii) Characteristics of Constructor and its Uses

Refer Section 18.2.

(iii) Recursive Function

Recursive function is a function that calls itself. Such functions are effectively used to solve problems where the solution is expressed in terms of successively applying the same solution to

subsets of the problem. While writing recursive functions, an `if` statement must be included in the code so as to force the function to return instead of executing the recursive call again.
The following code shows the definition of a power function using recursive approach:

```
double power(double x,int n)
{
  if(n==1)
    return(x);
  else
  return(x*power(x,n-1)); //recursive function call
}
```

Q.3 **(a)** Write a program to generate Fibonacci series of numbers (total 20 numbers) i.e. 1, 1, 2, 3, 5, 8….etc **05**

(b) List out the categories of functions of C. Explain any one category with an example. **04**

(c) Write a program to store 10 numbers in an array. Then find out sum, maximum and average of these 10 numbers. **05**

3. (a) Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
int num1=0, num2=1,i,fab;
int len=20;
printf("\t\t<----FIBONACCI SERIES---->");
printf("\n%d %d",num1,num2);

for(i = 1; i <= len-2; i++)
{
fab=num1 + num2;
printf(" %d",fab);
num1=num2;
num2=fab;
}
getch();
  }
```

Output

```
<----FIBONACCI SERIES---->
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

3. (b) Category of Functions

Refer Section 10.9 and 10.10.

3. (c) Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int arr[10], i;
  int sum=0, max=0;
  float avg;
  printf("Enter the 10 elements of the array:\n");
  for(i=0;i<10;i++)
  {
    printf("arr[%d] = ",i);
    scanf("%d",&arr[i]);
  }
  for(i=0;i<10;i++)
    sum=sum+arr[i];
  for(i=0;i<10;i++)
    if(arr[i]>max)
      max=arr[i];
  avg = sum/10;
  printf("\nSum = %d\nMaximum = %d\nAverage = %.2f",sum,max,avg);
  getch();
}
```

Output

```
Enter the 10 elements of the array:
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
arr[6] = 7
arr[7] = 8
arr[8] = 9
arr[9] = 99
Sum = 144
Maximum = 99
Average = 14.00
```

OR

Q.3 **(a)** Explain following string manipulation functions. **04**
strcmp(), strlen(), strcat(), strstr() ;

(b) Write a program to convert temperature entered by user from Fahrenheit into Celsius. **05**

(c) Write a C++ program which has a class named **`student`** .It has two member functions named **"read_data()" and "print_data()"**. It has member elements called **`Rollno`**, `Name`

and **Percentage**. Assume suitable data type for variable. And read the data from the user and print the same on the screen using above functions. **05**

3. (a) String Manipulation Functions

Refer Section 9.8.

3. (b) Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
  float F, C;
    printf("Enter the temperature value in Fahrenheit: ");
  scanf("%f",&F);
  C=(F-32.0)/1.8;
  printf("The equivalent temperature value in degrees Celsius is: %.2f",C);
  getch();
}
```

Output

```
Enter the temperature value in Fahrenheit: 100
The equivalent temperature value in degrees Celsius is: 37.78
```

3. (c) Program

```
#include <iostream>
#include <conio.h>
using namespace std;
class student
{
      int rollno;
      char name[20];
      float percent;
public:
      void read_data(int,char[],float);
      void print_data(void);
};
void student :: read_data(int r, char *n, float p)
{
rollno = r;
strcpy(name,n);
percent=p;
}
void student :: print_data(void)
{
      cout<<"Roll Number: "<<rollno;
      cout<<"\nName: "<<name;
      cout<<"\nPercentage: "<<percent;
```

Contd.

```
    }
    void main()
    {
         student st;
         st.read_data(21,"Anil Kumar",65.75);
         st.print_data();
         getch();
    }
Output
    Roll Number: 21
    Name: Anil Kumar
    Percentage: 65.75
```

Q.4 **(a)** Write a program to read two 5 X 5 matrices from the user and store the addition of two matrices in the resultant matrix. **06**

i.e. $C = A + B$, where A,B,C each is a 5 X 5 matrix.

(b) Write the short notes on following. **08**

(i) Polymorphism

(ii) Difference between object-oriented and procedure-oriented language

4. (a) Program

```
    #include <stdio.h>
    #include <conio.h>
    void main()
    {
    int i,j,a[5][5],b[5][5],c[5][5];
    printf("Enter the first 5 X 5 matrix:\n");
    for(i=0;i<5;i++)
    {
      for(j=0;j<5;j++)
      {
      printf("a[%d][%d] = ",i,j);
      scanf("%d",&a[i][j]);
      }
    }
    printf("Enter the second 5 X 5 matrix:\n");
    for(i=0;i<5;i++)
    {
      for(j=0;j<5;j++)
      {
      printf("b[%d][%d] = ",i,j);
      scanf("%d",&b[i][j]);
      }
    }
    printf("\nThe entered matrices are: \n");
```

Contd.

```
    for(i=0;i<5;i++)
    {
      printf("\n");
      for(j=0;j<5;j++)
      {
      printf("%d ",a[i][j]);
      }
      printf("\t\t");
      for(j=0;j<5;j++)
      {
      printf("%d ",b[i][j]);
      }
    }
    for(i=0;i<5;i++)
    {
      for(j=0;j<5;j++)
      c[i][j] =a[i][j]+b[i][j];
    }
    printf("\n\nThe sum of the two matrices is shown below: \n");
    for(i=0;i<5;i++)
    {
      printf("\n\t\t ");
      for(j=0;j<5;j++)
      {
      printf("%d ",c[i][j]);
      }
    }
    getch();
    }
```

Output

```
    Enter the first 5 X 5 matrix:
    a[0][0] = 1
    a[0][1] = 1
    a[0][2] = 1
    a[0][3] = 1
    a[0][4] = 1
    a[1][0] = 1
    a[1][1] = 1
    a[1][2] = 1
    a[1][3] = 1
    a[1][4] = 1
    a[2][0] = 1
    a[2][1] = 1
    a[2][2] = 1
    a[2][3] = 1
    a[2][4] = 1
    a[3][0] = 1
```

Contd.

```
a[3][1] = 1
a[3][2] = 1
a[3][3] = 1
a[3][4] = 1
a[4][0] = 1
a[4][1] = 1
a[4][2] = 1
a[4][3] = 1
a[4][4] = 1
Enter the second 5 X 5 matrix:
b[0][0] = 2
b[0][1] = 2
b[0][2] = 2
b[0][3] = 2
b[0][4] = 2
b[1][0] = 2
b[1][1] = 2
b[1][2] = 2
b[1][3] = 2
b[1][4] = 2
b[2][0] = 2
b[2][1] = 2
b[2][2] = 2
b[2][3] = 2
b[2][4] = 2
b[3][0] = 2
b[3][1] = 2
b[3][2] = 2
b[3][3] = 2
b[3][4] = 2
b[4][0] = 2
b[4][1] = 2
b[4][2] = 2
b[4][3] = 2
b[4][4] = 2
The entered matrices are:
1 1 1 1 1      2 2 2 2 2
1 1 1 1 1      2 2 2 2 2
1 1 1 1 1      2 2 2 2 2
1 1 1 1 1      2 2 2 2 2
1 1 1 1 1      2 2 2 2 2
The sum of the two matrices is shown below:
               3 3 3 3 3
               3 3 3 3 3
               3 3 3 3 3
               3 3 3 3 3
               3 3 3 3 3
```

4. (b)

(i) Polymorphism

Refer Section 13.5.5.

(ii) Difference between Object Oriented and Procedure Oriented Language

Refer Section 13.4

OR

Q.4 (a) What is a pointer? Give its benefits. Write a program to do swapping of two elements using function with two pointers as arguments. **06**

(b) Write the short notes on following. **08**

(i) Inheritance (ii) Friend Function

4. (a) Pointer and its Benefits

Refer Section 11.1.

Program

```
#include <stdio.h>
#include <conio.h>
void swap(int*,int*);
void main()
{
  int num1, num2;
  num1=10;
  num2=20;
  printf("Before Swap:\nnum1 = %d\tnum2 = %d",num1,num2);
  swap(&num1,&num2);
  printf("\n\nAfter Swap:\nnum1 = %d\tnum2 = %d",num1,num2);
  getch();
}
void swap(int *n1,int *n2)
{
  int temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

Output

```
Before Swap:
num1 = 10      num2 = 20
After Swap:
num1 = 20      num2 = 10
```

4. (b)

(i) Inheritance

Refer Section 13.5.4.

(ii) Friend Function

Refer Section 17.15.

Q.5 **(a)** Explain the syntax of **switch…case** statement. Write a program using `switch…case` statement. **04**

(b) Write a program to print the following triangle for *n* lines. **06**

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

…up to *n* such Lines

(c) State the difference between structure and union. **04**

5. (a) Syntax of switch…case Statement

Refer Section 6.7.

Program Demonstrating use of switch…case Statement

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int choice;
  float num1, num2;
  printf("**********Simple Calc**********");/*Displaying CalC options*/
  printf("\n\nChoose a type of operation from the following: ");
  printf("\n\t1. Addition");
  printf("\n\t2. Subtraction");
  printf("\n\t3. Multiplication");
  printf("\n\t4. Division\n");
  scanf("%d", &choice);/*Reading user's choice*/
  printf("\n\nEnter the two operands: ");
  scanf("%f %f", &num1, & num2);/*Reading operands*/
  /*Using the Switch statement to choose the right operation*/
  switch (choice)
  {
  case 1:
     printf("\n%.2f + %.2f = %.2lf", num1, num2, num1+num2);
     break;
  case 2:
     printf("\n%.2f - %.2f = %.2lf", num1, num2, num1-num2);
     break;
  case 3:
     printf("\n%.2f * %.2f = %.2lf", num1, num2, num1*num2);
     break;
  case 4:
     printf("\n%.2f / %.2f = %.2lf", num1, num2, num1/num2);
     break;
  default:
     printf("\nIncorrect Choice!");
     }
     getch();
}
```

5. (b) Program

```
#include <stdio.h>
#include <conio.h>
void main ()
{
  int n,i,j;
  printf("Enter the value of n: ");
  scanf("%d",&n);
  printf("\n");
  for(i=1;i<=n;i++)
  {
 for(j=0;j<=n-i;j++)
  printf(" ");
 for(j=1;j<=i;j++)
  printf( "* ");
 printf("\n");
 }
 getch();
}
```

Output

```
Enter the value of n: 5
       *
      * *
     * * *
    * * * *
   * * * * *
```

5. (c) Difference between Structure and Union

Structure	Union
A structure is defined with 'struct' keyword.	A union is defined with 'union' keyword.
All members of a structure can be manipulated simultaneously.	The members of a union can be manipulated only one at a time.
The size of a structure object is equal to the sum of the individual sizes of the member objects.	The size of a union object is equal to the size of largest member object.
Structure members are allocated distinct memory locations.	Union members share common memory space for their exclusive usage.
Structures are not considered as memory efficient in comparison to unions.	Unions are considered as memory efficient particularly in situations when the members are not required to be accessed simultaneously.

Example:
`struct book`
`{`
`char title[25];`
`char author[25];`
`int pages;`
`float price;`
`};`
	Example:
	`union result`
	`{`
	`int marks;`
	`char grade;`
	`float percent;`
	`};`

OR

Q.5 **(a)** Explain the different data types of the C language. **05**

(b) Write a C program to check whether the given number is prime or not. **05**

(c) What is inline function? Explain with an example. **04**

5. (a)

C Data Types

Refer Section 3.7

5. (b) Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int num,i;
  printf("Enter a number: ");
  scanf("%d",&num);/*Reading a number*/
  for(i=2;i<num/2;i++)
    if(num%i==0)/*Checking if num is divisible by another number*/
      {
      printf("\n%d is not a prime number",num);
      break;
      }
      else
      ;
  if(i>=num/2)
  printf("\n%d is a prime number",num);
  getch();
}
```

Output

```
Enter a number: 103
103 is a prime number
```

5. (c) Inline Function Refer Section 16.6.

Gujarat Technological University

B.E. Sem-Ist Regular Examination January 2011

Subject code: 110003

Subject Name: Computer Programming and Utillization

Date: 05/01/2011 **Time: 10.30 am – 01.00 pm**

Total Marks: 70

Instructions

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**

Q.1 **(a)** Explain the 'C' development life cycle using a flowchart in detail. **06**

(b) Explain the storage class specifiers in C. **04**

(c) Write an algorithm to find out the factorial of given numbers and draw a flowchart for it. **04**

1. (a) C Development Life Cycle

Refer Section 2.9.

1. (b) Storage Class Specifiers

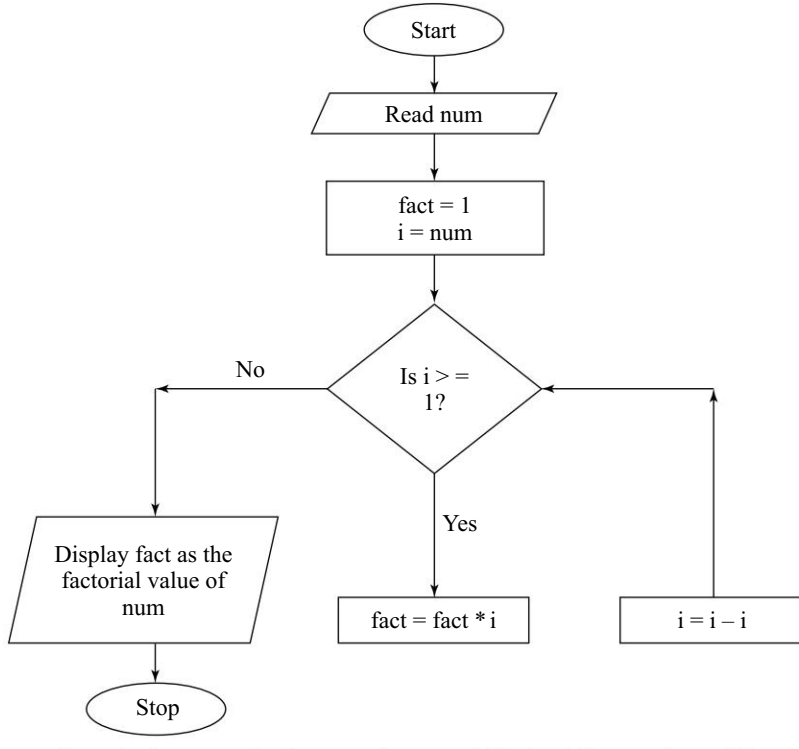
C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. The following table lists the various storage class specifiers along with their meanings:

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

1. (c) Algorithm

```
Step 1 - Start
Step 2 - Read num
Step 3 - Initialize fact = 1
Step 4 - Initialize the looping counter i = num
Step 5 - Repeat Steps 6-7 while i>=1
Step 6 - fact = fact * i;
Step 7 - i=i-1;
Step 8 - Display fact as the factorial value of num
Step 9 - Stop
```

Flowchart



Q.2 **(a)** What are trigraph characters? How are they useful? Explain any five of them. **05**

(b) Explain the classification of operators in C in brief. **07**

2. (a) Trigraph Characters

Trigraph sequences provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters, i.e., two question marks followed by a character. Trigraph sequences are particularly useful in situations where a special character is to be used in a program but is not directly available on the keyboard.

The following table shows the various trigraph sequences:

Trigraph Sequence	Trigraph Character
??=	#
??(	[
??)	]
??!	|
??-	~

2. (b) Operator Classification

Refer Sections 4.1 – 4.9.

OR

(b) What is type conversion? Explain the type conversions supported by the C language. **07**

(c) What will be the output of the following program?

```
main()
{
int x = 10, y, z;
z = y = x ;
y - = x-- ;
z - = --x ;
x - = --x - x-- ;
printf(‘ y = %d z = %d x = %d ‘ , y, z, x) ;
} 02
```

2. (b) Type Conversion

Refer Section 4.14.

2. (c) Output

```
y = 0 z = 2 x = 6
```

Q.3 (a) Write a C program to determine whether a given number is ‘odd’ or ‘even’ and print the following message:

NUMBER IS EVEN

Or

NUMBER IS ODD

(i) Without using **else** option (ii) With **else** option **05**

(b) Explain the **break and continue** statements using suitable example. **05**

(c) Explain **getchar() and gets()** functions with suitable examples. **04**

3. (a) (i) Without using else option

Program

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
int num;
printf("Enter a number: ");
scanf("%d",&num);
if(num%2==0)
{
     printf("\nNUMBER IS EVEN");
     getch();
     exit(1);
}
printf("\nNUMBER IS ODD");
getch();
}
```

3. (a) (ii) With else option

Program

```
void main()
{
int num;
printf("Enter a number: ");
scanf("%d",&num);
if(num%2==0)
  printf("\nNUMBER IS EVEN");
else
  printf("\nNUMBER IS ODD");
getch();
}
```

3. (b) Break and Continue Statements

Refer Section 7.5.

3. (c) getchar( ) and gets( ) Functions

Refer Section 9.3.3.

OR

Q.3 (a) Write a C program to evaluate the square root of five numbers using the **goto** statement. **05**

(b) Write the basic differences between the **while loops** and **do-while loop** with suitable examples. **05**

(c) Explain **putchar( )** and **puts( )** functions with a suitable example. **04**

3. (a)

Program

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
  float num[5] = {2.0, 7.5, 10.0, 24.0, 55.55};
  int i=0;
    label:
    printf("num[%d] = %.2f\tSquare Root = %.2f\n",i,num[i],sqrt(num[i]));
  i=i+1;
  if(i<5)
    goto label;
  else
  ;
  getch();
}
```

```
Output
   num[0] = 2.00   Square Root = 1.41
   num[1] = 7.50   Square Root = 2.74
   num[2] = 10.00  Square Root = 3.16
   num[3] = 24.00  Square Root = 4.90
   num[4] = 55.55  Square Root = 7.45
```

3. (b) Difference between **while** and **do while**

While...Do	Do...While
It is an entry-controlled loop.	It is an exit-controlled loop.
In **while...do** or simply **while** loop, the test condition is checked first and the body of the loop is executed only if the test condition is true.	In **do...while** loop, the body of the loop is executed first and then the test condition is checked to determine whether the next iteration of the loop will be executed or not.
The body of the loop may never get executed even once if the test condition fails in its very first evaluation.	The body of the loop will be executed at least once even if the test condition fails in its very first evaluation.
Example	**Example**
`. . int i; i=1; while(i<=10) { printf("%d",i); printf("\n"); i=i+1; } . .`	`. . int i; i=1; do { printf("%d",i); printf("\n"); i=i+1; } while(i<=10); . .`

3. (c) purchar( ) and **puts( )** functions

Refer Section 9.4.2

Q.4 (a) Write a C program in which a 5-digit positive integer is entered through the keyboard; write a function to calculate the sum of digits of the 5-digit number using recursion. **05**

(b) Differentiate between structures and unions with examples. **05**

(c) Explain dynamic memory allocation functions with syntax. **04**

4. (a) Program

```
#include <stdio.h>
#include <conio.h>
int sow(int);
void main()
{
long num, temp;
printf("Enter a five digit integer value: ");
```

Contd.

```
    scanf("%ld",&num);
    printf("The sum of digits of %ld is %d",num,sow(num));
    getch();
    }
    int sow(int x)
    {
      if(x%10==0)
        return x;
      else
         return (x%10+sow(x/10));
    }
```

Output

```
    Enter a five digit integer value: 12345
    The sum of digits of 12345 is 15
```

4. (b) Difference between structure and union

Structure	Union
A structure is defined with **'struct'** keyword.	A union is defined with **'union'** keyword.
All members of a structure can be manipulated simultaneously.	The members of a union can be manipulated only one at a time.
The size of a structure object is equal to the sum of the individual sizes of the member objects.	The size of a union object is equal to the size of largest member object.
Structure members are allocated distinct memory locations.	Union members share common memory space for their exclusive usage.
Structures are not considered as memory efficient in comparison to unions.	Unions are considered as memory efficient, particularly in situations when the members are not required to be accessed simultaneously.

Example:
`struct book`
`{`
`char title[25];`
`char author[25];`
`int pages;`
`float price;`
`};`
	Example:
	`union result`
	`{`
	`int marks;`
	`char grade;`
	`float percent;`
	`};`

4. (c) Dynamic Memory Allocation

The process of allocating memory at run time is known as dynamic memory allocation. C supports dynamic memory allocation through four memory-management functions, viz., **malloc( ), calloc( ), free( )** and **realloc( )**. These functions help in allocating and freeing memory during program execution.

Syntax

Following is the syntax of using **malloc( )** function for the purpose of dynamic memory allocation:

```
ptr = (cast-type *) malloc(byte-size);
```

Here, the malloc() function returns a pointer (of cast-type) to an area of memory with size byte-size.

Example `x = (int *) malloc(100*sizeof(int));`

OR

Q.4 **(a)** Explain the various types of function calls in C. **05**

(b) Explain the various ways to access the structure members with proper syntax in the C language. **05**

(c) List the advantages and limitations of arrays in C. **04**

4. (a) Types of Function Calls

The various types of function calls in C are

- Call by value
- Call by reference

Call by value

In this technique, a function is called by passing the value of a variable as argument. The passed value is copied into another variable in the parameter list of the function. Thus, any change made to the new variable's value inside the called function is not reflected back in the calling function. The following code snippet depicts this scenario:

```
Program
   void change(int n)
   {
        n=100;
   }
   void main()
   {
        int num=50;
        printf("Before function call, num = ",num);
        change(num);
        printf("\nAfter function call, num = ",num);
   getch();
   }
Output
   Before function call, num = 50
   After function call, num = 50
```

Call by reference

In this technique, the memory addresses of the variables rather than the copies of values are passed to the function. Thus, any changes made to the values stored at the memory addresses stay permanent and get reflected back to the main function. The following code snippet depicts this scenario:

Code

```
    void change(int *n)
    {
         *n=100;
    }
    void main()
    {
      int num=50;
      printf("Before function call, num = ",num);
      change(&num);
      printf("\nAfter function call, num = ",num);
    getch();
    }
Output
    Before function call, num = 50
    After function call, num = 100
```

4. (b) Ways of Accessing Structure Members

Refer Sections 12.4 and 12.12.

4. (c) Advantages of Arrays

The various advantages of arrays are the following:

- An array helps to logically group together objects of similar type.
- It allows the programmer to use a single variable name for referring to multiple data values.
- Arrays are particularly effective in representing sequential data.
- Arrays allow the storage of multi-dimensional data sets.

Limitations of Arrays

The various limitations of arrays are the following:

- An array does not support grouping of dissimilar data types.
- Addition or deletion of elements at the middle of the array requires shifting of adjoining elements.
- The use of arrays may lead to inefficient utilization of memory resources as it is not based on dynamic memory allocation technique.

Q.5 (a) State the benefits of object-oriented programming over procedure-oriented programming. **07**

(b) What is an inline function? How is it different from macros? Write a simple C++ program which shows the difference between them. **07**

5. (a) **Benefits of Object Oriented Programming over Procedure Oriented Programming**

Refer Sections 13.4 and 13.6.

5. (b)

Inline Function

A function defined as inline is expanded at the time of its invocation. That means, the compiler replaces the function call with the actual function code. The purpose of defining a function as

inline is to save the overhead of function calls for smaller functions that have very few lines of code.

Difference between Inline Functions and Macros

Inline Function	Macro
Inline functions are expanded by the compiler during program compilation.	Macros are expanded by the pre-processor before the actual program compilation starts.
It supports error checking during program compilation.	It does not support error checking during program compilation.
Type safety is applied on inline functions just like other normal functions.	This is not the case with macros.

The following sample program demonstrates the use of macros and inline functions.

```
Program
   #include <iostream>
   #include <conio.h>
   using namespace std;
   #define max(num1, num2) (num1 > num2 ? num1 : num2)
   class test
   {
   public:
     inline int max_f(int num1, int num2);
   };
   inline int test :: max_f(int num1, int num2)
   {
     return num1 > num2 ? num1 : num2;
   }
   void main()
   {
     test t1;
     cout<<max(99, -27);
     cout<<"\n"<<t1.max_f(8,4);
     getch();
   }
```

OR

Q.5 (a) What is a constructor function? List at least five characteristics of a constructor function. **07**

(b) What is a friend function? Write a simple C++ program which shows a friend function as a bridge between classes. **07**

5. (a) Constructor and its Characteristics

Refer Section 18.2.

5. (b) Friend Function

Refer Section 17.15.

C++ Program

Refer Figure 17.16 in Section 17.15.