

**THIRD
EDITION**

Programming in **C**

**ASHOK N. KAMTHANE
AMIT A. KAMTHANE**

ALWAYS LEARNING

PEARSON

PROGRAMMING IN C

THIRD EDITION

Ashok N. Kamthane

Associate Professor

Shri Guru Gobind Singhji Institute of Engineering
and Technology, Nanded, Maharashtra, India

PEARSON

Delhi • Chennai



Associate Editor—Acquisitions: Neha Goomer
Editor—Production: G Sharmilee

Copyright © 2015 Pearson India Education Services Pvt. Ltd

Copyright © 2006, 2012 Dorling Kindersley (India) Pvt. Ltd

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of both the copyright owner and the publisher of this book.

ISBN 978-93-325-4355-3

First Impression

Published by Pearson India Education Services Pvt. Ltd, CIN: U72200TN2005PTC057128, formerly known as TutorVista Global Pvt. Ltd, licensee of Pearson Education in South Asia.

Head Office: A-8(A), 7th Floor, Knowledge Boulevard, Sector 62, Noida 201 309, Uttar Pradesh, India.

Registered Office: Module G4, Ground Floor, Elnet Software City, TS-140, Block 2 & 9,

Rajiv Gandhi Salai, Taramani, Chennai 600 113, Tamil Nadu, India.

Fax: 080-30461003, Phone: 080-30461060

www.pearson.co.in, Email: companysecretary.india@pearson.com

Compositor: Mukesh Technologies Pvt. Ltd

Printed in India

CONTENTS

	<i>Preface</i>	xi
	<i>Acknowledgements</i>	xiv
	<i>About the Author</i>	xv
1	Basics and Introduction to C	2
	1.1 Why to Use Computers?	3
	1.2 Basics of a Computer	3
	1.3 Latest Computers	5
	1.4 Introduction to C	6
	1.5 About ANSI C Standard	7
	1.6 Machine, Assembly and High-Level Language	7
	<i>1.6.1 Assembly Language</i>	8
	<i>1.6.2 High-Level Language</i>	8
	1.7 Assembler, Compiler and Interpreter	8
	1.8 Structure of a C Program	10
	1.9 Programming Rules	12
	1.10 Executing the C Program	12
	1.11 Standard Directories	16
	1.12 The First C Program	16
	1.13 Advantages of C	17
	1.14 Header Files	17
	1.15 Algorithm	18
	<i>1.15.1 Program Design</i>	20
	1.16 Classification of Algorithms	21
	1.17 Flowcharts	22
	1.18 Pseudocode	26
	<i>Summary</i>	28
	<i>Exercises</i>	28
2	The C Declarations	32
	2.1 Introduction	33
	2.2 The C Character Set	33
	2.3 Delimiters	34

2.4	Types of Tokens	35
2.5	The C Keywords	35
2.6	Identifiers	35
2.7	Constants	36
	2.7.1 Numerical Constants	37
	2.7.2 Character Constant	38
2.8	Variables	39
2.9	Rules for Defining Variables	39
2.10	Data Types	40
2.11	C Data Types	41
2.12	Integer and Float Number Representations	43
	2.12.1 Integer Representation	43
	2.12.2 Floating-Point Representation	44
2.13	Declaring Variables	44
2.14	Initializing Variables	45
2.15	Dynamic Initialization	46
2.16	Type Modifiers	46
2.17	Type Conversion	47
2.18	Wrapping Around	49
2.19	Constant and Volatile Variables	50
	2.19.1 Constant Variable	50
	2.19.2 Volatile Variable	50
	<i>Summary</i>	51
	<i>Exercises</i>	52

3

Operators and Expressions	58
3.1 Introduction	59
3.2 Operator Precedence	59
3.3 Associativity	62
3.4 Comma and Conditional Operator	63
3.5 Arithmetic Operators	64
3.6 Relational Operators	67
3.7 Assignment Operators and Expressions	68
3.8 Logical Operators	71
3.9 Bitwise Operators	74
<i>Summary</i>	80
<i>Exercises</i>	80

4

Input and Output in C 84

4.1	Introduction	85
4.2	Formatted Functions	86
4.3	Flags, Widths and Precision with Format String	88
4.4	Unformatted Functions	99
4.5	Commonly Used Library Functions	104
4.6	Strong Points for Understandability	105
	<i>Summary</i>	<i>106</i>
	<i>Exercises</i>	<i>106</i>

5

Decision Statements 112

5.1	Introduction	113
5.2	The if Statement	114
5.3	The if-else Statement	116
5.4	Nested if-else Statements	120
5.5	The if-else-if Ladder Statement	121
5.6	The break Statement	129
5.7	The continue Statement	130
5.8	The goto Statement	130
5.9	The switch Statement	132
5.10	Nested switch case	144
5.11	The switch case and nested ifs	146
	<i>Summary</i>	<i>147</i>
	<i>Exercises</i>	<i>148</i>

6

Loop Control 154

6.1	Introduction	155
	6.1.1 What is a Loop?	155
6.2	The for Loop	156
6.3	Nested for Loops	172
6.4	The while Loop	194
6.5	The do-while Loop	207
6.6	The while Loop within the do-while Loop	212
6.7	Bohm and Jacopini's Theory	213
	<i>Summary</i>	<i>213</i>
	<i>Exercises</i>	<i>213</i>

7

Data Structure: Array	220
7.1 Introduction	221
7.2 Array Declaration	221
7.3 Array Initialization	222
7.4 Array Terminology	223
7.5 Characteristics of an Array	224
7.6 One-Dimensional Array	226
7.7 One-Dimensional Array and Operations	238
7.8 Operations with Arrays	240
7.9 Predefined Streams	246
7.10 Two-Dimensional Array and Operations	251
7.10.1 Insert Operation with Two-Dimensional Array	253
7.10.2 Delete Operation with Two-Dimensional Array	255
7.11 Three- or Multi-Dimensional Arrays	269
7.12 The <code>sscanf()</code> and <code>sprintf()</code> Functions	274
7.13 Drawbacks of Linear Arrays	276
<i>Summary</i>	276
<i>Exercises</i>	277

8

Strings and Standard Functions	286
8.1 Introduction	287
8.2 Declaration and Initialization of String	287
8.3 Display of Strings with Different Formats	289
8.4 String Standard Functions	291
8.5 String Conversion Functions	315
8.6 Memory Functions	316
8.7 Applications of Strings	318
<i>Summary</i>	328
<i>Exercises</i>	328

9

Pointers	336
9.1 Introduction	337
9.2 Features of Pointers	338
9.3 Pointers and Address	338
9.4 Pointer Declaration	339
9.5 The Void Pointers	345
9.6 Wild Pointers	346
9.7 Constant Pointers	347
9.8 Arithmetic Operations with Pointers	348

10

9.9	Pointers and Arrays	352
9.10	Pointers and Two-Dimensional Arrays	357
9.11	Pointers and Multi-Dimensional Arrays	358
9.12	Array of Pointers	360
9.13	Pointers to Pointers	362
9.14	Pointers and Strings	364
	<i>Summary</i>	371
	<i>Exercises</i>	371
10	Functions	378
10.1	Introduction	379
10.2	Basics of a Function	379
	10.2.1 Why Use Functions?	379
	10.2.2 How a Function Works?	380
10.3	Function Definition	380
10.4	The <code>return</code> Statement	384
10.5	Types of Functions	386
10.6	Call by Value and Reference	393
10.7	Function Returning More Values	395
10.8	Function as an Argument	396
10.9	Function with Operators	399
10.10	Function and Decision Statements	406
10.11	Function and Loop Statements	409
10.12	Functions with Arrays and Pointers	412
10.13	Passing Array to a Function	418
10.14	Nested Functions	420
10.15	Recursion	422
10.16	Types of Recursion	424
10.17	Rules for Recursive Function	425
10.18	Direct Recursion	427
10.19	Indirect Recursion	428
10.20	Recursion Versus Iterations	430
10.21	The Towers of Hanoi	432
10.22	Advantages and Disadvantages of Recursion	435
10.23	Efficiency of Recursion	436
10.24	Library Functions	437
	<i>Summary</i>	438
	<i>Exercises</i>	438

11

Storage Classes	446
11.1 Introduction	447
<i>11.1.1 Lifetime of a Variable</i>	447
<i>11.1.2 Visibility of a Variable</i>	448
11.2 Automatic Variables	448
11.3 External Variables	450
11.4 Static Variables	453
11.5 Static External Variables	454
11.6 Register Variables	455
<i>Summary</i>	456
<i>Exercises</i>	456

12

Preprocessor Directives	462
12.1 Introduction	463
12.2 The #define Directive	463
12.3 Undefining a Macro	466
12.4 Token Pasting and Stringizing Operators	467
12.5 The #include Directive	468
12.6 Conditional Compilation	469
12.7 The #ifndef Directive	471
12.8 The #error Directive	472
12.9 The #line Directive	473
12.10 The #pragma inline Directive	473
12.11 The #pragma save regs	473
12.12 The #pragma Directive	473
12.13 The Predefined Macros in ANSI and TURBO-C	475
12.14 Standard I/O Predefined Streams in stdio.h	478
12.15 The Predefined Macros in ctype.h	478
12.16 Assertions	480
<i>Summary</i>	481
<i>Exercises</i>	481

13

Structure and Union	490
13.1 Introduction	491
13.2 Features of Structures	491
13.3 Declaration and Initialization of Structures	492
13.4 Structure within Structure	496

14

13.5	Array of Structures	498
13.6	Pointer to Structure	501
13.7	Structure and Functions	504
13.8	typedef	506
13.9	Bit Fields	510
13.10	Enumerated Data Type	513
13.11	Union	517
13.12	Calling BIOS and DOS Services	518
13.13	Union of Structures	524
	<i>Summary</i>	524
	<i>Exercises</i>	525
Files		532
14.1	Introduction of a File	533
14.2	Definition of File	533
14.3	Streams and File Types	534
	14.3.1 File Types	534
14.4	Steps for File Operations	535
	14.4.1 Opening of File	536
	14.4.2 Reading a File	536
	14.4.3 Closing a File	536
	14.4.4 Text Modes	537
	14.4.5 Binary Modes	543
14.5	File I/O	545
14.6	Structures Read and Write	553
14.7	Other File Function	558
14.8	Searching Errors in Reading/Writing Files	561
14.9	Low-Level Disk I/O	572
14.10	Command Line Arguments	576
14.11	Application of Command Line Arguments	577
14.12	Environment Variables	579
14.13	I/O Redirection	580
	<i>Summary</i>	582
	<i>Exercises</i>	582

15

Graphics		590
15.1	Introduction	591
15.2	Initialization of Graphics	591
15.3	Few Graphics Functions	591

16

15.4	Programs Using Library Functions	593
15.4.1	Program on Moving Moon	596
15.5	Working with Text	598
15.5.1	Stylish Lines	599
15.6	Filling Patterns with Different Colours and Styles	600
15.7	Mouse Programming	604
15.8	Drawing Non-common Figures	607
	Summary	608
	Exercises	608
	Dynamic Memory Allocation and Linked List	612
16.1	Dynamic Memory Allocation	613
16.2	Memory Models	613
16.3	Memory Allocation Functions	614
16.4	List	621
16.5	Traversal of a List	621
16.6	Searching and Retrieving an Element	622
16.7	Predecessor and Successor	623
16.8	Insertion	624
16.9	Linked Lists	626
16.10	Linked List with and without Header	627
16.10.1	Linked List with Header	627
	Summary	636
	Exercises	637
	Appendix A	641
	American Standard Code for Information Interchange	641
	Appendix B	649
	Priority of Operators and Their Clubbing	649
	Appendix C	651
	Header Files and Standard Library Functions	651
	Appendix D	655
	ROM-BIOS Services	655
	Appendix E	657
	Scan Codes of Keyboard Keys	657
	Index	660

PREFACE

I am indeed very delighted to present the third edition of *Programming in C* with elaborated concepts supported with more solved and unsolved problems. I have tried to make the book friendly using simple and lucid language. In this edition, a chapter on Graphics with thought provoking questions and programming examples are added.

This book is proposed for beginners, intermediate level students, and for all those who are pursuing education in computers. It would be extremely useful for the students who enroll for diploma, degree in science and engineering, certificate courses in computer languages in training institutes or those who appear for the C aptitude tests/interviews on C Language conducted by various software companies and enhance their C knowledge. It can be used as a reference book for those who want to learn or enrich their knowledge in C.

All the programs given in this book are compiled and run on Turbo C compiler. A few applications are provided in this book which are fully tested and run on Turbo C compiler. The programmer can develop advanced applications based on real-life problems using basics of C language. It contains numerous examples which include solved and unsolved programming exercises that make the book most interesting. Multiple choice questions are also provided at the end of each chapter for testing the skills of a programmer.

An attempt has been made to cover the C syllabi of different branches of various universities. Hence this book can be adopted as a text or a reference book in engineering/degree/diploma and other courses.

In order to bridge the gap between theory and practical, each concept is explained at length in an easy-to-understand manner supported with numerous worked-out examples and programs. The book contains solved illustrative problems and exercises. The programmer can run the solved programs, can see the output and enjoy the concepts of C.

BOOK ORGANIZATION

The first chapter describes the fundamental concepts of a computer, components of a computer, an overview of compilers and interpreters, structure of a 'C' program, programming rules, how to execute the program, and flowchart for execution of a program. This chapter also presents the techniques of solving a problem using algorithm and flowchart.

Chapter 2 explains the fundamentals of 'C'. These concepts are essential for writing programs and contain character set supported by C language. Various delimiters used with 'C' statements, keywords and identifiers are also provided. Different constants, variables and data types supported by C are also given. This chapter covers the rules for defining variables and methods to initialize them. Dynamic initialization is also presented in this chapter. Type conversion of a variable, type modifiers and wrapping around, constant and volatile variables are also explained.

Chapter 3 covers various C operators and their priorities. This chapter presents arithmetic, relational and logical operators. It also embodies increment, decrement (unary operators) and assignment operators. Other operators such as comma, conditional operator and bitwise operators are presented with programming examples.

Chapter 4 deals with formatted input and output functions such as `scanf()` and `printf()` functions. The unformatted functions such as `putchar()`, `getche()` and `gets()` are described in this chapter. Different data types and conversion symbols to be used in the C programs have also been elaborated. The special symbols such as escape sequences together with their applications are also discussed. Few of the commonly used library functions to be used in the programs such as `clrscr()` and `exit()` are also described.

Chapter 5 is essential for knowing the decision-making statements in C language. This chapter presents how to transfer the control from one part to the other part of the program. The programmer can make the program powerful by using control statements such as `if`, `if-else`, `nested if-else` statements and `switch case`. To change the flow of the program, the programmer can use keywords such as `break`, `continue` and `goto`.

Chapter 6 is devoted to control loop structures in which how statements are executed several times until a condition is satisfied. In this chapter, the reader follows program loop which is also known as iterative structure or repetitive structure. Three types of loop control statements are illustrated with `for`, `while` and `do-while` programming examples. Syntaxes of these control statements are briefed together with programming examples. The other statements such as the `break`, `continue` and `goto` statements are also narrated.

Chapter 7 deals with the array in which the reader can follow how to initialize array in different ways. The theme of this chapter is to understand the array declaration, initialization, accessing array elements and operations on array elements. How to specify the elements of one-, two- and three- or multi-dimensional arrays are explained in detail together with ample examples. The functions such as `sscanf()` and `sprintf()` are demonstrated through programming examples. The reader can develop programs after learning this chapter on arrays. This chapter also gives an overview of the string. It covers operations on array such as deletion, insertion and searching an element in the array and how to traverse all the array elements.

Chapter 8 is focused on strings. This chapter teaches you how to learn declaration and initialization of a string. It is also very important to identify the end of the string. This is followed by `NULL` (`'\0'`) character. The various formats for display of the strings are demonstrated through numerous examples.

String handling has strong impact in real-life string problems such as conversion of lower to upper case, reversing, concatenation, comparing, searching and replacing of string elements. It is also discussed how to perform these operations with and without standard library functions. Memory functions such as `memcpy()`, `memmove()` and `memchr()` are also illustrated together with programming examples.

Chapter 9 deals with the most important feature of the C language, i.e. pointer, it is important but difficult to understand easily. The reader is made familiar with pointers with numerous examples. The reader is brought to light about declaration and initialization of pointers, and how to access variables using pointers. How pointers are used to allocate memory dynamically at run time is also illustrated using memory allocation functions such as `malloc()` and `calloc()` functions. How memory is handled efficiently with pointers is also explained. This chapter consists of arithmetic operations on pointers, pointers and arrays, pointers to pointers and pointers to strings. Memory models are also explained.

Chapter 10 is one more important chapter on functions. How a large size program is divided in smaller ones and how a modular program should be developed is learnt in this chapter. Programmer learns the definition and declaration of function. What are the `return` statements, types of functions and functions with passing arguments are described in detail. What do you mean by “call by value” and “call by reference”? — Their answers are given with many programming examples. This chapter also incorporates functions and loop statements, function and arrays and association of functions and pointers.

The reader should know that the function always returns an integer value. Besides a function can also return a non-integer data type but function prototype needs to be initialized at the beginning of the program. The recursive nature of function is also explained with suitable example. Direct and indirect recursive functions have been explained with programming examples.

Chapter 11 enlightens on the variables used in C in different situations. It also covers types of variables such as local and global variables. The various storage classes of a variable are also covered in this chapter. Explanations on `auto`, `extern`, `static` and `register` variables are also presented in this chapter.

Chapter 12 narrates how to make use of preprocessor directives and how various macros are to be used. This chapter enlightens preprocessor directives such as `#define`, `#undef`, `#include`, `#line`, token pasting and stringizing operations and conditional compilation through C are illustrated. It covers `#define` directive, `undef` macro, `include` directive, predefined macros in ANSI and Turbo C. A reader learns how to display programmer's own error messages using `#error` directive and making various warnings on/off displayed by compiler using `#pragma` directive. You are exposed to predefined macros in `ctype.h` in this chapter.

Chapter 13 is on structures and unions. A reader can get derived data type using structures and unions. User can decide the heterogeneous data types to be included in the body of a structure. Use of dot operator (.) and pointer (->) are explained for accessing members of structure. Declaration and initialization of structure and union are also explained. The `typedef` facility can be used for creating user-defined data types and illustrated with many examples. Enumerated data type and union are the important subtitles of this chapter. Enumerated data type provides user-defined data types. Union is a principal method by which the programmer can derive dissimilar data types. The last but not the least the DOS and ROM-BIOS functions and their applications are also explained.

Chapter 14 is on files. This chapter explains the procedure for opening a file, storing information and reading. How to read a file and how to append information are explained in this chapter. Many file handling commands are also discussed. Text and binary files are explained. Command line arguments to accept arguments from command prompt are described. Simulation of various DOS commands with examples is also narrated. A reader is also made familiar with I/O redirections in which MSDOS redirects to send the result to disk instead of seeing information on monitor.

Chapter 15 is on graphics. How to draw various figures/images using C library graphics functions are to be studied from graphics chapter. This chapter enlightens the reader about the initialization of graphics with library graphics functions and number of programming examples. Few programs have been provided on mouse programming.

Chapter 16 enlightens the reader on dynamic memory allocations, memory models and linked lists. Dynamic memory allocation deals with memory functions such as `malloc()`, `calloc()`, `coreleft()` and `realloc()` and release the allocated memory using `free()` function. The linked list is described in brief in this chapter. In the linked list, creation of linked list, traversing, searching, inserting and deleting an element are described with figures and programming examples.

Utmost care has been taken to write third edition of the book in order to make it error free. The suggestions and feedback for the improvement of the book are always welcome, the readers can directly mail me at ankamthane@gmail.com.

Ashok N. Kamthane

ACKNOWLEDGEMENTS

I would like to thank all those who have encouraged me, especially Professor B.M. Naik, former principal of Shri Guru Gobind Singhji Institute of Engineering and Technology, who has been always a source of inspiration.

Special thanks are due to members of board of governors of SGGS institute who motivated me for writing this book, Baba Kalyani, Chairman and Managing Director of Bharat Forge Ltd Pune, Ram Bhogle, C.Y. Gavhane, Mr Kamlesh Pande, Sanjay Kumar, Dr Nirmal Singh Sehra, and Director of our Institute Dr L.M. Waghmare.

My sincere thanks to Professor S.D. Mahajan, Director of Technical Education, Maharashtra State, and ex-Board of Governors of this college, Dr M.B. Kinhalkar, Former Home Minister and ex-Principal, Dr T.R. Sontakke for inspiring me to write this book.

I am grateful to all my colleagues, friends and students, who extended morale support, Dr Y.V. Joshi, Dr R.R. Manthalkar, Dr S.S. Gajre, Dr S.V. Bonde, Dr P.G. Jadhav, Professor N.G. Megde, Professor P.S. Nalawade, Dr A.R. Patil, Dr A.B. Gonde, Dr M.B. Kokre, Dr U.V. Kulkarni, Dr P. Pramanik, Dr V.M. Nandedkar, Dr A.V. Nandedkar, Dr B.M. Patre, Dr S.T. Hamde, Dr R.C. Thool, Dr V.R. Thool, Mrs U.R. Kamble, Dr D.D. Doye, Dr V.G. Asutkar, Professor R.K. Chavan, Professor Ghanwat Vijay, Dr V.K. Joshi, Dr S.G. Kahalekar, Dr A. Chakraborty, Dr P. Kar, Dr P.G. Solankar, Dr B.M. Dabde, Dr M.L. Waikar, Dr R.S. Holambe, Dr J.V.L. Venkatesh, Mrs S.S. Kandhare, Professor S.S. Hatkar, Narayan Patil, Dr P.D. Dahe, Dr P.D. Ullagadi, Dr P.B. Londhe, Dr A.S. Sontakke, Professor A.M. Bainwad, Professor Deepak Bacchewar, Professor R.P. Parvekar, Professor N.M. Khandare, Dr V.B. Tungikar, Dr R.N. Joshi, Dr L.G. Patil, Professor A.I. Tamboli, Mr Bhalerao M.V. and Professor S.B. Dethe.

I am also thankful to my friends, Professor S.L. Kotgire, Maruti Damkondawar, G.M. Narlawar, Anil Joshi, D.V. Deshpande, Professor Balaji Bacchewar, M.M. Jahagirdar, L.M. Buddhewar, K.M. Buddhewar, S.R. Kokane, Ganpat Shinde, M.G. Yeramwar, S.R. Tumma, S.P. Tokalwad, P.R. Navghare, Somajawar H.S. and Annes for their morale support.

I am thankful to the wonderful editorial team of Pearson Education for specific invaluable inputs and bringing this book out in a record time.

I also express my thanks to my son Lecturer Amit, students Jadhav Gopal and Wanjare Sainath for their critical review and suggesting improvements.

Last but not the least, my thanks are due to my wife Surekha for her patience and support. My son Amol, daughter Sangita and daughter-in-law Swaroopa were of great help and supported me all the times.

Ashok N. Kamthane

ABOUT THE AUTHOR

Ashok N. Kamthane is presently working as an associate professor in Department of Electronics and Telecommunication Engineering at Shri Guru Gobind Singhji Institute of Engineering and Technology, Nanded, Maharashtra. He has over 32 years of teaching experience. He was associated with the development of hardware and software using 8051 on acoustic transceiver system for submarines. Professor Kamthane is also the author of bestselling books *Object-Oriented Programming with ANSI and Turbo C++*; *Introduction to Data Structures in C*; and *C Programming: Test Your Skills*, published by Pearson Education.

This page is intentionally left blank

CHAPTER

1

Basics and Introduction to C

Chapter Outline

- 1.1** Why to use Computers?
- 1.2** Basics of a Computer
- 1.3** Latest Computers
- 1.4** Introduction to C
- 1.5** About ANSI C Standard
- 1.6** Machine, Assembly and High-Level Language
- 1.7** Assembler, Compiler and Interpreter
- 1.8** Structure of a C Program
- 1.9** Programming Rules
- 1.10** Executing the C Program
- 1.11** Standard Directories
- 1.12** The First C Program
- 1.13** Advantages of C
- 1.14** Header Files
- 1.15** Algorithm
- 1.16** Classification of Algorithms
- 1.17** Flowcharts
- 1.18** Pseudocode

1.1 WHY TO USE COMPUTERS?

Computers play a vital role in the socioeconomic progress of a country. Every nation is paying much importance to computer literacy. Progress of individuals, surrounding region, nation and of the world is ensured only with the introduction and use of computers. In every walk of life, computers are being used increasingly. In every part of the world, computers are employed to increase the overall productivity. Moreover, the quality of the products due to application of computers is substantially improved. The impact of computers is very high on mass education, entertainment and productivity in all fields. With the use of computers the cost of production reduces drastically, a lot of time is saved and the best quality is ensured.

With the adoption of the policy of liberalization, privatization and globalization by the governments all over the world, it is necessary to become competitive in the market. Luxurious survival is not easy as was possible in ancient days. With land, labour, muscle power and capital, economy can certainly be boosted to a certain degree, but the intellectual power is highly superior to all these assets. With some know-how (for the common users) and a lot of technical innovations (by the developers), numerical problems, business transactions and scientific applications, can be computed in almost no time. A computer does various tasks on the basis of programs. Intellectual computer is certainly a powerful gadget but human intellectual capability is much higher than that of a computer. Thus, computer does not have its own brain. Brainpower of computer is limited. Thus, human intellectual power is further reinforced by the use of computers frequently in the day-to-day life. By learning programming languages, developing software packages and using hardware of computers, one can make his/her life and nation prosperous.

1.2 BASICS OF A COMPUTER

Data to be processed by a computer appears in different forms, such as numeric, alphabetical characters either in uppercase or in lowercase, special characters and multimedia. Research applications in universities and colleges frequently use numeric data on large scales, whereas businessmen and people in the corporate sector use both numeric and character data. However, in animation multimedia is used which include text, video and audio.

A computer is a programmable electronic machine that accepts instructions and data through input devices, manipulating data according to instructions and finally providing result to the output device. The result can be stored in memory or sent to the output device. Figure 1.1 shows the conventional block diagram of a conventional computer.

Figure 1.2 shows a computer system containing a monitor, a keyboard and a rectangular box comprising CPU on the motherboard.

In brief, the various blocks of a computer are described as follows.

Input Device: Input device is used to accept the data and instructions in to the computer. Through the input device, *i.e. keyboard*, instructions and data are stored in a computer's memory. Stored instructions are further read by the computer from its memory and thus a computer manipulates the data according to the instructions. The various input devices that can be used in a computer are

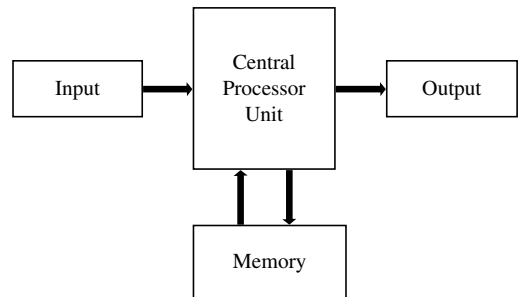


Figure 1.1 Block diagram of a conventional computer

keyboards, mouse, analog-to-digital converters, light pen, track ball, optical character reader and scanner. Figure 1.3 shows the input devices of a computer. Floppy, compact disc, etc. can be used as input or output device.

Central Processor Unit: CPU is the abbreviation for central processor unit. It is the heart of the computer system. It is a hardware device in the

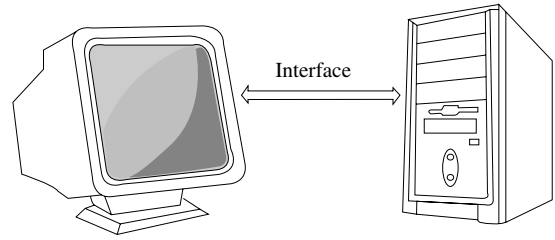


Figure 1.2 computer system

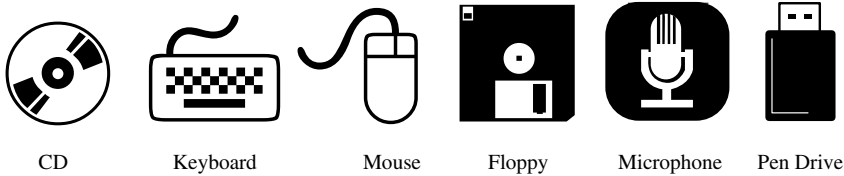


Figure 1.3 Input devices of a computer

computer and quite often it called as microprocessor chip. Since it is a tiny chip hence called as microprocessor chip. This chip is produced from silicon vapor over which millions of transistors are mounted with modern fabrication techniques.

The brain of the computer is CPU. This chip is responsible to interpret and execute the instructions. It comprises arithmetic and logical unit, registers and control unit. The arithmetic and logical unit performs various arithmetic and logical operations on the data based upon the instructions. The control unit generates the timing and control signals for carrying out operations within the processor. Registers are used for holding the instructions and storing the results temporarily. Instructions are stored in the memory and they are fetched one by one and executed by the processor.

Output Device: The output device is used to display the results on the screen or to send the data to an output device. The processed data is ultimately sent to the output device by the computer. The output device can be a monitor, a printer, an LED, seven-segment display, D to A converter, plotter, and so on. Figure 1.4 shows different output devices of a computer.

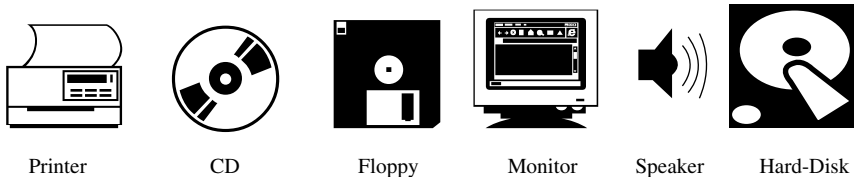


Figure 1.4 Output devices of a computer

Memory: Memory is used to store the program. There are two types of semi-conductor memories. They are as follows:

- (i) RAM (Random access memory)
- (ii) ROM (Read only memory)

semi-conductor memory is used for storing the instructions and data in the form of ones and zeros. The memory can be called user's memory or read-write memory. Processor first reads the instructions and data from the primary memory (semi-conductor memory). Then, it executes the instructions.

One more memory device used by a computer is called read only memory (ROM). This contains a fixed software program for providing certain operations. This is non-volatile memory. Its contents cannot be eliminated when power supply goes off. The basic input-output system (BIOS) is a software used to control various peripheral devices such as a keyboard, a monitor, a printer, a mouse, ports including serial and parallel ports. In fact, this is an operating system. It is possible to access the users' written programs, i.e. by loading a file, saving it and doing modifications to it in the later stage. As soon as a personal computer is switched on, the software gets booted from ROM. Thus, various functions are assigned to all supporting peripherals of a central processing unit (CPU) and easy interactions are provided to the user by BIOS while booting the system.

For storing volumous data, secondary storage devices can be used. They are optical disk, magnetic disc, tapes, etc.

1.3 LATEST COMPUTERS

The discussion on the various types of computers existing in the market is beyond the scope in this book. Latest computers available in the market are in various shapes and sizes. These computers are just like notebooks or even require lesser space than the notebook. They are handy, light weighted and portable. Because of these reasons many computer users now carry laptops while travelling for their day-to-day work. In the next couple of years, it is predicted that every school-going child would be using a laptop or palmtop, and these devices would be sold just like hot cakes in the market with affordable prizes. A lot of research work is going on in the foreign and Indian Universities, institutes and industries on the reduction of the size of a computer, building more functions in it with low cost, minimum battery power for operating it for large duration in the absence of A.C. mains power supply. In a couple of years, laptops and palmtops would be flooded in the market. These notebook-shaped laptops are as powerful as desktop computers. In other words, the computing power of a laptop would be advancing to that of desktop computers and minicomputers. They are used for the purpose of text processing, web surfing, multimedia operation, image processing and so on. Performances of these computers are constantly improving, and the cost is drastically reducing. A wireless network can be formed easily with laptops with the Bluetooth technology. Wifi wireless network is quite popular nowadays, and there we use these computers quite often.

Typical specifications of a laptop computer are as follows,

- Processor: Intel Core i7 (8M cache,2.80GHz) OR AMD
- RAM: Minimum 2 GB to Maximum 16 GB
- Hard disk: Minimum 320 GB up to 1 TB or higher
- CD/DVD Drive: Dual Layer DVD+/-RW Drive, SuperDrive, Bluetooth and Camera
- Screen Size: 15.6" up to 23" Flat screen with LCD/LED display
- Ethernet: 10/100/1000 Mbps Ethernet
- Port: USB 3.0
- Graphics Card: From 520 MB or higher
- Rechargeable Battery back up
- Operating system such as windows XP/ windows 7/ Linux

For simpler application, one can go for Celeron-based laptop where heat generated by the processor is little. Pentium processors are used only for dedicated applications. Heat generated by this kind of laptop is more and battery is to be frequently charged (Figure 1.5).



Figure 1.5 The laptop

1.4 INTRODUCTION TO C

C is one of the most popular general-purpose programming languages. C language has been designed and developed by Dennis Ritchie at Bell Laboratories, USA, in 1972. Several important concepts of C are drawn from ‘Basic combined programming language’ and ‘B’ language. Figure 1.6 shows the development of C language from the two languages. Martin Richards developed BCPL in 1967. The impact of BCPL on C is observed indirectly through the language B, which was developed by Ken Thompson in 1970. C is also called an offspring of the BCPL. Table 1.1 illustrates the evolution of languages and their inventors of programming language.

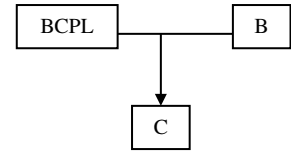


Figure 1.6 Evolution of C

Table 1.1 Languages and their inventors

Sr. No.	Language	Inventor	Year
1.	BCPL	Martin Richards	1967
2.	B	Ken Thompson	1970
3.	C	Dennis Ritchie	1972

The C language is not tied to any particular operating system. It can be used to develop new operating systems. Refer to Figure 1.7 in which the C language is shown associated with the various operating systems. The C language is closely associated with the UNIX operating system. The source code for the UNIX operating system is in C. C runs under a number of operating systems including MS-DOS. The C programs are efficient, fast and highly portable, i.e. C programs written on one computer can be run on another with mere or almost no modification.

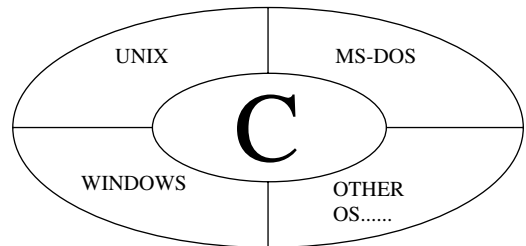


Figure 1.7 C and operating systems

The C programming language contains modules called functions. The C functions are the basic building blocks and the most programmers take its benefit. Programmers include these functions in their program from the C standard library.

C language is a middle-level computer language. It does not mean that C is not powerful and rugged for writing programs like in Fortran and Pascal. It also does not mean that it is troublesome like assembly level languages. It combines the features of a high-level language and functionality like assembly languages. In C, one can develop a program fast and execute fast. It reduces the gap between high- and low-level languages; that is why it is known as a middle level language. It is well suited for writing both application and system softwares.

C is a structural language. It has many similarities with the other structural languages such as Pascal and Fortran. Structured language facilitates the development of a variety of programs in small modules or blocks. Lengthy programs can be divided into shorter programs. The user requires thinking of a problem in terms of functional blocks. With appropriate collection of different modules, the programmer can make a complete program.

It is easy for writing, testing, debugging and maintenance with structured programming. In case some changes are to be done in a program, the programmer can refer to the earlier written module with editor and incorporate the appropriate changes. Hence, structured languages are easier and most of the developers prefer these languages, than the non-structured languages like BASIC and COBOL. Programming with non-structured languages is tough in comparison to structured languages.

C is also called a system-programming language because it is greatly helpful for writing operating systems, interpreters, editors, compilers, database programs and network drivers.

BCPL and B are data type-less languages. However, C language has a variety of data types. The standard data types in C are integers, floating point, characters. Also, derived data types can be created such as pointers, arrays, structures and unions. Expressions are built from operands and operators. Any expression or an assignment or a call to function can be a statement. The pointers provide machine-independent address arithmetic.

C also provides control-flow statements such as decision-making statements (`if-else`) and (`switch-case`) multi-choice statement. C supports `for`, `while` and `do-while` looping statements.

C does not have any operator to perform operation on composite object. There does not exist any function or operator that handles entire array or string. For example, to assign elements of one array to another array simply with single assignment statement is not enough, but an element-to-element assignment is to be done. However, structure objects can be copied as a unit.

C is not a strongly typed language. But typed statements are checked thoroughly by C compilers. The compiler will issue errors and warning messages when syntax rules are violated. There is no automatic conversion of incompatible data types. A programmer has to perform explicit type conversion.

UNIX: The UNIX is an interactive operating system. It is useful in microcomputers, minicomputers and main frame computers. This operating system is very portable and supports multi-user processing, multi-tasking and networking. Several users can use UNIX at once for performing the same task. This operating system was developed to connect various machines together. UNIX is primarily used for workstations and minicomputers.

1.5 ABOUT ANSI C STANDARD

For many years, there was no standard version of C language, and the definition provided in reference manual was followed. Due to this reason portability feature of C language was not provided from one computer to another. To overcome this discrepancy, a committee was set up in the summer of 1983 to create a standard C version which is popularly known as American National Standard Institute (ANSI) standard.

This committee had defined once and for all a standard C language. The standardization process took about six years for defining the C language. The ANSI C standard was adopted in 1989, and its first copy of C language was introduced in the market in 1990.

Thus, ANSI C is internationally recognized as a standard C language. The purpose of this standard is to enhance the portability and efficient execution of C language programs on different computers. Today, all compilers of C support the ANSI standard. In other words, almost all C compilers available in the market have been designed to follow ANSI C standard. ANSI C and other C compilers such as turbo-C version 2 support programs developed in this book.

1.6 MACHINE, ASSEMBLY AND HIGH-LEVEL LANGUAGE

It is a computer's natural language, which can be directly understood by the system. This language is machine dependent, i.e. it is not portable. A program written in 1's and 0's is called a machine language. A binary code is used in a machine language for a specific operation. A set of instructions in binary pattern is associated with each computer. It is difficult to communicate with a computer in terms of 1s and 0s. Hence, writing a program with a machine language is very difficult. Moreover, speed of writing, testing and debugging is very slow in machine language. Chances of making careless errors are bound to be there in this language. The machine language is defined by the hardware design of that hardware platform. Machine languages are tedious and time consuming.

1.6.1 | Assembly Language

Instead of using a string of binary bits in a machine language, programmers started using English-like words as commands that can be easily interpreted by programmers. In other words, the computer manufacturers started providing English-like words abbreviated as mnemonics that are similar to binary instructions in machine languages. The program is in alphanumeric symbols instead of 1s and 0s. The designer chooses easy symbols that are to be remembered by the programmer, so that the programmer can easily develop the program in assembly language. The alphanumeric symbols are called mnemonics in the assembly language. The ADD, SUB, MUL, DIV, RLC and RAL are some symbols called mnemonics.

The programs written in other than the machine language need to be converted to the machine language. Translators are needed for conversion from one language to another. Assemblers are used to convert assembly language program to machine language. Every processor has its own assembly language. For example, 8085 CPU has its own assembly language. CPUs such as 8086, 80186, 80286 have their own assembly languages.

Following disadvantages are observed with the assembly languages:

- (i) It is time consuming for an assembler to write and then test the program.
- (ii) Assembly language programs are not portable.
- (iii) It is necessary to remember the registers of CPU and mnemonic instructions by the programmer.
- (iv) Several mnemonic instructions are needed to write in assembly language than a single line in high-level language. Thus, assembly language programs are longer than the high language programs.

1.6.2 | High-Level Language

Procedure-oriented languages are high-level languages. These languages are employed for easy and speedy development of a program. The disadvantages observed with assembly languages are overcome by high-level languages. The programmer does not need to remember the architecture and registers of a CPU for developing a program. The compilers are used to translate high-level language program to machine language. Examples of HLL are COBOL, FORTRAN, BASIC, C and C++. The following advantages are observed with HLL languages:

- (i) Fast program development.
- (ii) Testing and debugging a program is easier than in the assembly language.
- (iii) Portability of a program from one machine to other.

1.7 ASSEMBLER, COMPILER AND INTERPRETER

A program is a set of instructions for performing a particular task. These instructions are just like English words. The computer interprets the instructions as 1's and 0's. A program can be written in assembly language as well as in high-level language. This written program is called the source program. The source program is to be converted to the machine language, which is called an object program. A translator is required for such a translation.

Program translator translates source code of programming language into machine language-instruction code. Generally, computer programs are written in languages like COBOL, C, BASIC and ASSEMBLY LANGUAGE, which should be translated into machine language before execution.

Programming language translators are classified as follows. A list of translators is given in Figure 1.8.

Translators are as follows:

- (i) Assembler
- (ii) Compiler
- (iii) Interpreter

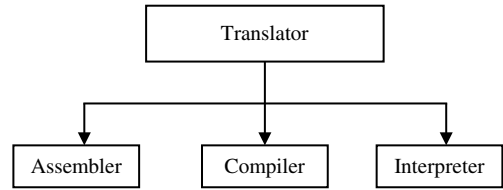


Figure 1.8 Translators

Assembler: An assembler translates the symbolic codes of programs of an assembly language into machine language instructions (Figure 1.9). The symbolic language is translated to the machine code in the ratio of one is to one symbolic instructions to one machine code instructions. Such types of

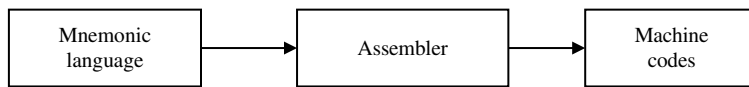


Figure 1.9 Assembler

languages are called low-level languages. The assembler programs translate the low-level language to the machine code. The translation job is performed either manually or with a program called assembler. In hand assembly, the programmer uses the set of instructions supplied by the manufacturer. In this case, the hexadecimal code for the mnemonic instruction is searched from the code sheet. This procedure is tedious and time-consuming. Alternate solution to this is the use of assemblers. The program called assembler provides the codes of the mnemonics. This process is fast and facilitates the user in developing the program speedily.

Compiler: Compilers are the translators, which translate all the instructions of the program into machine codes, which can be used again and again (see Figure 1.10). The program, which is to be translated, is called the source program and after translation the object code is generated.

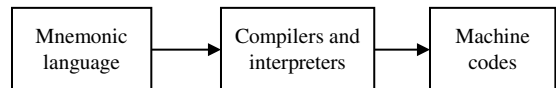


Figure 1.10 Compiler/interpreter

The source program is input to the compiler. The object code is output for the secondary storage device. The entire program will be read by the compiler first and generates the object code. However, in interpreter each line is executed and object code is provided. M-BASIC is an example of an interpreter. High-level languages such as C, C++ and Java compilers are employed. The compiler displays the list of errors and warnings for the statements violating the syntax rules of the language. Compilers also have the ability of linking subroutines of the program.

Interpreter: Interpreters also come in the group of translators. It helps the user to execute the source program with a few differences as compared to compilers. The source program is just like English statements in both interpreters and compilers. The interpreter generates object codes for the source program. Interpreter reads the program line by line, whereas in compiler the entire program is read by the compiler, which then generates the object codes. Interpreter directly executes the program from its source code. Due to this, every time the source code should be inputted to the interpreter. In other words, each line is converted into the object codes. It takes very less time for execution because no intermediate object code is generated.

Linking: C language provides a very large library, which contains numerous functions. In some applications of C the library may be a very large file. Linker is a program that combines source code and codes from the library. Linking is the process of bringing together source program and library code.

The library functions are relocatable. The addresses of various machine codes are defined absolutely and only the offset information is kept. When the source program links with the standard library functions, offset of the memory addresses is used to create the actual address.

1.8 STRUCTURE OF A C PROGRAM

Every C program contains a number of building blocks known as functions. Each function of it performs a specific task independently. A C program comprises different sections shown in Figure 1.11.

- (i) **Include Header File Section:** C program depends upon some header files for function definition that are used in the program. Each header file has extension `'.h'`. The header files are included at the beginning of the program in the C language. These files should be included using `#include` directive as given below.

Example:

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

In this example, `<stdio.h>` file is included, i.e. all the definitions and prototypes of function defined in this file are available in the current program. This file is also compiled with the original program. The programmer can include the appropriate header files while executing solved or unsolved programming examples given in this book.

- (ii) **Global Declaration:** This section declares some variables that are used in more than one function. These variables are known as global variables. This section must be declared outside of all the functions.
- (iii) **Function `main()`:** Every program written in C must contain `main()` and its execution starts at the beginning of this function. In ASCII C standard, first line of C program from where program execution begins is written as follows.

```
int main(void)
```

This is the function definition for `main()`. Parenthesis followed to `main` is to tell the user again that `main()` is a function. The `int main(void)` is a function that takes no arguments and returns a value of type `int`. Here in this line, `int` and `void` are keywords and they have special meanings assigned by the compiler. In case `int` is not mentioned in the above statement, by default the function returns an integer.

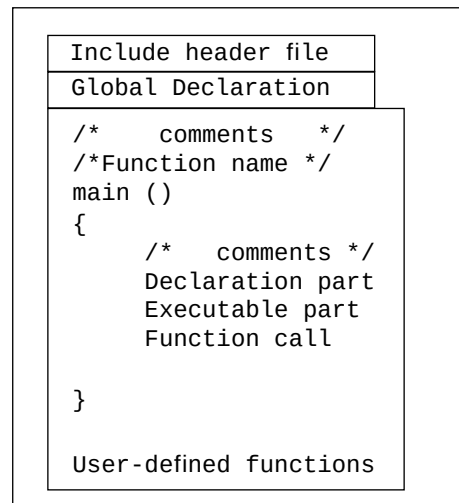


Figure 1.11 Structure of a C program

Alternately, one can also write the first line of C program from where program execution begins is as follows.

```
void main(void)
```

Here, this function takes no arguments and returns nothing. Alternately, one can also write the same function as follows.

```
void main(): This functions returns nothing and takes no arguments.
```

In all chapters, in maximum programming examples the `main` function is written as `void main()`. This procedure is followed in this book only to avoid writing `return` statement at the end of each program. This step helps to minimize source code lines. At few places in this book, `main` function is initialized with `int main(void)`. In such a case, `return` statement is used at the end of program (before closing brace). The programmer can either write the function `main` with `int main(void)` or `void main()`. Only in the formal case, `return` statement should be used before the end of function for terminating the execution of the `main` function.

The program contains statements that are enclosed within the braces. The opening brace (`{`) and closing brace (`}`) are used in C. Between these two braces, the program should declare declaration and executable part. The opening curly brace specifies the start of the definition of the `main` function. The closing curly brace specifies the end of the code for the `main` function.

- (iv) **Declaration Part:** The declaration part declares the entire local variables that are used in executable part. Local variable scope is limited to that function where the local variables are declared. The initializations of variables can also be done in this section. The initialization means providing initial value to the variables.
- (v) **Executable Part:** This part contains the statements following the declaration of the variables. This part contains a set of statements or a single statement.
- (vi) **Function Call:** From the `main()` a user defined function can be invoked by the user as per need/application.
- (vii) **User-defined Function:** The functions defined by the user are called user-defined functions. These functions are defined outside the `main()` function.
- (viii) **Body of the Function:** The statements enclosed within the body of the function (between opening and closing brace) are called body of the function.
- (ix) **Comments:** Comments are not necessary in a program. However, to understand the flow of programs a programmer can insert comments in the program. Comments are to be inserted by the programmer. It is useful for documentation. The clarity of the program can be followed if it is properly documented.

Comments are statements that give us information about the program which are to be placed between the delimiters `/*` and `*/`. The programmers in the programs for enhancing the lucidity frequently use comments. The compiler does not execute comments. Thus, we can say that comments are not a part of executable programs.

A user can frequently use any number of comments that can be placed anywhere in a program. Please note that comment statements can be nested. The user should select the **OPTION MENU** of the editor and select the **COMPILER-SOURCE - NESTED COMMENTS ON/OFF**. The comments can be inserted with single statement or in nested statements.

Example:

```
/* This is single comment */
/* This is an example of /* nested comments */ */
/* This is an example of
    of comments in
    multiple lines */ /* It can be nested */
```

1.9 PROGRAMMING RULES

A programmer while writing a program should follow the following rules:

- (i) Every program should have `main()` function.
- (ii) C statements should be terminated by a semi-colon. At some places, a comma operator is permitted. If only a semi-colon is placed it is treated as a statement. For example:

```
while(condition)
;
```

The above statement generates infinite loop. Without semi-colon the loop will not execute.

- (iii) An unessential semi-colon if placed by the programmer is treated as an empty statement.
- (iv) All statements should be written in lowercase letters. Generally, uppercase letters are used only for symbolic constants.
- (v) Blank spaces may be inserted between the words. This leads to improvement in the readability of the statements. However, this is not applicable while declaring a variable, keyword, constant and function.
- (vi) It is not necessary to fix the position of statement in the program; i.e. a programmer can write the statement anywhere between the two braces following the declaration part. The user can also write one or more statements in one line separating them with a semi-colon (;). Hence, it is often called a free-form language. The following statements are valid:

```
a=b+c;
d=b*c;
or
a=b+c; d=b*c;
```

- (vii) The opening and closing braces should be balanced, i.e. if opening braces are four; closing braces should also be four.

1.10 EXECUTING THE C PROGRAM

A C program must go through various phases such as creating a program with editor, execution of preprocessor program, compilation, linking, loading and executing the program. The following steps are essential in C when a program is to be executed in MS-DOS mode:

- (i) **Creation of a Program:** The program should be written in C editor. The file name does not necessarily include extension '.C'. The default extension is '.C'. The user can also specify his/her own extension. The C program includes preprocessor directives.
- (ii) **Execution of a Preprocessor Program:** After writing a program with C editor the programmer has to compile the program with the command (Alt-C). The preprocessor program executes

first automatically before the compilation of the program. A programmer can include other files in the current file. Inclusion of other files is done initially in the preprocessor section.

- (iii) **Compilation and Linking of a Program:** The source program contains statements that are to be translated into object codes. These object codes are suitable for execution by the computer. If a program contains errors the programmer should correct them. If there is no error in the program, compilation proceeds and translated program is stored in another file with the same file name with extension '`.obj`'. This object file is stored on the secondary storage device such as a disc.

Linking is also an essential process. It puts together all other program files and functions that are required by the program. For example, if the programmer is using `pow()` function, then the object code of this function should be brought from `math.h` library of the system and linked to the `main()` program. After linking, the program is stored on the disc.

- (iv) **Executing the Program:** After the compilation the executable object code will be loaded in the computer's main memory and the program is executed. The loader performs this function. All the above steps/phases of C program can be performed using menu options of the editor.

As shown in Figure 1.12, pre-processor directories/program is executed before compilation of the main program. The compiler checks the program and if any syntax error is found, the same

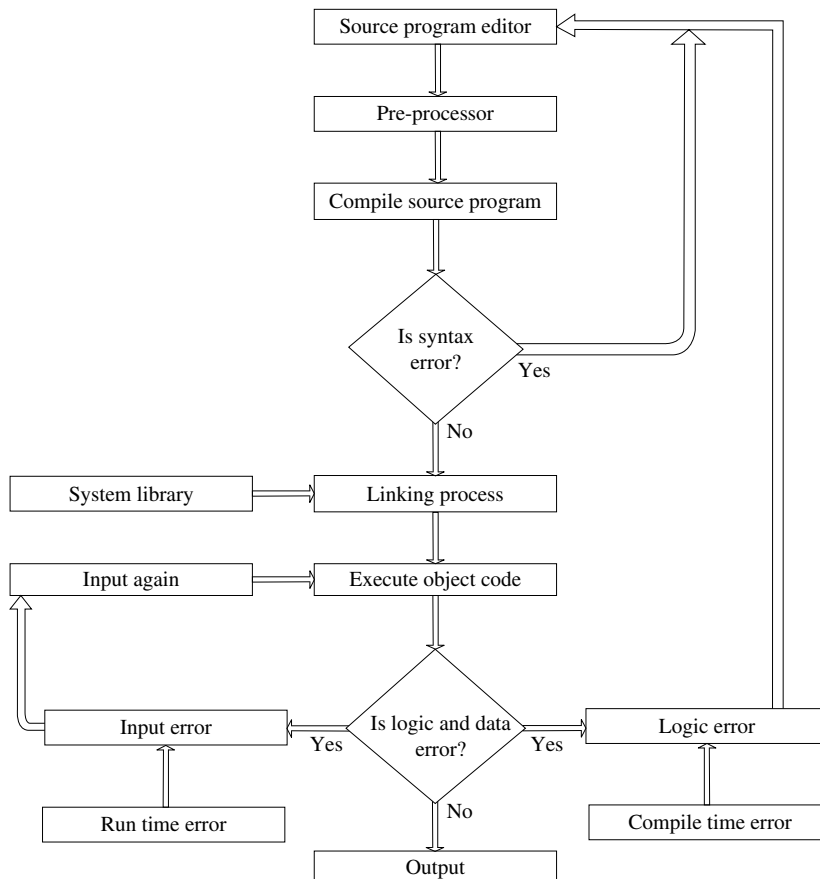


Figure 1.12 Flow chart of a program in C

is displayed. The user is again forced to go to edit window. After removing an error, the compiler compiles the program. Here, at this stage object code is generated. During the program execution, if user makes mistakes in inputting data, the result would not be appropriate. Therefore, the user again has to enter the data. The output is generated when a program is error free.

Editing, compiling and executing a program file with an editor (Figure 1.13):

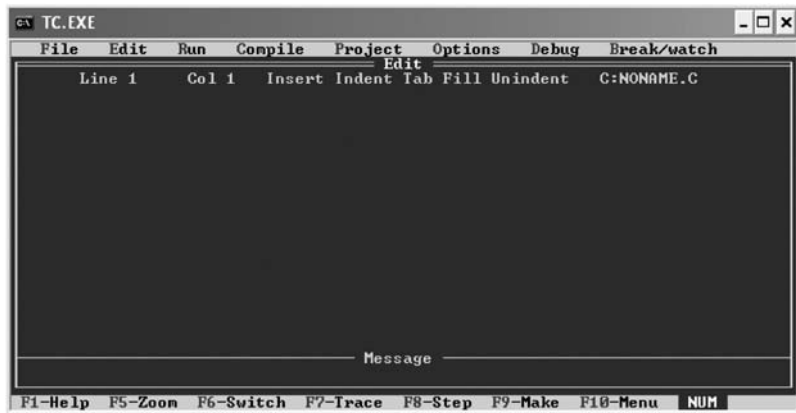


Figure 1.13 Window to Turbo C editor

The programmer can use the Turbo C editor/compiler. The Turbo C editor is a software, in which the programmer can write the program in C source code. The window to Turbo C editor appears when we invoke *shortcut to TC* icon, which can be placed on desktop. This window is used for editing, compiling and running the program.

Its menu bar comprises eight menus: File, Edit, Run, Compile, Project, Options, Debug and Break/watch.

File: This menu is used for creating a new program file, loading an existing C file, writing the file with appropriate path, invoking DOS (OS Shell), changing directory and quitting the program.

Edit: The Edit menu provides editing options.

Run: The Run menu provides options such as Run, Program reset, Go to cursor, Trace into, Step over and User screen.

On pressing Alt+F keys, one can go to the File menu and select either *New* for creating a new file or in case file is already existing then use *Load* option and load the file by giving the appropriate path. Extension of the file with .c is automatically provided by the editor. The programmer can put extension .c or by default .c is provided by the editor. The programmer can now write a program with C syntax.

It is better to create and save programs in a separate folder/subdirectory in the home directory of the disk. The folder/subdirectory is the working directory. This is due to the association of the program file with several files created during compiling and running. Files created are your own files (source code file and data files) and besides, some other files created are after compilation and running the program.

For example if C:\Turboc2 is home directory, a subdirectory can be created with the Command prompt. Assume that the created subdirectory is Vishal, then working directory path would be C:\Turboc2\Vishal. So, create and save all programs in the working directory Vishal (Figure 1.14).



Figure 1.14 Opening a new file

After editing the program file, the same should be saved either with F2 key or Alt+F & Save from *File* menu and compiled from compile menu. Now programmer can use the keys Alt+C in the editor as shown below (Figure 1.15).



Figure 1.15 Compilation of program

On compilation, the same program is to be run with Alt+R keys. The screen appears as shown below. Of course, this step is adapted when there are no errors after compilation. After running the program, answer appears on the other screen. The programmer can try a simple program as cited below in the window. Executable file is created after running the program file. The .exe file created can be observed (Figure 1.16).



Figure 1.16 Running a program

It is expected to edit, compile and run a program given in the above snapshot by the user in the C editor.

1.11 STANDARD DIRECTORIES

The turbo-C has three standard directories; they are `include`, `sys` and `lib`. The `Sys` is the sub-directory of `include`. The `include` directory contains all the header files and `lib` contains all the library files. Before executing the program the path setting should be done. In turbo-c edit, select Option menu Directories option. Here, set the path of `include`, `library` and `turbo-c-directories`.

1.12 THE FIRST C PROGRAM

- 1.1 Write a program to display message “Hello! C Programmers”.

```
void main()
{
    printf("Hello! C Programmers");
}
```

OUTPUT:

Hello! C Programmers

Explanation:

This program displays the message ‘Hello! C Programmers’ using the `printf()` statement. Follow the following steps to execute the program through Turbo-c editor.

After typing a program by pressing F2, a program can be saved. If you are saving for the first time, a name will be asked. Type the file and the extension (.C) will automatically be added.

By pressing ALT+C you can reach the compile option. Also by pressing F9, you can compile the program. To execute the program you can press CTRL + F9. To see the output of the program, press ALT+F5. User can make `exe` file by pressing F9 key twice.

- 1.2 Write a program to know about the use of comments (how to use comments?).

```
void main()
{
    clrscr(); /* clears the screen */
    printf("\nThis program explains comments");
    /* How to use comment? */
}
```

OUTPUT:

This program explains comments

Explanation:

In the above program, we can observe how comments are inserted in a program. The comments are not an executable part. It is only useful for the programmer to understand the flow of a program. This program prints the message as shown in the output. The function `clrscr()` clears the screen, defined in header file `<conio.h>`. Although the file is not included, some compilers allow execution and some will flag an error message.

- 1.3 Write a program to return the value from `main()`.

```
main()
{
    return 0;
}
```

Explanation:

The above program produces no output. The `main()` should return a value of either 0 or 1. Some operating systems check the return value of `main()`. If `main()` returns 0, i.e. program executed successfully; else for other value OS assumes the opposite. If user fails to put the return statement, the compiler would not complain.

1.13 ADVANTAGES OF C

- (i) It contains a powerful data definition. The data type supported are characters, alphanumeric, integers, long integer, floats and double. It also supports string manipulation in the form of character array.
- (ii) C supports a powerful set of operators.
- (iii) It also supports powerful graphics programming and directly operates with hardware. Execution of program is faster.
- (iv) An assembly code is also inserted into C programs.
- (v) C programs are highly portable on any type of OS platforms.
- (vi) System programs such as compilers, operating systems can be developed in C. For example, the popular operating system UNIX is developed in C.
- (vii) The C language has 32 keywords and about 145 library functions and near about 30 header files.
- (viii) C works closely with machines and matches assembly language in many ways.

1.14 HEADER FILES

stdio.h: Standard input and output files. All formatted and unformatted functions include file operation functions defined in this file. The most useful formatted `printf()` and `scanf()` are defined in this file. This file must be included at the top of the program. Most useful functions from this header files are `printf()`, `scanf()`, `getchar()`, `gets()`, `putc()` and `putchar()`.

conio.h: Console input and output. This file contains input and output functions along with a few graphic-supporting functions. The `getch()`, `getche()` and `clrscr()` functions are defined in this file.

math.h: This file contains all mathematical and other useful functions. The commonly useful functions from this files are `floor()`, `abs()`, `ceil()`, `pow()`, `sin()`, `cos()` and `tan()`. The list of commonly used header files are given in Table 1.2

Table 1.2 Commonly useful header files

<i>Sr. No.</i>	<i>Header File</i>	<i>Functions</i>	<i>Function Examples</i>
1.	<code>stdio.h</code>	Input, output and file operation functions	<code>printf()</code> , <code>scanf()</code>
2.	<code>conio.h</code>	Console input and output functions	<code>clrscr()</code> , <code>getche()</code>
3.	<code>alloc.h</code>	Memory allocation-related functions	<code>malloc()</code> , <code>realloc()</code>
4.	<code>graphics.h</code>	All graphic-related functions	<code>circle()</code> , <code>bar3d()</code>
5.	<code>math.h</code>	Mathematical functions	<code>abs()</code> , <code>sqrt()</code>
6.	<code>string.h</code>	String manipulation functions	<code>strcpy()</code> , <code>strcat()</code>
7.	<code>bios.h</code>	BIOS accessing functions	<code>biosdisk()</code>
8.	<code>assert.h</code>	Contains macros and definitions	<code>void assert(int)</code>
9.	<code>ctype.h</code>	Contains prototype of functions which test characters	<code>isalpha(char)</code>
10.	<code>time.h</code>	Contains date- and time-related functions	<code>asctime()</code>

There are different ways of representing the logical steps for finding a solution of a given problem. They are as follows:

- (i) Algorithm
- (ii) Flowchart
- (iii) Pseudo-code

In the algorithm, a description of the steps for solving a given problem is provided. Here, stress is given on the text. Flowchart represents the solution of a given problem graphically. Pictorial representation of the logical steps is a flowchart. Another way to express the solution of a given problem is by means of a pseudo-code.

1.15 ALGORITHM

Algorithm is a very popular technique used to obtain a solution for a given problem. The algorithm is defined as ‘the finite set of steps, which provide a chain of actions for solving a definite nature of problem’. Each step in the algorithm should be well defined. This step will enable the reader to translate each step into a program. Gradual procedure for solving a problem is illustrated in this section.

An algorithm is a well-organized, pre-arranged and defined textual computational module that receives some value or set of values as input and provides a single value or a set of values as output. These well-defined computational steps are arranged in a sequence, which processes the given input into an output. Writing precise description of the algorithm using an easy language is most essential for understanding the algorithm. An algorithm is said to be accurate and truthful only when it provides the exact required output. Lengthy procedure is sub-divided into small parts and thus steps are made easy to solve a given problem. Every step is known as an instruction.

In our daily life, we come across numerous algorithms for solving problems. We perform several routine tasks, for example riding a bike, lifting a phone, making a telephone call, switching on a television set and so on.

For example, to establish a telephonic communication between two subscribers, following steps are to be followed:

- (i) Dial a phone number
- (ii) Phone rings at the called party
- (iii) Caller waits for the response
- (iv) Called party picks up the phone

- (v) Conversation begins between them
- (vi) After the conversation, both disconnect the call

Another real life example is to access Internet through Internet service provider with dial up facility. To log on to the Internet, the following steps are to be followed:

- (i) Choose the Internet service provider for accessing the Internet.
- (ii) Obtain from service provider a dial up number.
- (iii) Acquire IP address of the service provider.
- (iv) Acquire login ID and password.
- (v) Run the Internet browsing software.

When one writes an algorithm, it is essential to know how to analyse the algorithm. Analysing algorithm refers to calculating or guessing resources needed for an algorithm. Resources mean computer memory, processing time, logic gates. In all the above factors, time is the most important factor because the program developed should be faster in processing. The analysis can also be made by reading the algorithm for logical accuracy, tracing the algorithm, implementing it, and checking with some data and with mathematical techniques to confirm its accuracy.

Algorithms can also be expressed in a simple method. It will help the user to put into operation easily. However, this approach has a few drawbacks. It requires more space and time. It is very essential to consider the factors such as time and space of an algorithm. With minimum resources of system such as CPU's time and memory, an efficient algorithm must be developed.

Practically, it is not possible to do a simple analysis of an algorithm to conclude the execution time of an algorithm. The execution time is dependent upon the machine and the way of implementation. The timing analysis depends upon the input required. To accurately carry out the time analysis, it is also very essential to know the exact directives executed by the hardware and the execution time passed for each statement.

➤ 1.4 Write a program to swap two numbers using a third variable.

Algorithm for swapping two numbers:

STEP 1: Start.

STEP 2: Declare variables a, b, & c.

STEP 3: Read values of a & b.

STEP 4: Copy value of a to c; b to a; & c to b.

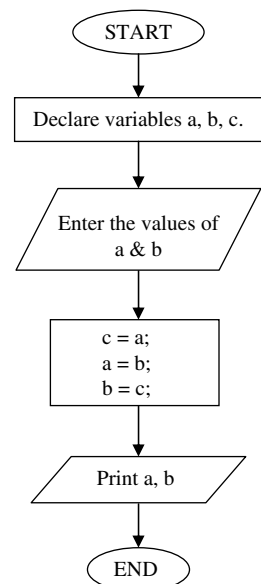
STEP 5: Print swapped values of a & b.

STEP 6: Exit.

Program:

```
void main()
{
    int a,b,c;
    clrscr();
    printf("\nEnter two numbers for A & B:");
    scanf("%d %d",&a,&b);
    c=a;
    a=b;
    b=c;
    printf("\nAfter swaping the values
are:A=%d & B=%d.",a,b);
    getch();
}
```

Flowchart for swapping two numbers:



Hint: Copy value of a in third variable c, value of b in a, and value of c in b.

Explanation:

In the above cited program, first three variables a, b, & c are declared. The program invokes two values of a & b from user. The user assigns values to the variables a & b.

Now copy the value of a in third variable c, value of b in a, and value of c in b.

Here, when we assign the value of a to c ($c = a$), the value of a is copied in variable c. When we assign ($a = b$), the value of b is copied in variable a and the previous value is replaced. Now, b holds the value of variable c in statement ($b = c$).

The `printf( )` statement prints the values of a & b.

- 1.5 Write a program to swap two numbers without using a third variable.

Algorithm:

STEP 1: Start.

STEP 2: Declare two variables a & b.

STEP 3: Addition of a & b and place result in a.

STEP 4: Subtraction of a & b and place result in b.

STEP 5: Subtraction of a & b and place result in a.

STEP 6: Print values of a & b.

STEP 7: End.

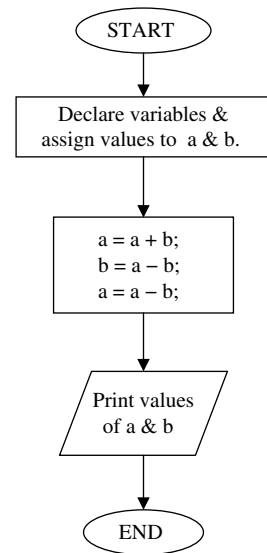
Program:

```
void main()
{
    int a=4,b=6;
    clrscr();
    a=a+b;
    b=a-b;
    a=a-b;
    printf("\nthe value of a=%d & b=%d",a,b);
    getch();
}
```

OUTPUT:

The value of a=6 & b=4

Flowchart:



Explanation:

In the above cited program, first two variables a & b are declared and initialized.

Values of a & b are added in the statement $a = a + b$. In the next statement, b is subtracted from a and result is stored in b. Finally, the value of a is obtained by subtracting b from a.

The `printf( )` statement prints the values of a & b.

1.15.1 | Program Design

When you have thoroughly studied the program and examined the requirement of the program, the next step is to design the program. To make the programs simple, divide the program into smaller modules. This makes thinking process easy and you can separately apply logic on each portion. After dividing the program, according to priority and importance write the code.

Coding Programs: Coding a program is the second step. Once you have understood the program, now you can implement through the code. If a program is short, start coding from the beginning to top in a sequence. Identify the different variables and selection or control structures required. Also write comments so that you can follow them in future. While coding, appropriate messages for user's direction should be prompted.

Testing Programs: After completing the coding of a program, the next step is to test the program. Confirm that the required source files and data files are at the specified location in the system.

1.16 CLASSIFICATION OF ALGORITHMS

The classification of algorithm is based on repetitive steps and on control transfer from one statement to another. Figure 1.17 shows the classification of Algorithms.

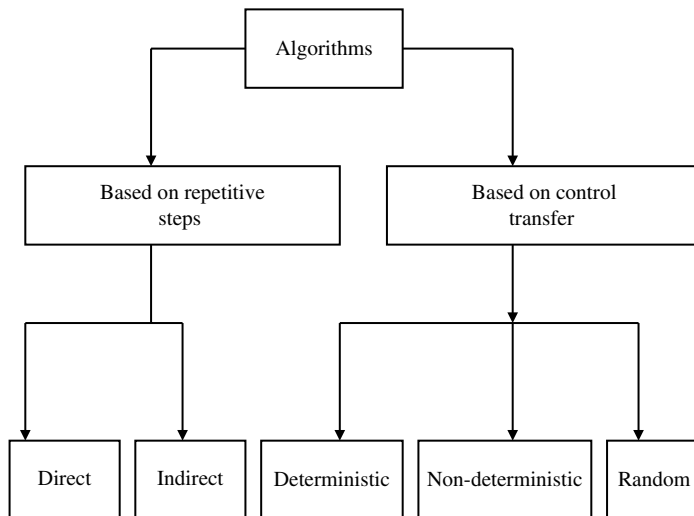


Figure 1.17 Classifications of algorithms

On the basis of repetitive steps, an algorithm can further be classified into two types.

- (i) **Direct Algorithm:** In this type of algorithm, the number of iterations is known in advance. For example, for displaying numerical numbers from 1 to 10, the loop variable should be initialized from 1 to 10. The statement would be as follows:
`for (j=1; j<=10; j++)`
 In the above statement, it is predicted that the loop will iterate 10 times.
- (ii) **Indirect Algorithm:** In this type of algorithm, repetitively steps are executed. Exactly how many repetitions are to be made is unknown.

For example, the repetitive steps are as follows:

- (i) To find the first five Armstrong numbers from 1 to n, where n is the fifth Armstrong number.
- (ii) To find the first three palindrome numbers.

Based on the control transfer, the algorithms are categorized in the following three types.

- (i) **Deterministic:** Deterministic algorithm is based on either to follow a ‘yes’ path or ‘no’ path based on the condition. In this type of algorithm when control comes across a decision logic, two paths ‘yes and ‘no’ are shown. Program control follows one of the routes depending upon the condition.

Example:

Testing whether a number is even or odd. Testing whether a number is positive or negative.

- (ii) **Non-deterministic:** In this type of algorithm to reach to the solution, we have one of the multiple paths.

Example:

To find a day of a week.

- (iii) **Random algorithm:** After executing a few steps, the control of the program transfers to another step randomly, which is known as a random algorithm.

Example:

A random search

Another kind of an algorithm is the infinite algorithm.

Infinite algorithms: This algorithm is based on better estimates of the results. The number of steps required would not be known in advance. The process will be continued until the best results emerged. For final convergence more iterations would be required.

Example:

To find shortest paths from a given source to all destinations in the network.

1.17 FLOWCHARTS

A flowchart is a visual representation of the sequence of steps for solving a problem. It enlightens what comes first, second, third, and so on. A completed flowchart enables you to organize your problem into a plan of actions. Even for designing a product a designer many times has to draw a flowchart. It is a working map of the final product. This is an easy way to solve the complex designing problems. The reader follows the process quickly from the flowchart instead of going through the text.

A flowchart is an alternative technique for solving a problem. Instead of descriptive steps, we use pictorial representation for every step. It shows a sequence of operations. A flowchart is a set of symbols, which indicates various operations in the program. For every process, there is a corresponding symbol in the flowchart. Once an algorithm is written, its pictorial representation can be done using flowchart symbols. In other words, a pictorial representation of a textual algorithm is done using a flowchart.

We give below some commonly used symbols in flowcharts.

Start and end: The start and end symbols indicate both the beginning and the end of the flowchart. This symbol looks like a flat oval or is egg shaped. Figure 1.18 shows the symbol of Start/stop. Only one flow line is combined with this kind of symbol. We write START, STOP or END in the symbols of this kind. Usually this symbol is used twice in a flowchart, that is, at the beginning and at the end.

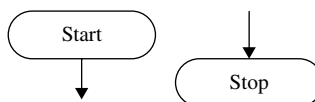


Figure 1.18 Start/stop symbol

Decision or test symbol: The decision symbol is diamond shaped. This symbol is used to take one of the decisions. Depending on the condition the decision block selects one of the alternatives. While solving a problem, one can take a single, two or multiple alternatives depending upon the situation. All these alternatives are illustrated in this section. A decision symbol with a single alternative is shown in Figure 1.18. In case the condition is satisfied /TRUE a set of statement(s) will be executed otherwise for false the control transfers to exit.

Single alternative decision: Here more than one flow line can be used depending upon the condition. It is usually in the form of a ‘yes’ or ‘no’ question, with branching flow line depending upon the answer. With a single alternative, the flow diagram will be as per Figure 1.19.

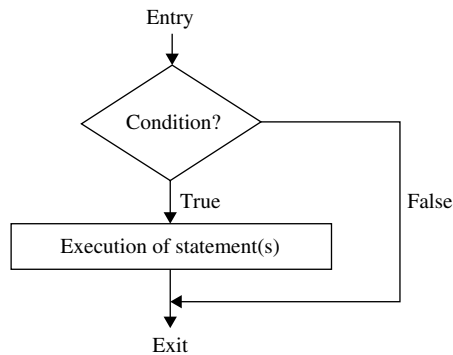


Figure 1.19 *Single alternative decision*

Two alternative decisions: In Figure 1.20 two alternative paths have been shown. On satisfying the condition statement(s) pertaining to 1 action will be executed, otherwise the other statement(s) for action 2 will be executed.

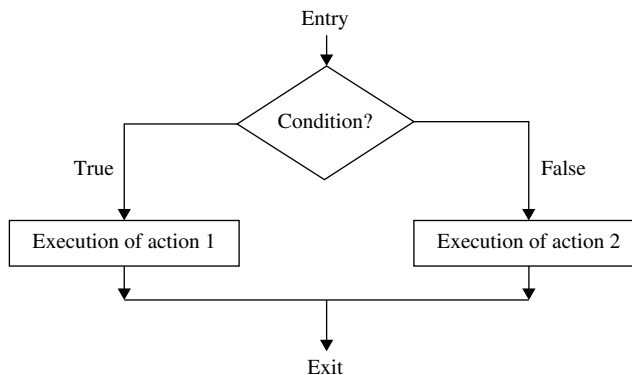


Figure 1.20 *Two alternative decisions*

Multiple alternative decisions: In Figure 1.21 multiple decision blocks are shown. Every decision block has two branches. In case the condition is satisfied, execution of statements of appropriate blocks take place, otherwise next condition will be verified. If condition 1 is satisfied then block 1 statements are executed. In the same way, other decision blocks are executed.

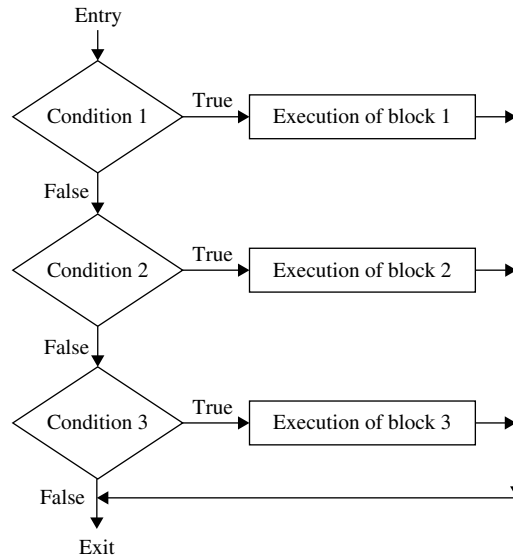


Figure 1.21 Multiple alternative decisions

Connector symbol: A connector symbol has to be shown in the form of a circle. It is used to establish the connection, whenever it is impossible to directly join two parts in a flowchart. Quite often, two parts of the flowcharts may be on two separate pages. In such a case, connector can be used for joining the two parts. Only one flow line is shown with the symbol. Only connector names are written inside the symbol, that is, alphabets or numbers. Figure 1.22 shows the connector symbol.

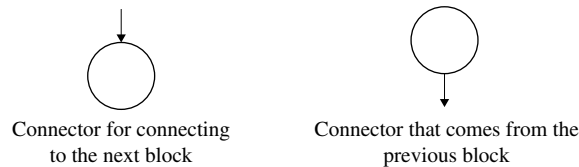


Figure 1.22 Connector symbol

Process symbol: The symbol of process block should be shown by a rectangle. It is usually used for data handling, and values are assigned to the variables in this symbol. Figure 1.23 shows the process symbol. The operations mentioned within the rectangular block will be executed when this kind of block is entered in the flowchart. Sometimes an arrow can be used to assign the value of a variable to another. The value indicated at its head is replaced by the tail values. There are two flow lines connected with the process symbol. One line is incoming and the other line goes out.

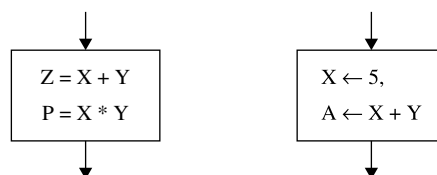


Figure 1.23 Process symbol

Loop symbol: This symbol looks like a hexagon. This symbol is used for implementation of for loops only. Four flow lines are associated with this symbol. Two lines are used to indicate the sequence of the program and remaining two are used to show the looping area, that is, from the beginning to the end.

For the sake of understanding, Figure 1.24. illustrates the working of for loop. The variable J is initialized to 0 and it is to be incremented by a step of 2 until it reaches the final value 10. For every increased value of J, body of the loop is executed. This process will be continued until the value of J reaches 10. Here the next block is shown for the repetitive operation.

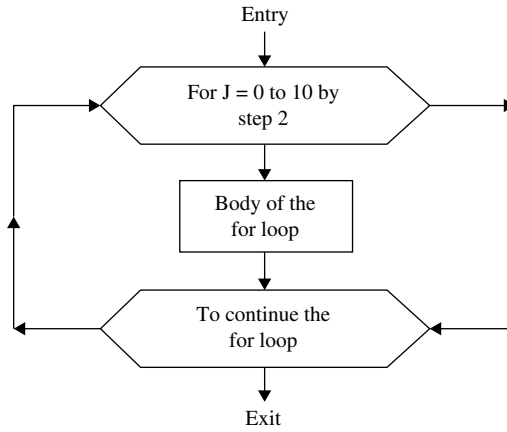


Figure 1.24 For loop

Input/output symbol: Input/output symbol looks like a parallelogram, as shown in Figure 1.25. The input/output symbol is used to input and output the data. When the data is provided to the program for processing, then this symbol is used. There are two flow lines connected with the input/output symbol. One line comes to this symbol and the other line goes from this symbol.

As per Figure 1.25 compiler reads the values of X, Y and in the second figure the result is displayed on the monitor or the printer.

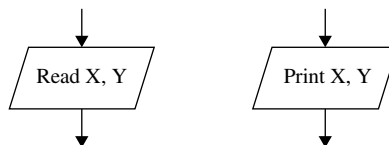


Figure 1.25 Input/output symbol

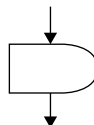


Figure 1.26 Delay symbol

Delay symbol: Symbol of delay is just like 'AND' gate. It is used for adding delay to the process. It is associated with two lines. One is incoming and the other is outgoing, as shown in Figure 1.26.

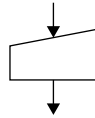


Figure 1.27 Manual input symbol

Manual input symbol: This is used for assigning the variable values through the keyboard, whereas in data symbol the values are assigned directly without manual intervention. Figure 1.27 represents the symbol of manual input.

In addition, the following symbols (Figure 1.28) can be used in the flowchart and they are parts of flowcharts.

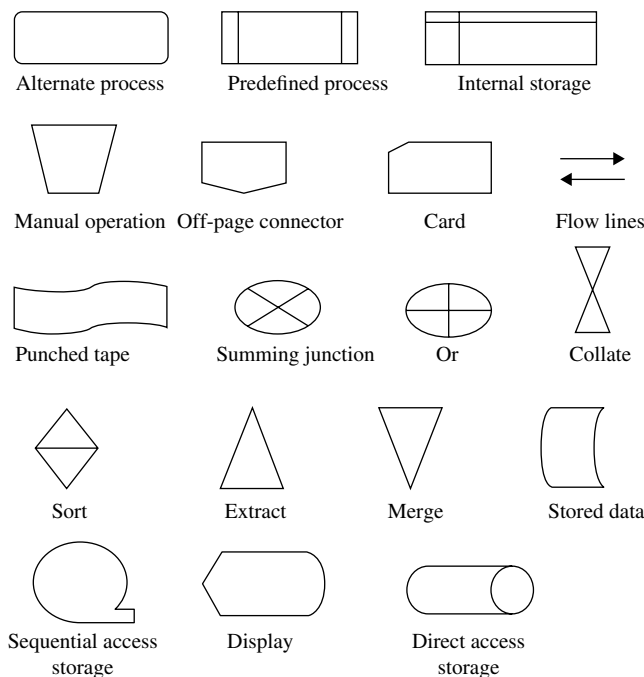


Figure 1.28 Some other symbols used in the flowchart

1.18 PSEUDOCODE

In pseudocodes english-like words are used to represent the various logical steps. It is a prefix representation. Here solution of each step is described logically. The pseudocode is just the raw idea about the problem. By using this tip, one can try to solve the problem. The meaning of pseudocode is 'false code.' The syntax rule of any programming language cannot be applied to pseudocode.

Example (a): Assume a and b are two numbers and find the larger out of them. In such a case, comparison is made between them.

This can be represented as follows.

Algorithm	Pseudocode
Input a and b.	get numbers a & b
Is $a > b$.	Compare a & b
If yes a is larger than b.	if a is large $\text{max} = a$
If no b is larger than a.	if b is large $\text{max} = b$
Print the larger number.	Larger number is max

Few skilled programmers prefer to write pseudocode for drawing the flowchart. This is because using pseudocode is analogous to writing the final code in the programming language. Few programmers prefer to write the steps in algorithm. Few programmers favour flowchart to represent the logical flow because with visualization things are clear for writing program statements. For beginners a flowchart is a straightforward tool for writing a program.

Example (b): The example (b) illustrates how the pseudo code is used to draw the flowchart for squaring a number.

- (i) Accept number
- (ii) Calculate square of the number
- (iii) Display number

All the steps of the program are written down in steps. Some programs follow pseudocode to draw flowcharts. Using pseudocode, final program can be written. Majority of programs have common tasks such as input, processing and output. These are fundamental tasks of any program.

Using pseudocode a flowchart can be drawn as per the following steps.

For the statement, that is, 'Accept number' the flowchart symbol is as per Figure 1.29.

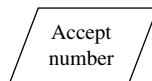


Figure 1.29 Input symbol

The statements including arithmetic operations are examples of processing statements. The representation of second statement 'Calculate square of the number' can be represented as in Figure 1.30.

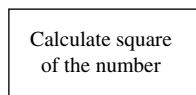


Figure 1.30 Processing symbol

The output statement, that is, 'Display number' can be represented as per Figure 1.31.

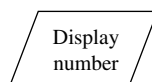


Figure 1.31 Output symbol

In addition, the flowchart has two more symbols to indicate the beginning and the end of the program as per Figure 1.32. The standard terminator symbol is racetrack.

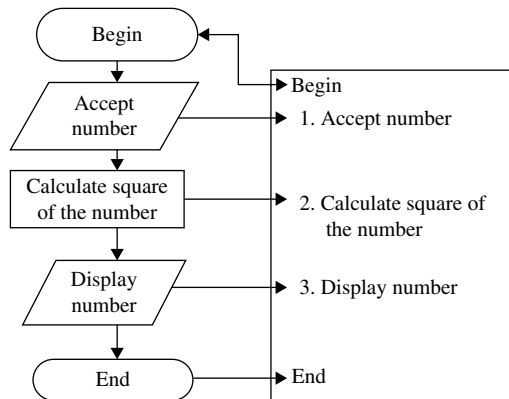


Figure 1.32 Pseudocode and flowchart of a program

SUMMARY

This chapter presents the evolution and basics of C. C is a structural language. It has many similarities like other structural languages such as Pascal and Fortran. C is also called a system-programming language. The ANSI C standard was adopted in December 1989 and the first copy of C language was introduced in the market in 1990.

The reader is exposed to an assembler that translates the symbolic code of programs of an assembly language into machine language instructions. Similarly, compilers are the translators, which translate all the instructions of the program into machine codes and can be used again and again. An interpreter comes in the group of translators. It helps the user to execute the source program with few differences as compared to compilers.

In this chapter, an overview of algorithms was given. An algorithm is defined as ‘the finite set of the steps, which provide a chain of actions for solving a definite nature of problem’. Algorithms are of two types, direct algorithm and indirect algorithm.

EXERCISES

I True or false:

1. C Language is developed by Ken Thompson.
2. C Language was developed in the year 1972.
3. C Language is closely associated with Linux.
4. C Programs are not portable.
5. The ANSI C standard was developed in 1989.
6. C Programs are translated into object code by a compiler.
7. An interpreter reads one line at a time.
8. Every C Program should have the `main()` function.
9. In C, all the statements should be written in small letters only.
10. After compilation, the object file of a source program is created.
11. It is not possible to create .exe file in C.
12. Compiler executes a program even if the program contains warning messages.

13. In Turbo-C editor Alt+C is used to execute the program.
14. A comment can be split in more than one line.
15. The source code for the UNIX operating system is in C.
16. The assembly language program is in alphanumeric symbols.
17. Linking software is used to bring together the source program and library code.
18. Assembler translates low-level language to machine code.
19. The compiler reads firstly entire program and generates the object code.
20. C does not have automatic conversion of compatible variable.
21. Every processor has its own assembly language.
22. Assembly language program is portable.

II Select the appropriate option from the multiple choices given below:

1. The C language has been developed by
 - (a) Patrick Naughton
 - (b) Dennis Ritchie
 - (c) Ken Thompson
 - (d) Martin Richards
2. The C programming is a
 - (a) high-level language
 - (b) low-level language
 - (c) middle-level language
 - (d) assembly language
3. The C programs are converted into machine language using
 - (a) an assembler
 - (b) a compiler
 - (c) an interpreter
 - (d) an operating system
4. The C language was developed in the year
 - (a) 1972
 - (b) 1980
 - (c) 1975
 - (d) 1971
5. The C language has been developed at
 - (a) AT & T Bell Labs, USA
 - (b) IBM, USA
 - (c) Borland International, USA
 - (d) Sun Microsystems
6. The C language is an offspring of the
 - (a) 'BPCL' language
 - (b) 'ALGOL' language
 - (c) 'Basic' language
 - (d) None of the above
7. The C program should be written only in
 - (a) lower case
 - (b) upper case
 - (c) title case
 - (d) sentence case
8. The role of a compiler is to translate source program statements to
 - (a) object codes
 - (b) octal codes
 - (c) decimal codes
 - (d) None of the above
9. The extension for C program files by default is
 - (a) '.c'
 - (b) '.d'
 - (c) '.obj'
 - (d) '.exe'
10. The C can be used with
 - (a) only UNIX operating system
 - (b) only LINUX operating system
 - (c) only MS-DOS operating system
 - (d) All the above
11. The C language is closely associated with
 - (a) MS-DOS
 - (b) LINUX
 - (c) UNIX
 - (d) MS-windows
12. The C programs are highly portable means
 - (a) same programs execute on different computers
 - (b) program executes only on the same computer
 - (c) program needs a lot of modification to run
 - (d) None of the above

13. Each instruction in C program is terminated by
 - (a) dot (.)
 - (b) comma (,)
 - (c) semi-colon (;)
 - (d) curly brace ({})
14. Which one of the following statements is incorrect?
 - (a) a compiler compiles the source program
 - (b) an assembler takes as assembly program as input
 - (c) interpreter executes the complete source code just like compiler
 - (d) None of the above
15. ANSI committee was set up in
 - (a) 1983
 - (b) 1985
 - (c) 1990
 - (d) 1976
16. The program which translates high-level program into its equivalent machine language program is called
 - (a) a translator
 - (b) a language processor
 - (c) a converter
 - (d) None of the above
17. C is an offspring of the
 - (a) basic combined programming language
 - (b) basic computer programming language
 - (c) basic programming language
 - (d) None of the above
18. An interpreter reads the source code of a program
 - (a) one line at a time
 - (b) two lines at a time
 - (c) complete program in one stroke
 - (d) None of the above
19. A compiler reads the source code of a program
 - (a) complete program in one stroke
 - (b) one line at a time
 - (c) two lines at a time
 - (d) None of the above
20. C keywords are reserved words by
 - (a) a compiler
 - (b) an interpreter
 - (c) a header file
 - (d) Both (a) and (b)
21. The declaration of C variable can be done
 - (a) anywhere in the program
 - (b) in declaration part
 - (c) in executable part
 - (d) at the end of program
22. Variables must begin with character without spaces but it permits
 - (a) an underscore symbol (_)
 - (b) an asterisk symbol (*)
 - (c) an ampersand symbol (&)
 - (d) None of the above
23. In C, the statements following `main()` are enclosed within
 - (a) { }
 - (b) ()
 - (c) < >
 - (d) None of the above
24. CPU generates
 - (a) timing signals
 - (b) control signals
 - (c) analog signals
 - (d) both a & b
25. The structural languages are
 - (a) C
 - (b) Pascal
 - (c) Fortran
 - (d) All the above
26. Data type-less languages are
 - (a) C
 - (b) BCPL
 - (c) B
 - (d) Both c & b
27. Input device/s of a computer
 - (a) Printer
 - (b) Speaker
 - (c) Monitor
 - (d) None of the above
28. Output device/s of the computer
 - (a) Monitor
 - (b) Speaker
 - (c) Printer
 - (d) All the above
29. CPU comprises of the
 - (a) Arithmetic and Logical unit
 - (b) Registers
 - (c) Control Signals
 - (d) All the above

30. Advantages of high level language are
- (a) fast Program Development
 - (b) testing and Debugging of program is easier
 - (c) portable
 - (d) All the above

III Answer the following questions:

1. Why to use a computer? Brief its benefits.
2. Illustrate the functions of various parts of a computer.
3. Why is the C language called a middle-level language?
4. What are the functions of an interpreter and a compiler?
5. What is the difference between an interpreter and a compiler?
6. What is meant by compilation? Explain in detail.
7. Write the rules for writing a C program.
8. Elaborate different sections of a C program.
9. Explain the functions of a linker.
10. What is the role of curly braces ({}) in a C program?
11. What is ANSI C standard?
12. What are the user-defined functions?
13. Explain the assembly-language and machine-language concepts.
14. Write details on header files `stdio.h` and `conio.h`.
15. Which are the standard directories? Where are .h files kept?
16. Write any three advantages of a C language.
17. What is an algorithm? Explain in short.
18. Mention and explain two types of algorithms.
19. Show the flowchart for finding cube of a number.
20. Write the pseudo code for finding largest out of three numbers.

ANSWERS

I True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	F	2.	T	3.	F	4.	F	5.	T
6.	T	7.	T	8.	T	9.	T	10.	T
11.	F	12.	T	13.	F	14.	T	15.	T
16.	T	17.	T	18.	T	19.	T	20.	T
21.	T	22.	F						

II Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	b	2.	c	3.	b	4.	a	5.	a
6.	a	7.	a	8.	a	9.	a	10.	d
11.	c	12.	a	13.	c	14.	c	15.	a
16.	a	17.	a	18.	a	19.	a	20.	a
21.	b	22.	a	23.	a	24.	d	25.	d
26.	d	27.	d	28.	d	29.	d	30.	d

CHAPTER

2

The C Declarations

Chapter Outline

- 2.1** Introduction
- 2.2** The C Character Set
- 2.3** Delimiters
- 2.4** Types of Tokens
- 2.5** The C Keywords
- 2.6** Identifiers
- 2.7** Constants
- 2.8** Variables
- 2.9** Rules for Defining Variables
- 2.10** Data Types
- 2.11** C Data Types
- 2.12** Integer and Float Number Representations
- 2.13** Declaring Variables
- 2.14** Initializing Variables
- 2.15** Dynamic Initialization
- 2.16** Type Modifiers
- 2.17** Type Conversion
- 2.18** Wrapping Around
- 2.19** Constant and Volatile Variables

2.1 INTRODUCTION

The programming languages are designed to support certain kind of data, such as numbers, characters, strings, etc., to get useful output known as result/information. Data consists of digits, alphabets and symbols.

A program should support these data types for getting the required output known as information. A program is a set of statements, which performs a specific task, executed by users in a sequential form. These statements/instructions are formed using certain words and symbols according to the rules known as syntax rules or grammar of the language. Every program must accurately follow the syntax rules supported by the language.

In this chapter, C character set, the type of variables, types of tokens, delimiters, data types, variable initialization and dynamic initialization are described.

2.2 THE C CHARACTER SET

A character is a part of a word, sentence or paragraph. By using different characters, words, expressions and statements can be created on the basis of the requirement. This creation depends upon the computer on which a program runs.

A character is represented by any alphabets in lowercase or uppercase, digits or special characters.

Figure 2.1 presents valid C character set which are as follows: (i) Letters (ii) Digits (iii) White Spaces & Escape Sequences and (iv) Special Characters.

As C character set consists of escape sequences so each escape sequence begins with back slash (\) and it represents a single character. Any character can be represented as character constant using escape sequence.

The complete character set is listed in Tables 2.1 and 2.2.

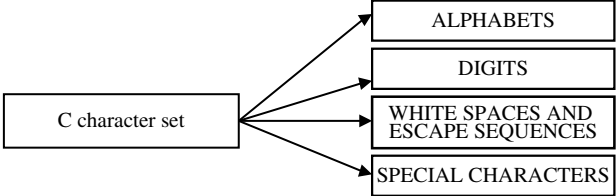


Figure 2.1 C character set

Table 2.1 Character set

Letters	Digits	White Spaces and Escape Sequences
Uppercase A to Z	All decimal digits 0 to 9	Back space \b
Lowercase a to z		Horizontal tab \t
		Vertical tab \v
		New line \n
		Form feed \f
		Backslash \\
		Alert bell \a
		Carriage return \r
		Question mark \?
		Single quote \'
		Double quote \"
		Octal number \o or \oo or \ooo
		Hexadecimal number \xhh

The special characters listed in Table 2.2 are represented in the computer by numeric values. The C characters are assigned unique codes. There are many character codes used in the computer system. The widely and standard codes used in computer are ASCII and EBCDIC (Extended Binary Coded Decimal Interchange Code).

Table 2.2 *List of special characters*

,	Comma	&	Ampersand
.	Period or dot	^	Caret
;	Semi-colon	*	Asterisk
:	Colon	—	Minus sign
'	Apostrophe	+	Plus sign
"	Quotation mark	<	Less than
!	Exclamation mark	>	Greater than
	Vertical bar	()	Parenthesis (left/right)
/	Slash	[]	Brackets (left/right)
\	Back slash	{ }	Curly braces (left/right)
~	Tilde	%	Percent sign
_	Underscore	#	Number sign or hash
\$	Dollar	=	Equal to
?	Question mark	@	At the rate

ASCII Code: ASCII stands for American Standard Code for Information Interchange, is the code of two types. One uses 7 bits and other uses 8 bits. The 7-bit code represents 128 different characters and the 8-bit code represents 256 characters. For example, the character A is represented in 7-bit code as 1000001_2 (decimal 65). Refer to Appendix A for the list of characters and their equivalent number.

2.3 DELIMITERS

The language pattern of C uses special kind of symbols, which are called delimiters. They are depicted in Table 2.3.

Table 2.3 *Delimiters*

<i>Delimiters</i>	<i>Symbols</i>	<i>Uses</i>
Colon	:	Useful for label
Semi-colon	;	Terminates statements
Parenthesis	()	Used in expression and function
Square brackets	[]	Used for array declaration
Curly braces	{ }	Scope of statement
Hash	#	Preprocessor directive
Comma	,	Variable separator
Angle brackets	<>	Header file

2.4 TYPES OF TOKENS

A compiler designer prescribes rules for making a program using statements. The smallest unit in a program/statement is called a token. The compiler identifies them as tokens. Tokens are classified in the following types. Figure 2.2 shows the types of tokens supported by C.

The tokens are as follows:

- (i) **Keywords:** Key words are reserved by the compiler. There are 32 keywords (ANSI Standard).
- (ii) **Variables:** These are user defined. Any number of variables can be defined.
- (iii) **Constants:** Constants are assigned to variables.
- (iv) **Operators:** Operators are of different types and are used in expressions.
- (v) **Special Characters:** These characters are used in different declarations in C.
- (vi) **Strings:** A sequence of characters.

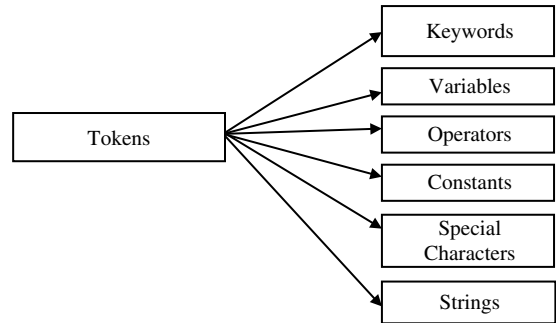


Figure 2.2 Types of tokens

2.5 THE C KEYWORDS

The C keywords are reserved words by the compiler. All the C keywords have been assigned fixed meanings and they cannot be used as variable names. However, few C compilers allow constructing variable names, which exactly coincide with the keywords. It is suggested that the keywords should not be mixed up with variable names. For utilizing the keywords in a program, no header file is required to be included. The descriptions of the keywords are explained in the different programs/chapters. The 32 C keywords provided in ANSI C are listed in Table 2.4.

Table 2.4 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			
Additional Keywords for Borland C						
asm	cdecl	far	huge	interrupt	near	pascal

2.6 IDENTIFIERS

A symbolic name is used to refer to a variable, constant, function, structure, union etc. This process is done with identifiers. In other words, identifiers are the names of variables, functions, arrays, etc.

They refer to a variety of entities such as structures, unions, enumerations, constants, **typedef** names, functions and objects (see Figure 2.3). An identifier always starts with an alphabet, and it is a plain sequence of alphabets and/or digits. C identifier does not allow blank spaces, punctuations and signs.

Identifiers are user-defined names. They are generally defined in lowercase letters. However, the uppercase letters are also permitted. The underscore (`_`) symbol can be used as an identifier. In general, an underscore is used by a programmer as a link between two words for the long identifiers.

The programmer can refer to programming example 2.3 in which the identifiers are not defined in lowercase but with a combination of lowercase and uppercase letters.

Valid identifiers are as follows:
length, area, volume, sUM, Average

Invalid identifiers are as follows:
Length of line, S+um, year's

Example:

User-defined identifiers are as follows:

- (i) `#define N 10`
- (ii) `#define a 15`

Here, 'N' and 'a' are user-defined identifiers.

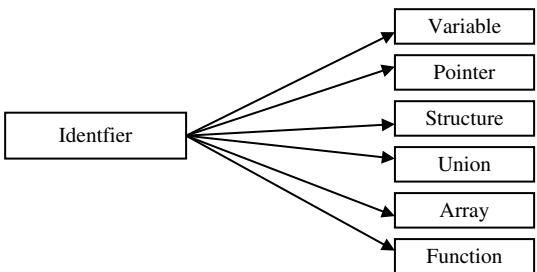


Figure 2.3 Identifiers

2.7 CONSTANTS

The constants in C are applicable to the values, which do not change during the execution of a program. There are several types of constants in C. They are classified into following groups as given in Figure 2.4 (e.g. Table 2.5).

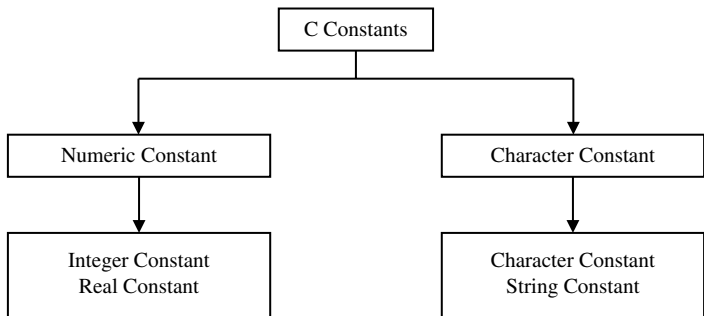


Figure 2.4 C constants

Table 2.5 Constant types

Example	Constant Type
542	Integer constant
35.254	Floating-point constant
0 × 54	Hexadecimal integer constant
0171	Octal integer constant
'C'	Character constant
"cpp"	String constant

2.7.1 | Numerical Constants

- (a) **Integer Constants:** These constants are represented with whole numbers. They require a minimum of 2 bytes and a maximum of 4 bytes of memory.

The following concepts are essential to follow the numerical constants:

- (a) Numerical constants are represented with numbers. At least one digit is needed for representing the number.
- (b) The decimal point, fractional part, or symbols are not permitted. Neither blank spaces nor commas are permitted.
- (c) Integer constant could be either positive or negative or may be zero.
- (d) A number without a sign is assumed as positive.

Some valid examples: 10, 20, +30, -15, etc.

Some invalid integer constants: 2.3, .235, \$76, 3⁶, etc.

Besides representing the integers in decimal, they can also be represented in octal or hexadecimal number system based on the requirement.

Octal number system has base 8 and the hexadecimal number system has base 16. The octal numbers are 0, 1, 2, 3, 4, 5, 6, and 7 and the hexadecimal numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

The representation of octal numbers in C would be done with leading digit 0 and for hex representation, with leading 0X or 0x.

Examples of octal and hexadecimal numbers:

Octal numbers – 027, 037, 072

Hexadecimal numbers – 0X9, 0Xab, 0X4

- (b) **Real Constants:** Real constants are often known as floating point constants. Real constants can be represented in exponential or fractional form. Integer constants are unfit to represent many quantities. Many parameters or quantities are defined not only in integers but also in real numbers. For example, length, height, price, distance, etc. are also measured in real numbers.

The following concepts are essential to follow the real numbers:

- (a) The decimal point is permitted.
- (b) Neither blank spaces nor commas are permitted.
- (c) Real numbers could be either positive or negative.
- (d) The number without a sign is assumed as positive.

Examples of real numbers are 2.5, 5.521, 3.14, etc.

The real constants can be written in exponential notation, which contains fractional and exponential parts. For example, the value 2456.123 can be written as 2.4561 X e³.

The part that precedes e is called mantissa and the part that follows it is an exponent. In this example, 2.4561 is the mantissa and +3 is the exponent.

The following points must be noted while constructing a real number in exponential form:

- (a) The real number should contain a mantissa and an exponent.
- (b) The letter 'e' separates the mantissa and the exponent and it can be written in lower or upper case.
- (c) The mantissa should be either a real number represented in decimal or integer form.
- (d) The mantissa may be either positive or negative.
- (e) The exponent should be an integer that may be positive or negative.

Some valid examples are 5.2e2, -2, 5.0e-5, 0.5e-3, etc.

In double type also the real numbers can be expressed with mantissa and exponent parts.

2.7.2 | Character Constant

- (i) **Single Character Constants:** A character constant is a single character. It can also be represented with single digit or a single special symbol or white space enclosed within a pair of single quote marks or character constants are enclosed within single quotation marks.

Example:

'a', '8', '-'

Length of a character, at the most, is one character.

Character constants have integer values known as ASCII values. For example the statement `printf("%c %d", 65, 'B')` will display character 'A' and 66.

- (ii) **String Constants:** String constants are a sequence of characters enclosed within double quote marks. The string may be a combination of all kinds of symbols.

Example:

"Hello", "India", "444", "a".

A programming example is given for various constants.

➤ 2.1 Write a program on various constants.

```
void main()
{
    int x;
    float y;
    char z;
    double p;
    clrscr();
    x=20;
    y=2e1;
    z='a';
    p=3.2e20;
    printf("\n%d %10.2f %c %.2lf", x, y, z, p);
    getch();
}
```

OUTPUT:

20 20.00 a 32000000000000000000.00

2.8 VARIABLES

Variables are the basic objects manipulated in a program. Declaration gives an introduction of variable to compiler and its properties like scope, range of values and memory required for storage. A variable is used to store values. It has memory location and can store single value at a time.

When a program is executed, many operations are carried out on the data. The data types are integers, real or character constants. The data is stored in the memory, and at the time of execution it is fetched for carrying out different operations on it.

A variable is a data name used for storing a data value. Its value may be changed during the program execution. The variable value keeps on changing during the execution of the program. In other words, a variable can be assigned different values at different times during the program execution. A variable name may be declared based on the meaning of the operation. Variable names are made up of letters and digits. Some meaningful variable names are as follows.

Example:

height, average, sum, avg12

➤ 2.2 Write a program to declare and initialize variables and display them.

```
void main()
{
    int age=20;
    float height=5.4;
    char sex='M';
    clrscr();
    printf("Age :%d \nHeight : %g \nSex : %c",age,height,sex);
}
```

OUTPUT:

```
Age: 20
Height: 5.4
Sex: M
```

Explanation:

In the above program, `int`, `height` and `char` variables are declared and values are assigned. Using `printf()` statement values are displayed.

2.9 RULES FOR DEFINING VARIABLES

- (i) A variable must begin with a character or an underscore without spaces. The underscore is treated as one type of character. It is very useful to increase readability of variables having long names. It is advised that the variable names should not start with underscore because library routines mostly use such variable names.
- (ii) The length of the variable varies from compiler to compiler. Generally, most of the compilers support eight characters excluding extension. However, the ANSI standard recognizes the maximum length of a variable up to 31 characters. Names of functions and external variables length may be less than 31 because external names may be used by assemblers and loaders over which language has no control. For external names, the variable names should be of six characters and in single case.

- (iii) The variable should not be a C keyword.
- (iv) The variable names may be a combination of uppercase and lowercase characters. For example, `SuM` and `Sum` are not the same. The traditional practice is to use lowercase characters for variable names and uppercase letters for symbolic constants.
- (v) The variable name should not start with a digit.
- (vi) Blanks and commas are not permitted within a variable name.

➤ 2.3 Write a program to declare variables with different names and data types.

```
void main()
{
    int Num=12;
    int WEIGHT = 25;
    float HeIgHt=4.5;
    char name[10]="AMIT";
    clrscr();
    printf("Num :%d \nWEIGHT : %d \nHeight : %g \nName
    =%s", Num, WEIGHT, HeIgHt, name);
}
```

OUTPUT:

```
Num: 12
WEIGHT: 25
Height: 4.5
Name = AMIT
```

Explanation:

In this program, you might have noticed that variables of different data types with different naming styles are defined. For example `Num`, `WEIGHT`, `HeIgHt` and `name` are variables. Values are assigned to variables and displayed in the same way as usual.

2.10 DATA TYPES

All C compilers support a variety of data types. This enables the programmer to select the appropriate data type as per the need of the application. Generally, data is represented using numbers or characters. The numbers may be integers or real. The basic data types are `char`, `int`, `float` and `double`.

The C supports a number of qualifiers that can be applied to these basic data type. The short and long qualifiers provide different length to `int`. The short is always 16 bits and long is 32 bits, i.e. `int` is either 16 or 32 bits. The compiler is free to choose a suitable size depending upon the hardware. However, the restriction is that `short` or `int` is at least of 2 bytes and `long` is at least of 4 bytes. The `short` should not be longer than `int` and `int` should not be longer than `long`.

The unsigned and signed qualifiers can be applied to `char`, `int` and `long`. The unsigned data types are always zero or positive. They follow the law of arithmetic modulo 2^n , where n is the number of bits in the type. For example, `char` are 8 bits with combination of negative and positive values (−128 to 127) (two's complement). The unsigned `char` supports values 0 to 255. The characters are signed or unsigned, the printable `char` are always positive (see Figure 2.5 for a detailed classification).

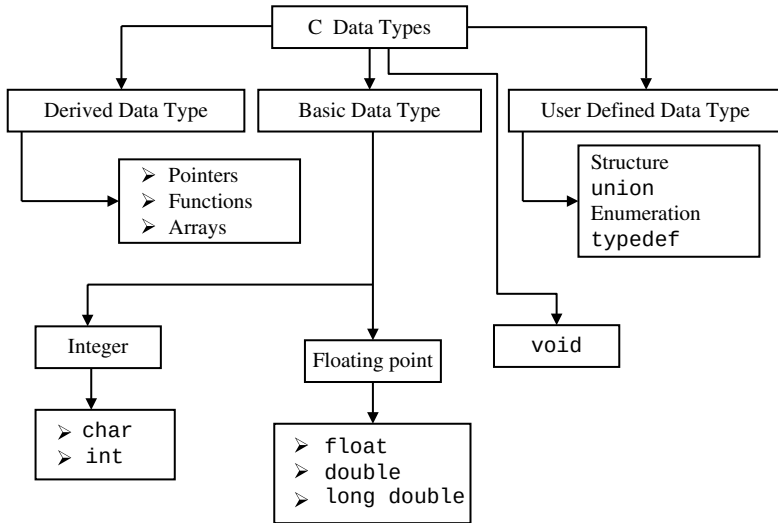


Figure 2.5 C data types

C data type can be classified as follows:

- (i) **Basic Data Type:**
 - (a) Integer (`int`), (b) character (`char`), (c) floating point (`float`), (d) double floating point (`double`).
- (ii) **Derived Data Type:** Derived data types are pointers, functions and arrays. Pointers are explained in Chapter 9, functions are explained in Chapter 10 and arrays are explained in Chapter 7.
- (iii) **User-defined Type:** Struct, union and typedef are user-defined data types, which are explained in Chapter 13.
- (iv) **void Data Type:** This data type is explained in Chapter 10.

2.11 C DATA TYPES

(1) Integer Data Type

(a) `int`, `short` and `long`

All C compilers offer different integer data types. They are `short` and `long`. Short integer requires half the space in memory than the long. The short integer requires 2 bytes and long integers 4 bytes. Brief description on them is given in Table 2.6.

Table 2.6 Difference between short and long integers

Short Integer	Long Integer
Occupies 2 bytes in memory	Occupies 4 bytes in memory
Range: -32,768 to 32,767	Range: -2147483648 to 2147483647
Program runs faster	Program runs slower
Format specifier is <code>%d</code> or <code>%i</code>	Format specifier is <code>%ld</code>

(Continued)

Table 2.6 *Difference between short and long integers (Continued)*

<i>Short Integer</i>	<i>Long Integer</i>
<i>Example:</i> <code>int a=2;</code> <code>short int b=2;</code>	<i>Example:</i> <code>long b=123456;</code> <code>long int c=1234567;</code>
When variable is declared without <code>short</code> or <code>long</code> keyword, the default is short-signed <code>int</code> .	

(b) Integers Signed and Unsigned

The difference between the signed integers and unsigned is given in Table 2.7.

Table 2.7 *Difference between signed and unsigned integers*

<i>Signed Integer</i>	<i>Unsigned Integer</i>
Occupies 2 bytes in memory	Occupies 2 bytes in memory
Range: -32,768 to 32,767	Range: 0 to 65535
Format specifier is <code>%d</code> or <code>%i</code>	Format specifier is <code>%u</code>
By default signed <code>int</code> is short signed <code>int</code>	By default unsigned <code>int</code> is short unsigned <code>int</code>
There are also long signed integers having Range from -2147483648 to 2147483647.	There are also long unsigned <code>int</code> with range 0 to 4294967295
<i>Example:</i> <code>int a=2;</code> <code>long int b=2;</code>	<i>Example:</i> <code>unsigned long b=567898;</code> <code>unsigned short int c=223;</code>
	When a variable is declared as unsigned the negative range of the data type is transferred to positive, i.e. doubles the largest size of the possible value. This is due to declaring unsigned <code>int</code> ; the 16 th bit is free and not used to store the sign of the number.

(2) `char`, signed and unsigned: Brief description on these data types is given in Table 2.8

Table 2.8 *Difference between signed and unsigned char*

<i>Signed Character</i>	<i>Unsigned Character</i>
Occupies 1 bytes in memory	Occupies 1 bytes in memory
Range: -128 to 127	Range: 0 to 255
Format specifier: <code>%C</code>	Format specifier: <code>%C</code>
When printed using <code>%d</code> format specifier, prints ASCII character.	When printed using <code>%d</code> format specifier, prints ASCII character
<code>char ch='b';</code>	<code>unsigned char ='b';</code>

(3) **Floats and Doubles:** Table 2.9 shows the description of floats and double floats.

Table 2.9 *Difference between floating and double floating*

<i>Floating</i>	<i>Double Floating</i>
Occupies 4 bytes in memory	Occupies 8 bytes in memory
Range: 3.4e-38 to +3.4e+38	Range: 1.7 e-308 to +1.7e+308
Format string: %f	Format string: %lf
<i>Example:</i> float a;	<i>Example:</i> double y;
	There also exist long double having ranged 3.4e - 4932 to 1.1e + 4932 and occupies 10 bytes in memory.
	<i>Example:</i> long double k;

- (4) Entire data types in C: The entire data types supported by the C as illustrated above are given in Table 2.10 for the convenience of the reader for understanding.

Table 2.10 *Data types and their control strings*

<i>Data Type</i>	<i>Size (bytes)</i>	<i>Range</i>	<i>Format String</i>
char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short or int	2	-32,768 to 32,767	%i or %d
unsigned int	2	0 to 65535	%u
long	4	-2147483648 to 2147483647	%ld
unsigned long	4	0 to 4294967295	%lu
float	4	3.4 e-38 to 3.4 e + 38	%f or %g
double	8	1.7 e-308 to 1.7 e + 308	%lf
long double	10	3.4 e-4932 to 1.1 e + 4932	%lf
enum	2*	-32768 to 32767	%d

*The size may vary according to the number of variables used with enum data type.

2.12 INTEGER AND FLOAT NUMBER REPRESENTATIONS

2.12.1 | Integer Representation

Recall that an integer with sign is called a signed integer. The signed integer has signs positive or negative. The signs are represented in the computer in the binary format as 1 for - (minus) and 0 for + (plus) or vice versa. The sign bit is always coded as leftmost bit. At the time of storing the signed number, system reserves the leftmost bit for the representation of the sign of the number.

For example, positive signed numbers are represented in the form called signed magnitude form. In this form, the sign is represented by a binary 0 and the remaining magnitude by equivalent binary form.

+7 is represented as 0 0000111

The signed negative integers can be represented in any one of the following forms:

- (a) signed—magnitude form
- (b) signed—1's complement form
- (c) signed—2's complement form

In the signed magnitude form, the sign of the number is represented as 1 and the magnitude by equivalent binary form.

Example:

–7 is represented as 1 0000111

In the signed 1's complement form, the sign of the integer is indicated by 1 and the magnitude is shown in 1's complement form as follows,

–7 is represented as 1 1111000

In the above signed – 2's complement form, the sign is indicated by 1 and magnitude by 2's complement form as follows,

–7 is represented by 1 1111001

2.12.2 | Floating-Point Representation

In most of the applications, fractions are often used. The system of the number representation that keeps track of the position of binary and decimal point is better than the fixed-point representation. This system is called floating-point representation.

The real number is integer part as well as fractional part. The real number is also called floating-point number. These numbers are either positive or negative. The real number 454.45 can be written as 4.5445×10^2 or 0.45445×10^3 . This type of representation of number is called the scientific representation. Using this scientific form, any number can be expressed as combination of mantissa and an exponent or we can say the number 'n' can be expressed as $n = m r^e$ where m is the mantissa and r is the radix of the number system and e is the exponent. Mantissa is the fixed-point number. The exponent indicates the position of the binary or decimal point. For example, the number 454.45

0.45445	3
Mantissa	Exponent

The zero in the left most position of the mantissa and exponent indicates the plus sign. The mantissa can be a fraction or integer depending upon make of a computer. Most of the computers use mantissa for fractional system representation.

Example in C:

1.2 E + 12, 2.5E – 2

2.13 DECLARING VARIABLES

The declaration of variables should be done in declaration part of the program. The variable must be declared before they are used in the program. Declaration ensures the following two things: (i) compiler obtains the variable name and (ii) it tells to the compiler data type of the variable being declared and

helps in allocating the memory. The variable can also be declared before `main()` such variables are called external variables. The syntax of declaring a variable is as follows.

Syntax:

```
Data_type variable_name;
```

Example:

```
int age;
char m;
float s;
double k;
int a,b,c;
```

The `int`, `char`, `float` and `double` are keywords to represent data types. Commas separate the variables, in case variables are more than one.

Table 2.11 shows various data types and their keywords.

Table 2.11 *Data types and keywords*

<i>Data Types</i>	<i>Keyword</i>
Character	<code>char</code>
Signed character	<code>signed char</code>
Unsigned character	<code>unsigned char</code>
Integer	<code>int</code>
Signed integer	<code>signed int</code>
Unsigned integer	<code>unsigned int</code>
Unsigned short integer	<code>unsigned short int</code>
Signed long integer	<code>signed long int</code>
Unsigned long integer	<code>unsigned long int</code>
Floating point	<code>float</code>
Double floating point	<code>double</code>
Extended double point	<code>long double</code>

2.14 INITIALIZING VARIABLES

Variables declared can be assigned or initialized using assignment operator '='. The declaration and initialization can also be done in the same line.

Syntax:

```
variable_name = constant;
or
data_type variable_name= constant;
```

Example:

```
x=5; where x is an integer variable.
```

Example:

```
int y=4;
```

Example:

```
int x, y, z;          /* Declaration of variables */
```

The third example as cited above for declaration of variables is also a valid statement. Illustration of the initialization of a variable is shown in Figures 2.6 and 2.7.

In Figure 2.7, the variable *z* is assigned a value 1, and *z* then assigns its value to *y* and again *y* to *x*. Thus, initialization in chain system is done. However, note that following declarations are invalid:

```
int x=y=z=3;          /*invalid statement*/
```

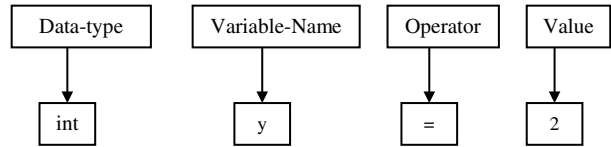


Figure 2.6 Value assignment

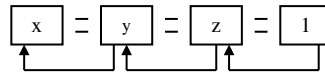


Figure 2.7 Multiple assignment

2.15 DYNAMIC INITIALIZATION

The initialization of variable at run time is called dynamic initialization. Dynamic refers to the process during execution. In C initialization can be done at any place in the program; however the declaration should be done at the declaration part only. Consider the following program.

➤ 2.4 Write a program to demonstrate dynamic initialization.

```
void main()
{
    int r=2;
    float area=3.14*r*r;
    clrscr();
    printf("Area=%g", area);
}
```

OUTPUT:

Area=12.56

Explanation:

In the above program, *area* is calculated and assigned to variable *area*. The expression is solved at a run time and assigned to *area* at a run time. Hence, it is called dynamic initialization.

2.16 TYPE MODIFIERS

The keywords *signed*, *unsigned*, *short* and *long* are type modifiers. A type modifier changes the meaning of basic data type and produces a new data type. Each of these type modifiers is applicable to the basic data type *int*. The modifiers *signed* and *unsigned* are also applicable to the type *char*. The *long* type modifier can be applicable to double type.

Example:

```

long l;           /*int data type is applied*/
int s;           /*signed is default*/
unsigned long int; /*int keyword is optional*/

```

➤ 2.5 Write a program to declare different variables with type modifiers and display them.

```

void main()
{
    short t=1;
    long k=54111;
    unsigned u=10;
    signed j=-10;
    clrscr();
    printf("\n t=%d", t);
    printf("\n k=%ld", k);
    printf("\n u=%u", u);
    printf("\n j=%d", j);
}

```

OUTPUT:

```

t=1
k=54111
u=10
j=-10

```

Explanation:

Here in the above program, all the variables are of integer type. The variable `t` and `k` are short and long type, whereas `u` and `j` are unsigned and signed integers, respectively.

2.17 TYPE CONVERSION

In C, it is possible to convert one data type to another. This process can be done either explicitly or implicitly. The following section describes explicit type conversion.

- (i) Explicit type conversion
 - a) Sometimes a programmer needs the result in certain data type, for example division of 5 with 2 should return float value. Practically the compiler always returns integer values because both the arguments are of integer data type.
- (ii) Implicit type conversion

In C automatically 'lower' data type is promoted to 'upper' data type. For example, data type `char` or `short` is converted to `int`. This type of conversion is called as automatic unary conversion. Even when binary operator has different data types, 'lower' data type is converted to 'upper' data type.

In case an expression contains one of the operands as unsigned operands and another non-unsigned, the latter operand is converted to unsigned.

Similarly, in case an expression contains one of the operands as float operands and another non-float, the latter operand is converted to float.

Similarly, in case an expression contains one of the operands as double operand and another non-double, the latter operand is converted to double.

Similarly, in case an expression contains one of the operands as long int and the other unsigned int operands, the latter operand is converted to long int.

For the sake of understanding, the following table describes automatic data type conversion from one data type to another while evaluating an expression.

<i>Expression</i>	<i>Operator</i>	<i>Operand 1</i>	<i>Operand 2</i>	<i>Outcome</i>
a	unary	short	–	int
a/b	binary	int	float	float
a/b-c	binary	float	long int	float
(a/b-c)*d	binary	float	double	double

➤ 2.6 The following program explains the above concepts.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    short a=5;
    long int b=123456;
    float c=234.56;
    double d=234567.78695;
    clrscr();

    printf("%lf", ((a+b)*c)/d);
    return 0;
}
```

OUTPUT:

123.456900

Explanation:

In the above program, we have taken the variable 'a' as short int, 'b' as long, c as float and 'd' as double. The expression ((a + b)*c)/d gives the result as double float.

Sometimes a programmer needs the result in a certain data type, for example division of 5 with 2 should return float value. Practically, the compiler always returns integer values because both the arguments are of integer data type. This can be followed by the execution of the following program.

➤ 2.7 Write a program to change data type of results obtained by division operations.

```
void main()
{
    clrscr();
    printf("\n Division operation Result");
    printf("\n Two Integers (5 & 2) : %d",5/2);
}
```

```

printf("\n One int & one float (5.5 & 2) : %g",5.5/2);
printf("\n Two Integers (5 & 2) : %g", (float)5/2);
}

```

OUTPUT:

Division operation	Result
Two Integers (5 & 2):	2
One int & one float (5.5 & 2):	2.75
Two Integers (5 & 2):	2.5

Explanation:

In the first division, operation data types are chosen as integer. Hence, the result turns out to be an integer. Actually, the result should be a `float` value but the compiler returns an integer value. In the second division operation, a float value is divided by an integer. The result of division in this case, yields a `float`. The limitation of the first operation is removed in the third division operation using type conversion. Here, both the values are of integer type. The result obtained is converted into `float`. The program uses type casting which is nothing but putting data type in a pair parenthesis before operation. The programmer can specify any data type.

- (i) Suppose you assign a `float` value to an integer variable.

Example:

```
int x=3.5;
```

Here, the fraction part will not be assigned to integer variable and the part will be lost. The integer is 2 bytes length data type without fractional part and float is 4 bytes data type with fractional part.

- (ii) In another example, suppose you assign 35425 to `int x`; you will not get expected result because 35425 is larger than short integer range.

The following assignments are invalid:

```

int=long;
int=float;
long=float
float=double

```

The following assignments are valid:

```

long=int
double=float
int=char

```

2.18 WRAPPING AROUND

When the value of variable goes beyond its limit, the compiler would not flag any error message. It just wraps around the value of a variable. For example, the range of unsigned integer is 0 to 65535. In this type, negative values and values greater than 65535 are compatible.

```
unsigned int x=65536;
```

The value assigned is greater by 1 than the range. Here, the compiler starts again from beginning after reaching the end of the range.

Consider the following example.

➤ 2.8 Write a program to demonstrate wrapping around.

```
void main()
{
    unsigned u=65537;
    clrscr();
    printf("\n u=%u",u);
}
```

OUTPUT:

u=1

Explanation:

In this program, the unsigned integer `u = 65537`; after reaching the last range 65535, the compiler starts from beginning. Hence, 1 is displayed. The value 65536 refers to 0 and 65537 to 1.

2.19 CONSTANT AND VOLATILE VARIABLES

2.19.1 | Constant Variable

If we want the value of a certain variable to remain the same or unchanged during the program execution, it can be done by declaring the variable as a constant. The keyword `const` is then prefixed before the declaration. It tells the compiler that the variable is a constant. Thus, constant declared variables are protected from modification.

Example:

```
const int m=10;
```

Where `const` is a keyword, `m` is variable name and 10 is a constant.

The compiler protects the value of 'm' from modification. The user cannot assign any value to 'm', but using pointer the value can be changed. When user attempts to modify the value of constant variable, the message 'cannot modify the constant object' will be displayed.

```
void main()
{
    const int num=12;
    ++num;
}
```

The above program code will not be executed and end with error message. Here, a constant variable attempted to modify its value.

2.19.2 | Volatile Variable

The volatile variables are those variables that can be changed at any time by other external program or the same program. The syntax is as follows:

```
volatile int d;
```

A program on the above concepts is given below.

➤ 2.9 Write a program to demonstrate wrapping around.

```
void main()
{
    volatile int x;
    volatile const int y=10;
    clrscr();
    printf("Enter an integer:");
    scanf("%d",&x);
    printf("Entered integer is :%d",x);
    x=3;
    printf("\nChanged value of x is :%d",x);
    printf("\nValue of y:%d",y);
    /* y=8; */
    getch();
}
```

OUTPUT:

```
Enter an integer:5
Entered integer is :5
Changed value of x is :3
Value of y:10
```

Explanation:

In the above example, the variable `x` is initialized with keyword `volatile` and `y` is with `volatile const`. Value of `x` can be changed but `volatile const` can not be changed. Make an attempt to include the comment statement in the program and see the response of the compiler. The error would be thrown by the compiler and message displayed on the screen would be 'cannot modify a const value'.

SUMMARY

After having studied the basics in the first chapter, the reader is now exposed to, in this second chapter, the additional fundamentals of C. These concepts are absolutely essential for writing programs. Readers are suggested go in depth of this chapter as it contains the fundamentals and basics. The reader is made aware of the following points:

- (i) Different types of characters like letters, digits, white space and special characters. Various delimiters used with C statements, keywords and identifies.
- (ii) Different constants, variables and data types.
- (iii) Rules for defining variables and initializing them. Also initialization of variables at run time (dynamic initialization) is studied.
- (iv) Type conversion of variable, type modifiers, wrapping around. Constant and volatile variables.

EXERCISES

I True or false:

1. Data means the combination of letters, digits, symbols.
2. In C, A to Z and a to z are treated as the same.
3. The `signed` is a C keyword.
4. The `new` is a C keyword.
5. 10.120 is a integer constant.
6. Identifiers are the name of variables, arrays, functions, and so on.
7. The `constant` in C is applicable to values which do not change during program execution.
8. 'A' is a character constant.
9. "India" is a string constant.
10. A variable is a data name used for storing data values.
11. A value of variable can be changed during the program execution.
12. A variable name can start with a digit.
13. The keyword can act as a variable name.
14. In C the variable name `sum` & `Sum` are the same.
15. `short int` and `int` are different from each other.
16. `int`, `short int` and `long int` belong to one category with different size and range of values.
17. `int` occupies 4 bytes in memory.
18. `const int a=10;` value of `a` cannot be changed during program execution.
19. `_abc` is a valid variable name.

II Select the appropriate option from the multiple choices given below:

1. A character variable can store only
 - (a) one character
 - (b) 20 character
 - (c) 254 character
 - (d) None of the above
2. A C variable cannot start with
 - (a) a number
 - (b) an alphabet
 - (c) a character
 - (d) None of the above
3. A short integer variable occupies memory of
 - (a) 1 bytes
 - (b) 2 bytes
 - (c) 4 bytes
 - (d) 8 bytes
4. C keywords are reserved words by
 - (a) a compiler
 - (b) an interpreter
 - (c) header file
 - (d) Both (b) and (c)
5. The declaration of C variable can be done
 - (a) anywhere in the program
 - (b) in declaration part
 - (c) in executable part
 - (d) at the end of a program
6. In C one statement can declare
 - (a) only one variables
 - (b) two variables
 - (c) 10 variables
 - (d) any number of variables
7. The word '`int`' is a
 - (a) keyword
 - (b) password
 - (c) header file
 - (d) None of the above
8. The variables are initialized using
 - (a) greater than (`>`)
 - (b) equal to (`=`)
 - (c) twice equal to (`==`)
 - (d) an increment operator (`++`)

9. An unsigned integer variable contains values
 - (a) greater or equal to zero
 - (b) less than zero
 - (c) only zeros
 - (d) Both (a) and (b)
10. The keyword 'const' keeps the value of a variable
 - (a) constant
 - (b) mutable
 - (c) variant
 - (d) None of the above
11. Identifiers are
 - (a) user-defined names
 - (b) reserved keywords
 - (c) C statements
 - (d) None of the above
12. In C every variable has
 - (a) a type
 - (b) a name
 - (c) a value
 - (d) a size
 - (e) all of the above
13. The range of character data type is
 - (a) -128 to 127
 - (b) 0 to 255
 - (c) 0 to 32767
 - (d) None of the above
14. An unsigned integer variable occupies memory
 - (a) 2 bytes
 - (b) 4 bytes
 - (c) 1 bytes
 - (d) 8 bytes
15. In C double type data needs memory of size
 - (a) 4 bytes
 - (b) 2 bytes
 - (c) 10 bytes
 - (d) None of the above
16. In C `main()` is a
 - (a) function
 - (b) user created function
 - (c) string function
 - (d) any number of variables
17. The word 'continue' is a
 - (a) keyword
 - (b) password
 - (c) header file
 - (d) None of the above
18. The volatile variables are those variables that remain/can be
 - (a) constant
 - (b) changed at any time
 - (c) Both of the above
 - (d) None of the above
19. In C the statements following `main()` are enclosed within
 - (a) `{ }`
 - (b) `( )`
 - (c) `<>`
 - (d) None of the above
20. In C the maximum value of unsigned character is
 - (a) 255
 - (b) 127
 - (c) 65535
 - (d) none of the above
21. The range of long signed integer is
 - (a) -2147483648 to 2147483647
 - (b) 0 to 255
 - (c) 0 to 4294967295
 - (d) None of the above
22. In C 'sizeof' is a/an
 - (a) variable
 - (b) operator
 - (c) keyword
 - (d) None of the above
23. Which is the incorrect variable name
 - (a) `else`
 - (b) `name`
 - (c) `age`
 - (d) `cha_r`
24. How many keywords are there in ANSI C?
 - (a) 32
 - (b) 33
 - (c) 42
 - (d) 15

54 ♦ Programming in C

25. Integer constants in C can be

- (a) positive
- (b) negative
- (c) positive or negative
- (d) None of the above

26. Which of the following statement is wrong?

- (a) `5+5=a;`
- (b) `ss=12.25;`
- (c) `st='m' * 'b';`
- (d) `is ='A'+10;`

27. The ANSI C standard allows a minimum of

- (a) 31 significant characters in identifier
- (b) 8 significant characters in identifier
- (c) 25 significant characters in identifier
- (d) unlimited characters

28. In C the first character of the variable should be

- (a) an integer
- (b) an alphabet
- (c) a floating number
- (d) None of the above

29. How many variables can be initialized at a time?

- (a) one
- (b) two

(c) five

(d) any number of variables

30. What is the output of the following program?

```
void main()
{
    unsigned long v=-1;
    clrscr();
    printf("\n %lu",v);
}
```

- (a) 4294967295
- (b) 0
- (c) -1
- (d) None of the above

31. What would be the values of variables c and u?

```
void main()
{
    char c=-127;
    unsigned char u=-127;
    clrscr();
    printf("\n c=%d u= %d",c,u);
}
```

- (a) `c=127 u=127`
- (b) `c=-127 u=129`
- (c) `c=127 u=128`
- (d) None of the above

III What will be the output/s of the following program/s?

1.

```
void main()
{
    char c=90;
    clrscr();
    printf("%c",c);
}
```

2.

```
void main()
{
    unsigned char c=65;
    clrscr();
    printf("%d %c %d ",c,c,c);
}
```

3.

```
void main()
{
    unsigned u=2147483647;
    long l=2147483647;
```

```
    clrscr();
    printf("\n u=%u l=%ld",
    u,l);
}
```

4.

```
void main()
{
    float a=3e-1,b=2e-2;
    clrscr();
    printf("a=%g b=%g",a,b);
}
```

5.

```
void main()
{
    int x,a=1e1, b=0;
    clrscr();
    b+=1e1;
    printf("a= %d b=%d",a,b);
}
```

IV Find the bug/s in the following program/s:

1.

```
void main()
{
    clrscr();
    printf("\n %d", 7/2);
    printf("\n%g", 7.0/2);
    printf("\n%g", float7/2);
    getche();
}
```

2.

```
void main()
{
    volatile d=15;
    const p=25;
    clrscr();
    printf("%d %d"d+10,p+1);
    getche();
}
```

3.

```
void main()
{
    unsigned int d=65535;
    unsigned char p=65;
    clrscr();
    printf("%c %c",d,p);
    getche();
}
```

4.

```
void main()
{
    float d=65535.43;
    double p=65789.987654;
    clrscr();
    printf("%d %d",d,p);
    getche();
}
```

5.

```
void main()
{
    float d=1234567.43;
    double p=987654321.1234567;
    clrscr();
    printf("\n%f %lf",d,p);
    printf("\n%d %d",size
of(d),size of(p));
    getche();
}
```

6.

```
#define
#define N= 90
main()
{
    int x=10,p;
    clrscr();
    p=x*N;
    printf("\n%d ", p);
    getche();
}
```

V Answer the following questions:

1. What are the different data types?
2. What are the differences between signed and unsigned data types?
3. What does we mean by a variable and constant?
4. Explain different types of constants in C.
5. What are the C keywords? Elaborate them.
6. List the rules for declaring a variable.
7. What are the identifiers?
8. Explain the methods for initialization of variables.
9. Explain constants and volatile variables.
10. Write about space requirement for variables of different data types.
11. What are the delimiters? Explain their uses.
12. Is 'main' a keyword? Explain.
13. List any three keywords with their use.
14. What is the difference between %f and %g control strings? Whether both can be used for representing float numbers?
15. What do you mean by type conversion? Why is it necessary?
16. What is wrapping around?
17. List the name of type modifiers.
18. What is dynamic initialization?

ANSWERS

I True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	T	2.	F	3.	T	4.	F	5.	F
6.	T	7.	T	8.	T	9.	T	10.	T
11.	T	12.	F	13.	F	14.	F	15.	F
16.	T	17.	F	18.	T	19.	T		

II Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	b	4.	a	5.	b
6.	d	7.	a	8.	b	9.	a	10.	a
11.	a	12.	b	13.	a	14.	a	15.	d
16.	b	17.	a	18.	b	19.	a	20.	a
21.	a	22.	c	23.	a	24.	a	25.	c
26.	a	27.	a	28.	b	29.	d	30.	a
31.	b								

III What will be the output/s of the following program/s?

Q.	Ans.	Q.	Ans.
1.	Z	4.	a=0.3 b=0.02
2.	65 A 65	5.	a=10 b=10
3.	u=65535 l=2147483647		

IV Find the bug/s in the following program/s:

Q.	Ans.
1.	Float must be enclosed with bracket, then answer will be 3 3.5 3.5
2.	Missing comma (,) in the printf(). Answer after correction 25 26
3.	First %C is to be replaced by %u and then answer will be 65535 A. However, with %C also program runs.

4. Control string provided is wrong. `%f` and `%lf` are needed. Answer would be 65535.429688 65789.987654. However, with `%d` & `%d` answers obtained would be wrong.
5. Space between size and of should be removed. The answer will be 1234567.375000 987654321.123457
4 8
6. Syntax error in first line `#define`. Remove `=` . Answer will be 900

CHAPTER

3

Operators and Expressions

Chapter Outline

- 3.1** Introduction
- 3.2** Operator Precedence
- 3.3** Associativity
- 3.4** Comma and Conditional Operator
- 3.5** Arithmetic Operators
- 3.6** Relational Operators
- 3.7** Assignment Operators and Expressions
- 3.8** Logical Operators
- 3.9** Bitwise Operators

3.1 INTRODUCTION

In our day-to-day life, we perform numerous operations on data with different operators. In order to perform different kinds of operations, C uses different operators. An operator indicates an operation to be performed on data that yields a new value. Using various operators in C, one can link the variable and constants. An operand is a data item on which operators perform operations. C is rich in use of different operators. C provides four classes of operators which are: (i) arithmetic, (ii) relational, (iii) logical and (iv) bitwise. Apart from these basic operators, C also supports additional operators. Basic operators and others along with their symbolic representation are shown in Table 3.1.

Table 3.1 *Types of operators*

Type of Operator	Symbolic Representation
Arithmetic operators	+, -, *, / and %
Relational operators	>, <, ==, >=, <= and !=
Logical operators	&&, and !
Increment and Decrement operators	++ and --
Assignment operator	=, *=, /=, %=, +=, -=, &=, ^=, =, <<=, >>=
Bitwise operators	&, , ^, >>, << and ~
Comma operator	,
Conditional operators	?:

3.2 OPERATOR PRECEDENCE

Precedence means priority. Every operator in C has assigned precedence (priority). An expression may contain a lot of operators. The operations on the operands are carried out according to the priority of the operators. The operators having higher priority are evaluated first and then lower priority.

- (i) For example, in arithmetic operators, the operators *, / and % have assigned highest priority and of similar precedence. The operators + and – have the lowest priority as compared to the above operators.

Example:

8+9*2–10

The operator * is the highest priority. Hence, the multiplication operation is performed first. The above expression becomes

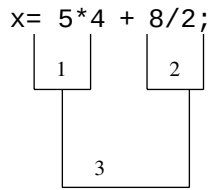
8+18 –10

In the above expression, + and – have the same priority. In such a situation, the left most operation is evaluated first. With this the above expression becomes

26–10

At last the subtraction operation is performed and answer of the expression will be as follows:
16

- (ii) When the operators of the same priority are found in the expression, precedence is given to the left most operators.

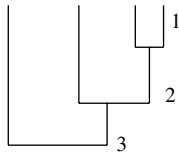
Example:

Here, $5 * 4$ is evaluated first, though $*$ and $/$ have the same priorities. The operator $*$ occurs before $/$ and hence evaluation starts from left. The answer for the above equation after evaluation becomes 24.

- (iii) If there is more number of parentheses in the expression, the innermost parenthesis will be solved first, followed by the second and lastly the outermost.

Example:

$(8 / (2 * (2 * 2)))$;

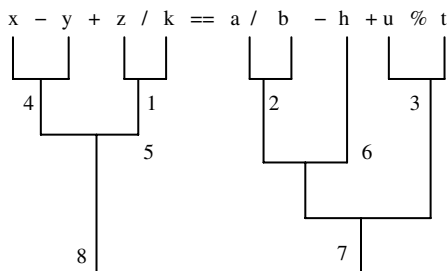


Here,

- Innermost bracket is evaluated first (i.e. $2 * 2 = 4$).
- Second innermost bracket is evaluated. 2 is multiplied with result of innermost bracket. The answer of $2 * 4 = 8$.
- The outer most bracket expression is evaluated at last. 8 is divided by 8 and gives result 1.

Example:

Show the steps for evaluation of the following equation.



The execution of the above equation is self-explanatory.

The following program is provided for understanding the concepts of precedence of operators.

➤ 3.1 Write a program to demonstrate the precedence of operators.

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

main ()
{
    int a=1,b=2,c=3,j;
    clrscr();
    j=a+b*c;
    printf ("\n j=%d",j);
    j=(a+b)*c;
    printf ("\n j=%d",j);
}
```

OUTPUT:

```
j=7
j=9
```

Explanation:

In this program, in the expression $j = a + b * c$, first multiplication operation is performed followed by addition operation; hence, value of j is 7. In the second expression, the parentheses give first priority to the addition operation. Hence, addition of a and b is performed first and followed by multiplication is performed.

The list of operators according to priority (Hierarchical) is given in Table 3.2

Table 3.2 Prioritywise list of operators

Operators	Operation	Associativity or Clubbing	Priority
()	Function call	Left to right	1st
[]	Array expression or square bracket		
->	Structure operator		
.	Structure operator	Right to left	2nd
+	Unary plus		
-	Unary minus		
++	Increment		
--	Decrement		
!	Not operator		
~	One's complement		
*	Pointer operator		
&	Address operator		
sizeof	Size of an object		
Type	Type cast		

(Continued)

Table 3.2 *Prioritywise list of operators (Continued)*

<i>Operators</i>	<i>Operation</i>	<i>Associativity or Clubbing</i>	<i>Priority</i>
*	Multiplication	Left to right	3rd
/	Division		
%	Modular division		
+	Addition (Binary plus)	Left to right	4th
-	Subtraction (Binary minus)		
<<	Left shift	Left to right	5th
>>	Right shift		
<	Less than	Left to right	6th
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7th
! =	Inequality (Not equal to)		
&	Bitwise AND	Left to right	8th
^	Bitwise XOR	Left to right	9th
	Bitwise OR	Left to right	10th
&&	Logical AND	Left to right	11th
	Logical OR	Left to right	12th
?:	Conditional operator	Right to left	13th
=, *=, /=, %= +=, -=, &=, ^=, = <=, >=	Assignment operators	Right to left	14th
,	Comma operator	Right to left	15th

3.3 ASSOCIATIVITY

When an expression contains operators with equal precedence then the associativity property decides which operation to be performed first. Associativity means the direction of execution. Associativity is of two types.

- (i) **Left to right:** In this type, expression, evaluation starts from left to right direction.

Example:

$12 * 4 / 8 \% 2$

In the above expression, all operators are having the same precedence, and so associativity rule is followed (i.e. direction of execution is from left to right).

= 48 / 8 % 2

= 6 % 2

= 0 (The above modulus operation provides remainder 0)

- (ii) **Right to left:** In this type, expression, evaluation starts from right to left direction.

Example:

a=b=c

In the above example, assignment operators are used. The value of c is assigned to b and then to a. Thus, evaluation, is from right to left.

3.4 COMMA AND CONDITIONAL OPERATOR

- (i) **Comma Operator (,):** The comma operator is used to separate two or more expressions. The comma operator has the lowest priority among all the operators. It is not essential to enclose the expressions with comma operators within the parentheses. For example, the following statements are valid.

Example:

a=2, b=4, c=a+b;

(a=2, b=4, c=a+b;)

- 3.2 Write a program to illustrate the use of comma (,) operator.

```
void main()
{
    clrscr();
    printf("Addition = %d\nSubtraction = %d", 2+3, 5-4);
}
```

OUTPUT:

Addition = 5
Subtraction = 1

Explanation:

In the above-mentioned program, the two equations are separated by commas. The results are obtained by solving the expressions separated by commas. The result obtained is printed through `printf()` statement.

- (ii) **Conditional Operator (?):** The conditional operator contains condition followed by two statements or values. The condition operator is also called the ternary operator. If the condition is true, the first statement is executed, otherwise the second statement is executed. Conditional operators (?) and (:) are sometimes called ternary operators because they take three arguments. The syntax of the conditional operator is as follows:

Syntax:

Condition ? (expression1) : (expression2);

Two expressions are separated by a colon. If the condition is true expression 1 gets evaluated, otherwise expression 2 gets evaluated. The condition is always written before the ? mark.

The below mentioned program illustrates the use of conditional operator.

- 3.3 Write a program to use the conditional operator with two values.

```
void main()
{
    clrscr();
    printf("Result = %d", 2==3 ? 4 : 5 );
}
```

OUTPUT:

Result = 5

Explanation:

In the above-given program, the condition $2=3$ is false. Hence, 5 is printed.

- 3.4 Write a program to use the conditional operator with two statements.

```
void main()
{
    clrscr();
    3>2 ? printf("True.") : printf("False.");
}
```

OUTPUT:

True.

Explanation:

In the above-given program, a full statement is used as the conditional operator. The condition $3>2$ is true. Hence, the first `printf()` statement is executed.

3.5 ARITHMETIC OPERATORS

There are two types of arithmetic operators. These are (i) binary operator and (ii) unary operator.

As shown in Figure 3.1, the unary and binary operators are defined in the following section and the ternary operator is described in Section 3.4. The conditional operator is nothing but the ternary operator.

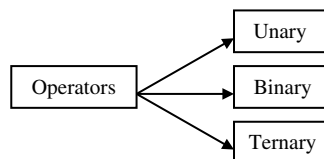


Figure 3.1 Operators

- (i) **Binary Operator:** Table 3.3 shows different arithmetic operators that are used in C. These operators are commonly used in most of the computer languages. These arithmetic operators are used for numerical calculations between the two constant values. They are also called Binary Arithmetic Operators. Binary operators are those operators which require two operands. The examples are also given in Table 3.3. In the program, variables are declared instead of constants.

C evaluates arithmetic operations as follows.

- (a) Division, multiplication and remainder operations are solved first. When an expression contains many operators such as multiplication, division and modular division operations, expression evaluation starts from left to right. Multiplication, division and modular division operators are having equal level of precedence.
- (b) The addition and subtraction operations are solved after division, multiplication and modular division operations. Evaluation starts from left to right. Addition and subtraction have equal level of precedence.

From Table 3.3, the modular division operator cannot be applied to `float` and `double` data types.

Table 3.3 *Arithmetic operators*

<i>Arithmetic Operators</i>	<i>Operator Explanation</i>	<i>Examples</i>
+	Addition	2+2=4
-	Subtraction	5-3=2
*	Multiplication	2*5=10
/	Division	10/2=5
%	Modular division	11%3=2 (Remainder 2)

- (ii) **Unary Operators:** The operators which require only one operand are called unary operators. Unary operators are increment operator (++), decrement (--) and minus (-). These operators and their descriptions are given in Table 3.4.

Table 3.4 *Unary arithmetic operators*

<i>Operator</i>	<i>Description or Action</i>
-	Minus
++	Increment
--	Decrement
&	Address operator
sizeof	Gives the size of an operator

- (a) **Minus (-):** Unary minus is used for indicating or changing the algebraic sign of a value.

Example:

```
int x=-50;
```

```
int y=-x;
```

assign the value of -50 to x and the value of -50 to y through x. The minus (-) sign used in this way is called the unary operator because it takes just one operand. There is no unary plus (+) in C. Even though, a value assigned with plus sign is valid, for example `int x=+50`, here, + is valid, but in practice this sign should not be attached in C.

- (b) **Increment (++) and Decrement (--) Operators:** The C compilers produce very fast and efficient object codes for increment and decrement operations. This code is better than generated by using the equivalent assignment statement. So, increment and decrement operators should be used whenever possible.

The operator `++` adds one to its operand, whereas the operator `--` subtracts one from its operand. For justification, `x=x+1` can be written as `x++`; and `x=x-1`; can be written as `x--`. Both these operators may either follow or precede the operand. That is `x=x+1`; can be represented as `x++`; or `++x`;

If `++` or `--` are used as a suffix to the variable name, then post-increment/decrement operations take place. Consider an example for understanding the `++` operator as a suffix to the variable.

```
x=20;
y=10;
z=x*y++;
```

In the above equation, the current value of `y` is used for the product. The result is 200, which is assigned to `'z'`. After multiplication the value of `y` is incremented by one.

If `++` or `--` are used as a prefix to the variable name, then pre increment/decrement operations take place. Consider an example for understanding `++` operator as a prefix to the variable.

```
x=20;
y=10;
z=x*++y;
```

In the above equation, the value of `y` is incremented and then multiplication is carried out. The result is 220, which is assigned to `'z'`. The following programs can be executed for verification of increment and decrement operations.

- 3.5 Write a program to show the effect of increment operator as a suffix.

```
void main()
{
    int a,z,x=10,y=20;
    clrscr();
    z=x*y++;
    a=x*y;
    printf("\n%d %d", z, a);
}
```

OUTPUT:

200 210

Explanation:

In the above program, the equation `z=x*y++` gives the result 200 because `'y'` does not get incremented. After multiplication, `'y'` incremented to 21. The second equation gives the result, i.e. 210. This can be verified by executing the above program.

- 3.6 Write a program to show the effect of increment operator as a prefix.

```
void main()
{
    int a,z,x=10,y=20;
    clrscr();
    z=x*++y;
```

```

    a=x*y;
    printf("\n%d %d",z,a);
}

```

OUTPUT:

210 210

Explanation:

In the above program, 'y' gets first incremented and the equations are solved. Here, both the equations give the same result, i.e. 210.

(c) sizeof and '&' Operator :

The **sizeof** operator gives the bytes occupied by a variable, i.e. the size in terms of bytes required in memory to store the value. The number of bytes occupied varies from variable to variable depending upon its data types.

The '&' operator prints address of the variable in the memory. The below mentioned example illustrates the use of these operators.

- 3.7 Write a program to use '&' and **sizeof** operator and determine the size of integer and float variables.

```

void main()
{
    int x=2;
    float y=2;
    clrscr();
    printf("\n sizeof(x)=%d bytes",sizeof(x));
    printf("\n sizeof(y)=%d bytes",sizeof(y));
    printf("\n Address of x=%u and y=%u",&x,&y);
}

```

OUTPUT:

sizeof(x)=2
 sizeof(y)=4
 Address of x=4066 and y=25096

Explanation:

In the above example, variables x and y are declared and initialized. The variable x is an integer and y is a float data type. Using **sizeof()** and '&' operator, their sizes and addresses can be displayed.

3.6 RELATIONAL OPERATORS

These operators are used to distinguish between two values depending on their relations. These operators provide the relationship between two expressions. If the relation is true then it returns a value 1 otherwise 0 for false. The relational operators together with their descriptions, example and return value are described in Table 3.5.

Table 3.5 *Relational operators*

<i>Operator</i>	<i>Description or Action</i>	<i>Example</i>	<i>Return Value</i>
>	Greater than	5>4	1
<	Less than	10<9	0
<=	Less than or equal to	10<=10	1
>=	Greater than equal to	11>=5	1
==	Equal to	2==3	0
!=	Not equal to	3!=3	0

The relational operator's symbols are easy for understanding. They are self-explanatory. However, for the benefit of the readers a program is illustrated below.

➤ 3.8 Write a program to use various relational operators and display their return values.

```
void main()
{
    clrscr();
    printf("\nCondition : Return Values\n" );
    printf("\n10!=10 : %5d", 10!=10);
    printf("\n10==10 : %5d", 10==10);
    printf("\n10>=10 : %5d", 10>=10);
    printf("\n10<=100 : %5d", 10<=100);
    printf("\n10!=9 : %5d", 10!=9);
}
```

OUTPUT:

```
Condition: Return Values
10!=10 :      0
10==10 :      1
10>=10 :      1
10<=100:      1
10!=9  :      1
```

Explanation:

In the above program, the true conditions return 1 and false 0. In this example, the first condition is false and remaining are true. Hence, the return value for the first is 0 and for remaining it is 1.

Assume a program to perform certain steps on the basis of a condition. If $A > 5$, then some action will be performed. The example, which is illustrated below, uses the relational operator.

3.7 ASSIGNMENT OPERATORS AND EXPRESSIONS

Assigning a value to a variable is very simple. For example, `int x=5;` here, 5 is assigned to x and this is carried out by the operator `=`. The equal sign (`=`) is used for assignment and hence, the name assignment operator has been given. The list of assignment operators is given in Table 3.6.

Table 3.6 Assignment operators

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

Consider the following example: `k=k+3`; in this expression, the variable on the left side of `=` is repeated on the right. The same can be written as follows:

```
k+=3;
```

The operators `=` and `+=` are called assignment operators. The binary operators which need two operands to their either side are surely have a corresponding assignment operator **operand** `=` to the left side.

Example:

```
j*=x+1;
or
j=j*(x+1);
```

- 3.9 Write a program to determine the value of 'b' depending on the inputted value of 'a'. The variable 'a' is used with the conditional operator.

```
void main()
{
    int a,b;
    clrscr();
    printf("Enter Any Integer either above 5 or below 5 :-");
    scanf("%d", &a);
    b=(a>5 ? 3 : 4);
    printf("Calculated Value of b is :- %d",b);
}
```

OUTPUT:

```
Enter Any Integer either above 5 or below 5:- 6
Calculated Value of b is:- 3
OR
Enter Any Integer either above 5 or below 5:- 3
Calculated Value of b is:- 4
```

Explanation:

On the execution of the above program, the value of `b` will be 3 if the value of `a` is greater than 5. Otherwise, it will be 4 for any number which is less than 5. Similarly, for the other operators, programs are as follows.

- 3.10 Determine the value of 'b' using the conditional statement.

```
void main()
{
    int a,b;
    clrscr();
```

```

printf("Enter Any Integer whose value is 5 or any other:-");
scanf("%d", &a);
b=(a==5 ? 3: 4);
printf("Calculated Value of b is :- %d\n",b);
}

```

OUTPUT:

```

Enter Any Integer whose value is 5 or any other:- 3
Calculated Value of b is:- 4

```

Explanation:

In the above-given program, the value of the variable 'a' is entered. The value is compared with 5. If a is equal to 5, 3 is assigned to b otherwise 4. In this example, 3 is entered which is not equal to five. Hence, 4 is assigned to b.

The following program illustrates the use of various relational operators.

- 3.11 Write a program to read three variables x, y and z. Use conditional statements and evaluate values of variables a, b and c. Perform the sum with two sets of variables. Check the sums for equality and print different messages.

```

void main()
{
    int x,y,z,a,b,c,m,n;
    clrscr();
    printf("Enter Values of x, y, z :-");
    scanf("%d %d %d", &x,&y,&z);
    a=(x>=5 ? 3 : 4);
    printf("\n Calculated value of a is :- %d", a);
    b=(y<=8 ? 10 : 9);
    printf("\n Calculated value of b is :- %d", b);
    c=(z==10 ? 20 : 30);
    printf("\n Calculated value of c is :- %d", c);
    m=x+y+z;
    n=a+b+c;
    printf("\n Addition of x,y,z is %d (m)", m);
    printf("\n Addition of a,b,c is %d (n)", n);
    printf("\n%s", m!=n ? "m & n NOT EQUAL" : "m & n ARE EQUAL");
}

```

OUTPUT:

```

Enter Values of x, y, z:- 5 2 7
Calculated value of a is:- 3
Calculated value of b is:- 10
Calculated value of c is:- 30
Addition of x,y,z is 14 (m)
Addition of a,b,c is 43 (n)
m & n NOT EQUAL

```

Explanation:

In the above given program, three integers are entered through the keyboard (x, y and z). Using conditional statements, values of a, b and c are obtained. The sum of x, y and z is stored in 'm' and the sum of a, b and c is stored in 'n'. The variables 'm' and 'n' are compared and appropriate messages are displayed.

3.8 LOGICAL OPERATORS

The logical relationship between the two expressions is tested with logical operators. Using these operators, two expressions can be joined. After checking the conditions, it provides logical true (1) or false (0) status. The operands could be constants, variables and expressions. Table 3.7 describes the three logical operators together with examples and their return values.

Table 3.7 Logical operators

Operator	Description or Action	Example	Return Value
&&	Logical AND	5>3 && 5<10	1
	Logical OR	8>5 8<2	1
!	Logical NOT	8!=8	0

From Table 3.7, following rules can be followed for logical operators:

- (i) The logical AND (&&) operator provides true result when both expressions are true, otherwise 0.
- (ii) The logical OR (||) operator provides true result when one of the expressions is true, otherwise 0.
- (iii) The logical NOT (!) provides 0 if the condition is true, otherwise 1.

➤ 3.12 Illustrate the use of logical operators.

```
void main()
{
    clrscr();
    printf("\nCondition : Return Values\n" );
    printf("\n5>3 && 5<10 : %5d",5>3 && 5<10);
    printf("\n 8>5 || 8<2 : %5d",8>5 || 8<2);
    printf("\n !(8==8) : %5d",!(8==8));
}
```

OUTPUT:

```
Condition : Return Values
5>3 && 5<10 : 1
8>5 || 8<2 : 1
!(8==8) : 0
```

Explanation:

In the above example, the first and second conditions are true. In the first condition, 5 is greater than 3 and smaller than 10; hence, output is 1. In the second condition, 8 is greater than 5 and due to OR operation its output is 1. The third condition is wrong. Hence, result returned will be 0.

- 3.13 Write a program to print logic 1 if input character is capital otherwise 0.

```
void main()
{
    char x;
    int y;
    clrscr();
    printf("\nEnter a Character :");
    scanf("%c", &x);
    y=(x>=65 && x<=90 ? 1 : 0);
    printf("Y :%d",y);
}
```

OUTPUT:

```
Enter a Character : A
Y: 1
Enter a Character : a
Y: 0
```

Explanation:

In the above-given program, a character is entered. Using logical operator AND, entered character's ASCII value is checked. If it is in between 65 and 90, the result displayed will be 1 otherwise 0. The AND operator joins two conditions. If the condition is true, 1 is assigned to y otherwise 0.

- 3.14 Write a program to display logic 0 if one reads a character through keyboard otherwise 1. (ASCII values for 0 to 9 are 48 to 57, respectively.)

```
void main()
{
    int y;
    char x;
    clrscr();
    printf("\n Enter The Character or Integer :");
    scanf("%c", &x);
    y=(x>=48 && x<=57 ? 1 : 0);
    Printf("\nValue of Y =%d",y);
}
```

OUTPUT:

```
Enter The Character or Integer : A
Value of Y = 0
OR
Enter The Character or Integer : 1
Value of Y = 1
```

Explanation:

The above-given program is the same as the previous one. Here, ASCII range 48 to 57 is used. Equivalent of these values are 0 to 9 digits, respectively. If the entered number is in between 0 to 9, the compiler returns 1 otherwise 0.

- 3.15 Write a program to display 1 if inputted number is between 1 and 100 otherwise 0. Use the logical AND (&&) operator.

```
void main()
{
    int x,z;
    clrscr();
    printf("Enter numbers :");
    scanf("%d", &x);
    z=(x>=1 && x<=100 ? 1 : 0);
    printf("Z=%d", z);
}
```

OUTPUT:

```
Enter numbers : 5
Z = 1
Enter numbers : 101
Z = 0
```

Explanation:

The logical operator checks the entered value whether it is in between 1 and 100. If the condition is true, 1 is assigned to z otherwise 0.

- 3.16 Write a program to display 1 if inputted number is either 1 or 100 otherwise 0. Use the logical OR (||) operator.

```
main()
{
    int x,z;
    clrscr();
    printf("Enter numbers :");
    scanf("%d", &x);
    z=(x==1 || x==100 ? 1 : 0);
    printf("Z=%d", z);
}
```

OUTPUT:

```
Enter numbers : 1
Z = 1
Enter numbers : 100
Z = 1
Enter numbers : 101
Z = 0
```

Explanation:

In the above-given program, the OR operator checks two conditions. If one of the conditions satisfies, 1 is assigned to z otherwise 0.

- 3.17 Write a program to display 1 if the inputted number is except 100 otherwise 0. Use the logical NOT operator (!).

```
void main()
{
    int x,z;
    clrscr();
    printf("Enter number :");
    scanf("%d", &x);
    z=(x!=100 ? 1 : 0);
    printf("d :%d", z);
}
```

OUTPUT:

```
Enter number : 100
d : 0
Enter number : 10
d : 1
```

Explanation:

In the above-given program ! NOT operator is used with the conditional operator. The value of x is also entered. Here, if the value of x is other than 100, then 1 is assigned to Z otherwise 0.

3.9 BITWISE OPERATORS

C supports a set of bitwise operators for bit manipulation as listed in Table 3.8. C supports six bit operators. These operators can operate only on integer operands such as `int`, `char`, `short`, `long`.

Table 3.8 Bitwise operators

Operators	Meaning
>>	Right shift
<<	Left shift
^	Bitwise XOR (exclusive OR)
~	One's complement
&	Bitwise AND
	Bitwise OR

- 3.18 Write a program to shift inputted data by two bits right.

```
void main()
{
    int x,y;
    clrscr();
    printf("Read The Integer from keyboard (x) :-");
    scanf("%d", &x);
}
```

```

x>>=2;
y=x;
printf("The Right shifted data is = %d",y);
}

```

OUTPUT:

Read The Integer from keyboard (x) :- 8
The Right shifted data is = 2

Before the execution of the program: The number entered through the keyboard is 8 and its corresponding binary number is 1 0 0 0.

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

After execution of the program: As per the above-given program, the inputted data x is to be shifted by 2 bits right side. The answer in binary bits would be as follows:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

Shifting two bits right means the inputted number is to be divided by 2^s where s is the number of shifts i.e. in short $y=n/2^s$, where n = number and s = the number of position to be shift.

As per the program cited above, $Y=8/2^2 = 2$.

Similarly, a program for shifting to the left can be written as follows:

➤ 3.19 Write a program to shift inputted data by three bits left.

```

void main()
{
    int x,y;
    clrscr();
    printf("Read The Integer from keyboard (x) :-");
    scanf("%d", &x);
    x<<=3;
    y=x;
    printf("The Right shifted data is = %d",y);
}

```

OUTPUT:

Read The Integer from keyboard (x) :- 2
The Left shifted data is = 16

Before the execution of the program: The number entered through the keyboard is 2 and its corresponding binary number is 1 0. The bits will be as follows:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

After execution of the program: As per the above-given program, the inputted data x is to be shifted by 3 bits left side. The answer in the binary bits would be as follows:

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

The corresponding decimal number is 16, i.e. answer should be 16.

Shifting three bits left means the number is multiplied by 8; in short $y = n * 2^s$ where n = number and s = the number of position to be shifted.

As per the program given above,

$$Y = 2 * 2^3 = 16.$$

➤ 3.20 Write a program to use bitwise AND operator between the two integers and display the results.

```
void main()
{
    int a,b,c;
    clrscr();
    printf("Read The Integers from keyboard (a & b) :-");
    scanf("%d %d", &a,&b);
    c=a & b;
    printf("The Answer after ANDing is (C)= %d",c);
}
```

OUTPUT:

```
Read The Integers from keyboard (a & b) : 8 4
The Answer after ANDing is (C) = 0
```

Binary equivalent of 8 is

Before execution:

a=8

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0

b=4

Binary equivalent of 4 is

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0

After execution

c=0

Binary equivalent of 0 is

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0

OR

Read The Integers through keyboard (a & b) : 8 8

The Answer after ANDing is (C) = 8

Before execution

a=8

Binary equivalent of 8

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

Before execution

b=8

Binary equivalent of 8

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

After execution

c=8

Binary equivalent of 8

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

The table for AND operation (Table 3.9) is as follows and can be used in future for reference. Similarly, the table for OR operator (Table 3.10) can be used as follows.

Table 3.9 Table for AND

Inputs		Output
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.10 Table for OR operator

Inputs		Output
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

➤ 3.21 Write a program to operate OR operation on two integers and display the result.

```
void main()
{
    int a,b,c;
    clrscr();
    printf("Read The Integer from keyboard (a & b) :-");
    scanf("%d %d", &a,&b);
    c=a | b;
    printf("The Oring operation bewteen a & b in c = %d",c);
    getch();
}
```

OUTPUT:

Read The Integer from keyboard (a & b) :- 8 4
The Oring operation between a & b in c = 12

Before execution

a=8

Binary equivalent of 8

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

Before execution

b=4

Binary equivalent of 4

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

After execution

c=12

Binary equivalent of 12

0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

The table for Exclusive OR (XOR) is as follows (Table 3.11).

Table 3.11 Table of exclusive OR

Inputs		Output
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

➤ 3.22 Write a program with Exclusive OR Operation between the two integers and display the result.

```
void main()
{
    int a,b,c;
    clrscr();
    printf("Read The Integers from keyboard (a & b) :-");
    scanf("%d %d", &a,&b);
    c=a^b;
```

```
printf("The data after Exclusive OR operation is in c= %d",c);
getche();
}
```

OUTPUT:

Read The Integers from keyboard (a & b) : 8 2
 The data after Exclusive OR operation is in c =10

Before execution

a=8

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

Before execution

b=2

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

After execution

c=10

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	0	

The table for bitwise complement-operator (Inverter Logic) is as follows (Table 3.12).

Table 3.12 Table of inverter logic

Input (X)	Output (X)
0	1
1	0

The operator ~ is used for inverting the bits with this operator 0 becomes 1 & 1 becomes 0.

➤ 3.23 Write a program to show the effect of ~ operator.

```
void main()
{
    unsigned int v=0;
    clrscr();
    printf("\n %u", ~v);
}
```

OUTPUT:

65535

SUMMARY

You have now studied the various operators such as arithmetic, logical and relational which are essential for writing and executing programs. The precedence and associativity of the operators in the arithmetic operations are also furnished in the form of a table. The conditional and comma operators and programs on them are also described in this chapter. You are made aware about the logical operators OR, AND and NOT. Full descriptions of bitwise operators have been illustrated. Numerous simple examples have been provided to the users to understand the various operators. The reader is expected to write more programs on this chapter.

EXERCISES

I Select the appropriate option from the multiple choices given below:

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

```
void main()
{
    int ans=2;
    int m=10;
    int k;
    k=!((ans<2) && (m>2));
    printf("\n %d", k);
}
```

- (a) 1
- (b) 0
- (c) -1
- (d) 2

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

```
void main()
{
    int m, j=3, k;
    m=2*j/2;
    k=2*(j/2);
    clrscr();
    printf("\n m=%d k=%d", m, k);
}
```

- (a) m=3 k=2
- (b) m=3 k=3
- (c) m=2 k=3
- (d) m=2 k=2

3. What will be the value of x, y and z after the execution of the following program?

```
void main()
{
    int x, y, z;
    y=2;
    x=2;
    x=2*(y++);
    z=2*(++y);
    printf("\n x=%d y=%d z=%d", x, y, z);
}
```

- (a) x=4 y=4 z=8
- (b) x=6 y=4 z=8
- (c) x=2 y=4 z=8
- (d) x=4 y=4 z=4

4. What will be the value of 'x' after the execution of the following program?

```
void main()
{
    int x=!0*10;
}
```

- (a) 10
- (b) 1
- (c) 0
- (d) None of the above

5. What is the value of !0?

- (a) 1
- (b) 0
- (c) -1
- (d) None of the above

6. Hierarchy decides which operator

- (a) is used first
- (b) is the most important
- (c) operates on large numbers
- (d) None of the above

7. What will be the output after the execution of the following program?

```
void main()
{
    int k=8;
    printf("k=%d", k++-k++);
}
```

- (a) k=-1;
- (b) k=0;
- (c) k=8;
- (d) k=9;

8. What will be the value of b after the execution of the following program?

```
void main()
{
    int b,k=8;
    b=(k++-k++-k++,k++);
}
```

- (a) b=11;
- (b) b=12;
- (c) b=7;
- (d) b=9;

9. The '&' operator displays

- (a) address of the variable
- (b) value of the variable
- (c) Both (a) and (b)
- (d) None of the above

10. Addition of two numbers is performed using

- (a) arithmetic operator
- (b) logical operator
- (c) unary operator
- (d) comma operator

11. What is the remainder of 8% 10?

- (a) 8
- (b) 2
- (c) 1
- (d) 0

12. The result of the expression (10/3)*3+5%3 is

- (a) 11
- (b) 10
- (c) 8
- (d) 1

13. The result of expression (23*2) % (int) 5.5 is

- (a) 2
- (b) 1
- (c) 3
- (d) 0

14. The result of 16>>2 is

- (a) 4
- (b) 8
- (c) 2
- (d) 5

15. The result of 5&&2 is

- (a) 0
- (b) 1
- (c) 2
- (d) 5

16. The value of c after the execution of the program will be

```
void main()
{
    int a,b,c;
    a=9;
    b=10;
    c=(b<a || b>a);
    clrscr();
    printf("\n c=%d",c);
}
```

OUTPUT:

- (a) c=1
- (b) c=0
- (c) c=-1
- (d) None of the above

II Attempt the following programming exercises:

- Write a program to shift the entered number by three bits left and display the result.
- Write a program to shift the entered number by five bits right and display the result.
- Write a program to mask the most significant digit of the entered number. Use AND operator.
- Write a program to enter two numbers and find the smallest out of them. Use conditional operator.
- Write a program to enter a number and carry out modular division operation by 2, 3 and 4 and display the remainders.
- Attempt the program (5) with division operation and find the quotients.
- Write a program to enter an integer number and display its equivalent values in octal and hexadecimal.
- Write a program to convert hexadecimal to decimal numbers. Enter the numbers such as 0x1c, 0x18, 0xbc, 0xcd.
- Write a program to find the average temperature of five sunny days. Assume the temperature in Celsius.
- Write a program to enter two numbers. Make a comparison between them with a conditional operator. If the first number is greater than the second perform multiplication otherwise division operation.
- Write a program to calculate the total cost of the vehicle by adding basic cost with (i) excise duty (15%) (ii) sales tax (10%) (c) octroi (5%) and (d) road tax (1%). Input the basic cost.
- Write a program to display ASCII equivalents of
 - 'A', 'B', 'C' and 'a', 'b', 'c'.
 - 'a'-'C', 'b'-'A' and 'c'-'B'.
 - 'a'+'c', 'b'*'a' and 'c'+12.
- Write a program to enter a number that should be less than 100 and greater than 9. Display the number in reverse order using modular division and division operation.
- Write a program to enter a four-digit number. Display the digits of the number in the reverse order using modular division and division operation. Perform addition and multiplication of digits.
- Write a program to display numbers from 0 to 9. Use ASCII range 48 to 59 and control string %C.
- Write a program to evaluate the following expressions and display their results.
 - $x_2 + (2 * x3) * (2 * x)$
 - $x1 + y2 + z3$
 assume variables are integers.
- Write a program to print whether the number entered is even or odd. Use conditional operator.

III Answer the following questions:

- Explain different types of operators supported by C.
- What are the uses of comma (,) and conditional (?) operators?
- What are unary operators and describe their uses?
- Describe logical operators with their return values.
- Distinguish between logical and bitwise operators.
- What are the relational operators?
- What is the difference between '=' and '=='?
- What are the symbols used for (a) OR, (b) AND, (c) XOR and (d) NOT operations?
- Explain the precedence of operators in arithmetic operations?
- List the operators from higher priority to least priority.
- What is the difference between %f and %g?
- What are the differences between division and modular division operations?
- What are the ASCII codes? List the codes for digits 1 to 9, A to Z and a to z.

14. Explain different types of assignment operators. 16. What are the differences between precedence and associativity?
15. Explain properties of operators.

ANSWERS

I Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	a	4.	a	5.	a
6.	a	7.	a	8.	a	9.	a	10.	a
11.	a	12.	a	13.	b	14.	a	15.	b
16.	a								

CHAPTER

4

Input and Output in C

Chapter Outline

- 4.1** Introduction
- 4.2** Formatted Functions
- 4.3** Flags, Widths and Precision with Format String
- 4.4** Unformatted Functions
- 4.5** Commonly Used Library Functions
- 4.6** Strong Points for Understandability

4.1 INTRODUCTION

Reading input data, processing it and displaying the results are the three tasks of any program. The data is read from the input device such as a keyboard. Operations on the data are performed on the basis of the operators, and the result is displayed on the screen. All the three tasks are important and none of them can be ignored.

There are two ways to accept the data. In one method, a data value is assigned to the variable with an assignment statement. The programmer writes assignment statements in most of the programs. The assignment statements are mentioned throughout the book. Examples of an assignment statement are as follows:

(i) `int year=2005;` (ii) `char letter='a';` (iii) `long int x=123456.`

Another way of accepting the data is with functions. In C the input and output functions can be used for inputting the data and for getting the results, respectively. To perform these tasks in a user friendly manner, C has a number of input and output functions. When a program needs data, it takes the data through the input functions and sends results obtained through the output functions. Thus, the input/output functions are the link between the user and terminals.

There are a number of I/O functions in C, based on the data types. The input/output functions are classified in two types:

- (i) Formatted functions
- (ii) Unformatted functions

With formatted functions, the input or output is formatted as per our requirement. The readability in easy way is possible with formatted functions. For example, with formatted functions one can decide how the result should appear or display on the screen. The result can be shown on the second line or it can appear after leaving some space or if the result is a real number then decisions on the number of digits before and after decimal point, etc. will be taken care in formatted functions. All I/O functions are defined as `stdio.h` header file, which can be initialized at the starting of a program; that is this header file should be included in the program at the beginning. However, formatting is not possible with unformatted functions. Various functions of these categories are listed in Figure 4.1.

Streams perform all input and output operations. The streams are nothing but a sequence of bytes. In input operations, the bytes (data) flow from input device such as keyboard, a disc drive or network connection to main memory. Similarly, in output operation bytes flow from main memory to output devices such as monitor, printer, disc drive, network connection.

When a program performs input and output operations, the streams are connected to the program automatically. The operating system always allows streams to redirect to other devices. While performing these operations, if any error occurs, it will be reported on the screen by the third stream called standard error stream.

Further elaboration on formatted functions is as follows.

- (i) **Formatted Functions:** The formatted input/output functions read and write, respectively, all types of data values. They require format string to produce formatted results. Hence, they can be used for both reading and writing of all data values. The formatted functions return values after execution. The return value is equal to the number of variables successfully read/written. Using this value, the user can find out the error that occurred during reading or writing of data. Using this function, the given numeric data can be represented in `float`, `integer` and `double` to possible available limits of the language.

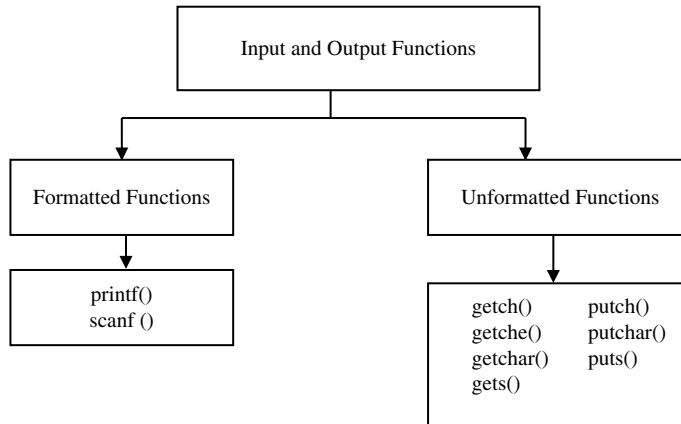


Figure 4.1 Formatted and unformatted functions

The syntax of input function for inputting the data such as `scanf()` is as follows:

```
scanf("control string", argu1, argu2, ...);
```

Precisely, if we write `scanf()` as `scanf("%d",&x);` where `%d` is a control string which is nothing but conversion specification and it is to be placed within double quote. The other part is the argument and a sign `&` (ampersand) must precede it. Arguments are the identifiers.

For displaying the result, `printf()` formatted function is used. In Section 4.2, `printf()` and `scanf()` are discussed in depth.

- (ii) **Unformatted Functions:** The unformatted input/output functions work only with character data type. They do not require format conversion symbol for formatting of data types because they work only with character data type. There is no need to convert data. In case, values of other data types are passed to these functions, they are treated as character data.

4.2 FORMATTED FUNCTIONS

- (i) The `printf()` statement: The formatted output as per the programmers requirement is displayed on the screen with `printf()`. The list of variables can be indicated in the `printf()`. The values of variables are printed according to the sequence mentioned in `printf()`. The `printf()` function prints all types of data values to the console. It translates internal values to characters. It requires format conversion symbol or format string and variable names to print the data. The format string symbol and variable names should be the same in number and type. The syntax of `printf()` statement is as follows:

```
printf("Control string",variable1,variable2,...variable n);
```

The control string specifies the field format such as `%d`, `%s`, `%g`, `%f`, and variables as taken by the programmer.

The following are a few examples of the `printf()` function.

Example:

```
void main()
{
    int x=2;
    float y=2.2;
    char z='C';
    printf("%d %f %c", x, y, z);
}
```

OUTPUT:

2 2.2000 C

In the above program, `%d` corresponds to 'x' variable, `%f` to y and `%C` to 'z'. The conversion symbol given by the user helps the `printf()` to identify the data type of a given variable. In case a mismatch occurs, the value of a variable is converted according to the conversion symbol given.

Example:

```
int main()
{
    int y=65;
    clrscr();
    printf("%c %d", y, y );
    return 0;
}
```

OUTPUT:

A 65

In the above example, the integer variable 'y' contains the value of 65. The variable 'y' is printed using two-conversion symbols, integer and character. As shown in the output, `%C` converts numeric 65 value to its corresponding character A. `%d` prints the value 65, as it is, because the variable is of integer type. Sometimes, if no conversion is possible between two data types, some garbage value is printed.

Example:

```
int main()
{
    int y=7;
    clrscr();
    printf("%f", y );
    return 0;
}
```

In the above example, it is attempted in the `printf()` statement to print the integer value as float value providing `%f` as a conversion symbol. This is not the proper way. While compiling time no error occurs, but after execution the `printf()` function will produce an error message 'floating points formats not linked'.

The format string is nothing but a string that begins and ends with the double quote. The `printf()` statement is used to display the data on console or stdout (standard output device). The format string is a combination of two types of character objects. They are plain character and conversion specification.

4.3 FLAGS, WIDTHS AND PRECISION WITH FORMAT STRING

The plain characters are straightforward and are used to write data on the screen. On the other hand, the conversion specification retrieves arguments from the list of arguments and apply different formatting to them. All format specification starts with `%` and a format specification letter after this symbol. It indicates the type of data and its format. In case the format string does not match the corresponding variable, the result will not be correct.

Flags: Flags are used for output justification, numeric signs, decimal points, trailing zeros. The flag `(-)` left justifies the result. If it is not given, the default is right justification. The plus `(+)` signed conversion result always starts with a plus `(+)` or a minus `(-)` sign.

Width specifier: It sets the minimum field width for an output value. Width can be specified through a decimal point or using an asterisk `'*'`.

Few programs are provided on width requirement.

- 4.1 Write a program to demonstrate the use of width specifier.

```
void main()
{
    clrscr();
    printf("\n%.2s", "abcdef");
    printf("\n%.3s", "abcdef");
    printf("\n%.4s", "abcdef");
}
```

OUTPUT:

```
ab
abc
abcd
```

Explanation:

Observe the above program carefully and watch the width specified along with conversion specification character `%S`. Although the actual string length is six characters, the number of printed characters as per `printf()` statements is 2, 3 and 4, respectively.

- 4.2 Write a program to demonstrate the use of width specifier.

```
void main()
{
    int x=55,y=33;
```

```
clrscr();
printf("\n %3d",x-y);
printf("\n %6d",x-y);
}
```

OUTPUT:

```
22
    22
```

Explanation:

In this program, in the first `printf()` statement width is given 3 and in second width is given 6. Hence, the results are displayed at different positions on the screen.

- 4.3 Write a program to demonstrate the use of '*' for formatting.

```
void main()
{
    int x=55,y=33;
    clrscr();
    printf("\n %*d",15,x-y);
    printf("\n %*d",5,x-y);
}
```

OUTPUT:

```
    22
    22
```

Explanation:

In this program, '*' is used along with format string or conversion specification character %d. An extra parameter is required to mention or set the starting column for printing. This value is given along the set of variables. You can observe in the `printf()` statements the values 15 and 5 that indicate the position from where printing on screen begins.

Precision specifiers: Precise results on the screen can be obtained. The precision specifier always starts with a period or a dot in order to separate it from any preceding width specifiers.

Consider the following program:

- 4.4 Write a program to demonstrate the use of precision specifiers.

```
void main()
{
    float g=123.456789;
    clrscr();
    printf("\n %.1f",g);
    printf("\n %.2f",g);
    printf("\n %.3f",g);
    printf("\n %.4f",g);
}
```

OUTPUT:

```
123.5
123.46
123.457
123.4568
```

Explanation:

In the above program, the precision value is specified before the format string. Fractional part after decimal point can be precisely shown in various `printf()` statements. The output shows these numbers.

From the above examples, it is now clear that outputs can be shown in different formats. Table 4.1 describes the various formats for presenting various outputs.

Table 4.1 *Formats for various outputs*

Sr. No.	Format	Meaning	Explanation
1	%wd	Format for integer output	w is width in integer and d is conversion specification
2	%w.cf	Format for float numbers	w is width in integer, c specifies the number of digits after decimal point and f specifies the conversion specification
3	%w.cs	Format for string output	w is width for total characters, c are used for displaying leading blanks and s specifies conversion specification

A programming example is provided on the above formats for readers understanding.

- 4.5 Write a program to display the integers, float point numbers and string with different formats as explained above.

```
void main()
{
    clrscr();
    printf("\n%5d", 12);
    printf("\n%5d", 123);
    printf("\n%5d", 1234);
    printf("\n %4.5f", 6.12);
    printf("\n %4.6f", 16.12);
    printf("\n %4.7f", 167.12);
    printf("\n %4.8f", 1678.12);
    printf("\n %8s", "Amitkumar");
    printf("\n %8.2s", "Amitkumar");
    getch();
}
```

OUTPUT:

```

12
123
1234
6.12000
16.120000
167.1200000
1678.12000000
Amitkumar
    Am

```

- (ii) The `scanf()` statement: The `scanf()` statement reads all types of data values. It is used for runtime assignment of variables. The `scanf()` statement also requires conversion symbol to identify the data to be read during the execution of the program. The `scanf()` statement stops functioning when some input entered does not match with format string. The syntax of the `scanf()` statement is the same as `printf()` except they work exactly opposite of each other.

Syntax:

The syntax of the input function for inputting the data is `scanf()`.

Example:

```
scanf("control string", address of variable 1, address of variable 2, ----);
```

The control string has to be enclosed within double quotes. It specifies the format specifier, such as `%d` for integer, `%f` for float, `%c` for character, etc. and the data are to be invoked by arguments, such as address of variable 1, address of variable 2, etc.

Precisely, we write `scanf()` as `scanf("%d", &x);`

Here, “`%d`” is the format specifier in the control string, which is nothing but the conversion specification and it is to be placed within double quotes. The other part is the variable and `&` (ampersand) must precede it.

The format specifiers and their meanings are given below.

`%d`: The data is taken as integer.

`%c`: The data is taken as character.

`%s`: The data string.

`%f`: The data is taken as float.

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

The `scanf()` statement requires ‘`&`’ operator called address operator. The address operator prints the memory location of the variable. Here, in the `scanf()` statement the role of ‘`&`’ operator is to indicate the memory location of the variable, so that the value read would be placed at that location.

The `scanf()` statement also returns values. The return value is exactly equal to the number of values correctly read. In case of any mismatch, error will be thrown. Otherwise, if the read value is convertible to the given format, conversion is made. The following program shows an example of such a mismatch case.

- 4.6 Write a program to show the effect of mismatch of data types.

```
void main()
{
    int a;
    clrscr();
    printf("Enter value of 'A' : ");
    scanf("%c",&a);
    printf("A=%c",a);
}
```

OUTPUT:

```
Enter value of 'A' : 8
A=8
```

Explanation:

In the above program, although the type of variable 'a' is `int`, it works perfectly with conversion symbol of character, i.e. character and integer data types are compatible to each other. When the two data types are compatible to each other, the compatible range is equal to the lowest range from the two data types. The above example illustrates this point.

- 4.7 Write a program to read and print the integer value using the character variable.

```
void main()
{
    char a;
    clrscr();
    printf("Enter value of 'A' : ");
    scanf("%d",&a);
    printf("A=%d",a);
}
```

OUTPUT:

```
Enter value of 'A' : 255
A=255
Enter value of 'A' : 256
A=0
```

Explanation:

In the above program variable 'a' is of character type, i.e. its valid range is 0 to 255. The variable 'a' is used with conversion symbol of integer data type, i.e. in the `printf()` and `scanf()` statements the variable 'a' is supposed as an integer type. The value read in the first execution is valid. Hence, it is printed as it is read. In the second execution, the value read is greater than the range of the character type. In such a case, the excess range is again considered as the beginning or starting point. Here, the excess value is 1. That is why 0 is printed.

Consider the following `scanf()` statements, where `a`, `b` and `c` are integer variables.

Examples:

- (i) `scanf( "%d %d %d", a, b, c );`
- (ii) `scanf( "%d,%d,%d", a, b, c );`

In the first statement, the format strings (%d) are separated by a space. It indicates that while inputting values for these variables, the values should be separated by space. Similarly, in the second statement, the format strings (%d) are separated by a comma; therefore, while inputting, values should be separated by a comma. The following program illustrates this.

- 4.8 Write a program to demonstrate the use of comma with `scanf( )` statement.

```
void main()
{
    int a,b,c;
    clrscr();
    printf("\nEnter values :");
    scanf( "%d, %d,%d",&a,&b,&c);
    printf("a=%d b=%d c=%d",a,b,c);
}
```

OUTPUT:

```
Enter values: 4,5,8
a=4 b=5 c=8
```

Explanation:

From the above program, it is very clear that if format strings are separated by commas, the inputs should also be separated by commas. The readers are advised to try this by writing more programs. Table 4.2 describes the formats for the various inputs.

Table 4.2 *Formats for the various inputs*

<i>Sr. No.</i>	<i>Format</i>	<i>Meaning</i>	<i>Explanation</i>
1.	%wd	Format for integer input	w is width in integer and d conversion specification
2.	%w.cf	Format for float point input	w is width in integer, c specifies the number of digits after decimal point and f is conversion specification
3.	%w.CS	Format for string input	w is the width for total characters, c are used for inserting blanks and s is conversion specification

An example illustrating a few formatted input is as follows.

- 4.9 Write a program to demonstrate the use of `scanf( )` with different formats.

```
void main()
{
    int a,b;
    float x;
```

```

char name[20];
clrscr();
printf("Enter two integers:-\n");
scanf("%4d %4d",&a,&b);
printf("\nEntered integers are");
printf("\n%4d %4d",a,b);
printf("\n");
printf("\nEnter a real number:-\n");
scanf("%f",&x);
printf("\n Entered float number is ");
printf("\n%f",x);
printf("\n");
printf("\nEnter a string :-\n");
scanf("%7s",name);
printf("\n Entered string ");
printf("\n%7s",name);
getche();
}

```

OUTPUT:

```

Enter two integers:-
1 2
Entered integers are
1 2
Enter a real number:-
12.3
Entered float number is
12.300000
Enter a string:-
Deelipkumar
Entered string
Deelipk

```

Explanation:

In the `scanf()` statements, the format for various inputs are taken. Similarly, the outputs are also provided with various formats. The readers can verify the input and output.

The `printf()` and `scanf()` statements follow different data types which are listed in Table 4.3. It can be seen from this table that format string is initialized with `%sign` as a special character, which indicates the format of the data to be displayed on the screen.

Table 4.3 Data types with conversion symbols

Data Type		Format String
Integer	Short integer	%d or %i
	Short unsigned	%u
	Long signed	%ld
	Long unsigned	%lu
	Unsigned hexadecimal	%x
	Unsigned octal	%o

(Continued)

Table 4.3 *Data types with conversion symbols (Continued)*

<i>Data Type</i>		<i>Format String</i>
Real	Floating	%f or %g
	Double floating	%lf
Character	Signed character	%c
	Unsigned character	%C
	String	%s
Octal number		%o
Displays Hexa decimal number in lowercase		%hx
Displays Hexa decimal number in uppercase		%p
Aborts program with error		%n

The `printf()` and `scanf()` statements follow the combination of characters called escape sequences. In order to come out, computers from routine sequence escape sequences are used. These are nothing but special characters starting with '\'. The escape sequences and their uses are illustrated in Table 4.4.

Table 4.4 *Escape sequences with their ASCII values*

<i>Escape Sequence</i>	<i>Use</i>	<i>ASCII Value</i>
\n	New line	10
\b	Backspace	8
\f	Form feed	12
\'	Single quote	39
\\	Backslash	92
\0	Null	0
\t	Horizontal tab	9
\r	Carriage return	13
\a	Alert	7
\"	Double quote	34
\v	Vertical tab	11
\?	Question mark	63

➤ 4.10 Write a program to show the effect of various escape sequences.

```
void main()
{
    int a=1,b=a+1,c=b+1,d=c+1;
    clrscr();
    printf("\tA=%d\nB=%d \tC=%d'",a,b,c);
    printf("\n\b***\D=%d***",d);
    printf("\n*****");
}
```

```
printf("\rA=%d B=%d", a, b);
}
```

OUTPUT:

```
      A=1
B=2 'C=3'
***D=4**
A=1 B=2*****
```

Explanation:

In the above program, a few commonly used escape sequences are described.

- (i) In the first `printf()` statement due to `'\t'`, prints the value of `'a'` after a tab. The `'\n'` splits the line and prints the value of B and C on the next line.
- (ii) In the second `printf()` statement the three prefixes `'*'` are written followed by the `'\b'`. The `'\b'` overwrites the last character. The output `D=4` will be displayed.
- (iii) In the third `printf()` statement only the sequences of `'*'` are printed, but in the output only half line is displayed because it is affected by the fourth `printf()` statement.
- (iv) In the fourth `printf()` statement `'\r'` is used which reverses the printable area one line before from the current location. Hence, the line generated by the third statement is replaced by the output of the fourth statement.

The programs illustrated below use `printf()` and `scanf()` statements.

- 4.11 Write a program to print the third power of 2 using `pow()` function. Assume the floating-point numbers.

```
#include <math.h>
void main()
{
    double x = 2.0, y = 3.0;
    clrscr();
    printf("%lf raised to %lf is %lf\n", x, y, pow(x, y));
}
```

OUTPUT:

```
2.000000 raised to 3.000000 is 8.000000
```

Explanation:

In the above program, two variables `x` and `y` are declared and initialized. In the `printf()` statement using `pow()` function expression `x^y` is calculated and displayed.

- 4.12 Write a program to print the third power of 10 using `pow10()` function. Assume the floating-point numbers.

```
#include <math.h>
void main
{
    int p = 3;
```

```
printf("Ten raised to %lf is %lf\n", p, pow10(p));
}
```

OUTPUT:

```
Ten raised to 3.000000 is 1000.000000
```

Explanation:

In the above program, power of 10 is calculated. Here, `p` is declared as an integer data type. The value returned by this function is of double data type. Hence, conversion symbol `%lf` is used.

- 4.13 Write a program to detect an error while inputting a data. Use return value of `scanf()` statement.

```
void main()
{
    int a,b,c,v;
    clrscr();
    printf("Enter value of 'A','B' & 'C' : ");
    v=scanf("%d %d %d",&a,&b,&c);
    (v<3 ? printf("\n Error In Inputting.") : printf("\n Values
    Successfully read.));
}
```

OUTPUT:

```
Enter value of 'A','B' & 'C' : 1 2 3
Value Successfully read.
Enter value of 'A','B' & 'C' : 1 J 2
Error In Inputting.
```

Explanation:

In the above program, the `printf()` statement returns values equal to the number of variables correctly read. The conditional statement checks the value of variable '`v`' and prints the respective messages.

- 4.14 Write a program to find the length of the string using `printf()` function.

```
void main()
{
    char nm[20];
    int l;
    clrscr();
    printf("Enter String :");
    scanf("%s",nm);
    l=printf(nm);
    printf("\nLength = %d",l);
}
```

OUTPUT:

```
Enter String : HELLO
Length = 5
```

Explanation:

The `printf()` function returns the length of the string entered. In the above program the string entered is 'HELLO'. Length of the string is 5, which is stored in variable 'l'.

- 4.15 Write a program to perform the addition of two numbers.

```
void main()
{
    int a,b,c;
    printf("\n ENTER TWO VALUES\n");
    scanf("\n %d %d", &a, &b);
    c=a+b;
    printf("\n Sum is=%d",c);
}
```

OUTPUT:

```
ENTER TWO VALUES 5 8
Sum is = 13
```

Explanation:

In the above program, variables `a`, `b` and `c` are declared. Values of `a` and `b` are read through the keyboard using `scanf()` statement. The addition of variables `a` and `b` is performed and assigned to variable `c`.

- 4.16 Write a program to find the square of the given number.

```
void main()
{
    int a,c;
    printf("\n ENTER ANY NUMBER\n");
    scanf("\n %d", &a);
    c=a*a;
    printf("\n SQUARE OF GIVEN NUMBER = %d",c);
}
```

OUTPUT:

```
ENTER ANY NUMBER 5
SQUARE OF GIVEN NUMBER 25
```

Explanation:

The above program is same as the previous one. Only difference is that instead of addition, the square of `a` is calculated.

- 4.17 Write a program to input a single character and display it.

```
void main()
{
    char ch;
```

```

clrscr();
printf("Enter any character :");
scanf("%c",&ch);
printf("\n Your Entered Character is : %c",ch);
}

```

OUTPUT:

```

Enter any character: C
Your Entered Character is: C

```

Explanation:

In the above program, a character is entered and stored in variable `ch`. The `printf()` statement displays the entered character.

- 4.18 Write a program to swap the values of two variables without the use third variable.

```

void main()
{
    int a=7,b=4;
    clrscr();
    printf("\n A= %d B= %d",a,b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf("Now A= %d B= %d",a,b);
}

```

OUTPUT:

```

A=7 B=4
Now A= 4 B=7

```

Explanation:

In the above program, no third variable is used as a mediator for swapping the values. The below given steps illustrate the working of the program.

- (i) In the first statement, variable 'a' contains the sum of $a+b$, i.e. 11.
- (ii) In the second statement, variable 'b' contains $a-b$, i.e. $11-4=7$.
- (iii) In the third statement, variable 'a' contains $a-b$, i.e. $11-7=4$.

Thus, the two values are interchanged.

4.4 UNFORMATTED FUNCTIONS

C has three types of I/O functions.

- (i) Character I/O
- (ii) String I/O
- (iii) File I/O

(i) Character I/O

1.
getchar() -

This function reads a character-type data from standard input. It reads one character at a time till the user presses the enter key. The syntax of the `getchar()` is as follows:

Variable name=`getchar()`;

Example:

```
char c;
c=getchar();
```

A program is supported for the following `getchar()` function.

- 4.19 Write a program to accept characters through keyboard using `getchar()` function.

```
void main()
{
    char c;
    clrscr();
    printf("\nEnter a char :");
    c=getchar();
    printf("a=%c", c);
}
```

OUTPUT:

```
Enter a char :g
a=g
```

Explanation:

In the above program, a character variable `c` is declared. The `getchar()` reads a character through the keyboard. The same is displayed by the `printf()` statement.

2.
putchar() -

This function prints one character on the screen at a time, read by the standard input.

The syntax is as follows:

`putchar(variable name);`

Example:

```
char c='C';
putchar (c);
```

A program is provided on `putchar()`.

- 4.20 Write a program to use `putchar()` in work.

```
void main()
{
    char c='C';
    clrscr();
    putchar(c);
}
```

OUTPUT:

C

Explanation:

In this program, the character variable `c` assigns a char 'C'; the same is displayed by the `putchar()` statement. The argument `C` is used with the `putchar()` statement.

3.
`getch()` & `getche()`

These functions read any alphanumeric character from the standard input device. The character entered is not displayed by the `getch()` function.

Syntax of `getche()` is as follows:

```
getche();
```

- 4.21 Write a program to show the effect of `getche()` and `getch()`.

```
void main()
{
    clrscr();
    printf("Enter any two alphabetic");
    getche();
    getch();
}
```

OUTPUT:

Enter any two alphabetic A

Explanation:

In the above program, even though two characters are entered, the user can see only one character on the screen. The second character is accepted but not displayed on the console. The `getche()` accepts and displays the character whereas `getch()` accepts but does not display the character.

4.
`putch()` :

This function prints any alphanumeric character taken by the standard input device.

- 4.22 Write a program to read and display the character using `getch()` and `putch()`.

```
void main()
{
    char ch;
    clrscr();
    printf("Press any key to continue");
    ch=getch();
    printf("\n You Pressed :");
    putch(ch);
}
```

OUTPUT:

```
Press any key to continue
You Pressed: 9
```

Explanation:

The function `getch()` reads a keystroke and assigns to the variable `ch`. The `putch()` displays the character pressed.

- (ii) String I/O

1.
`gets()` :

This function is used for accepting any string through `stdin` (keyboard) until enter key is pressed. The header file `stdio.h` is needed for implementing the above function. Format of `gets()` is as follows:

```
char str[length of string in number];
gets(str)
```

A program is given on `gets()`.

- 4.23 Write a program to accept string through the keyboard using the `gets()` function.

```
void main()
{
    char ch[30];
    clrscr();
    printf("Enter the String :");
    gets(ch);
    printf("\n Entered String : %s", ch);
}
```

OUTPUT:

```
Enter the String : USE OF GETS()
Entered String : USE OF GETS()
```

Explanation:

In the above program, `gets()` reads string through the keyboard and stores it in character array `ch[30]`. The `printf()` function displays the string on the console.

2.
puts() :

This function prints the string or character array. It is opposite to gets().

```
char str[length of string in number];
gets(str);
puts(str);
```

A program is given on puts().

➤ 4.24 Write a program to print the accepted character using puts() function.

```
void main()
{
    char ch[30];
    clrscr();
    printf("Enter the String :");
    gets(ch);
    puts("Entered String :");
    puts(ch);
}
```

OUTPUT:

```
Enter the String: puts is in use.
Entered String:
puts is in use.
```

Explanation:

This program is the same as the previous one. Here, to display the string puts() function is used.

3.
cgets() :

This function reads string from the console. The syntax is as follows.

Syntax:

```
cgets(char *st);
```

It requires character pointer as an argument. The string begins from st[2].

4.
cputs() :

This function displays string on the console. The syntax is as follows.

Syntax:

```
cputs(char *st);
```

- 4.25 Write a program to read string using `cgets()` and display it using `cputs()`.

```
void main()
{
    static char *t;
    clrscr();
    printf("\n Enter Text Here :");
    cgets(t);
    t+=2;
    printf("\n Your Entered Text :");
    cputs(t);
    getch();
}
```

OUTPUT:

```
Enter Text Here: How are you?
Your Entered Text: How are you?
```

Explanation:

In this example character pointer 't' is declared. The `cgets()` function reads string through the keyboard and the `cputs()` function displays the string on the console.

4.5 COMMONLY USED LIBRARY FUNCTIONS

1.
`clrscr()`

This function is used to clear the screen. It clears previous output from the screen and displays the output of the current program from the first line of the screen. It is defined in `conio.h` header file. The syntax is as follows.

Syntax:

```
clrscr();
```

2.
`exit()`

This function terminates the program. It is defined in `process.h` header file. The syntax is as follows.

Syntax:

```
exit();
```

3.
`sleep()`

This function pauses the execution of the program for a given number of seconds. The number of seconds is to be enclosed between parentheses. It is defined in `dos.h` header file. The syntax is as follows.

Syntax:

```
sleep(1);
```

An example on `sleep()` is given below.

- 4.26 Write a program to show the effect of the `sleep()` function.

```
void main()
{
    static char t[10];
    clrscr();
    printf("\n Enter Text Here :");
    gets(t);
    printf("\n Your Entered Text :");
    sleep(5);
    puts(t);
    getch();
}
```

OUTPUT:

```
Enter Text Here: ashok
Your Entered Text: ashok
```

Explanation:

The explanation is straightforward and self-explanatory. See the effect of `sleep(5)`. The display appears after taking a pause.

```
3.
system ()
```

This function is helpful in executing different DOS commands. It returns 0 on success and -1 on failure. The syntax is as follows.

Syntax:

```
system ("dir");
```

The command should be enclosed within double quotation marks. After we run this command using C, directory will be displayed. Programmer can verify this command.

4.6 STRONG POINTS FOR UNDERSTANDABILITY

Computer produces output, which is useful for the user. Therefore, the clarity and neatness of result should appear in the output. The following are a few steps which can be followed to produce the neatness output:

- (i) Give space between numbers.
- (ii) Provide suitable and problem-related variable names and headings.
- (iii) Provide user prompt so that the user can understand what to do.

- (iv) Provide a gap between two lines so that the text should be readable.
- (v) Alert the user about what to do and what not to do.
- (vi) Use formatted inputs and outputs for precisely inputting the data and outputting results.
- (vii) It is recommended to use escape sequence characters such as `\t`, `\b`, `\n`.

SUMMARY

This chapter dealt with formatted functions such as `printf()` and `scanf()` statements. The unformatted functions such as `putchar()`, `getche()`, `gets()` have been illustrated with suitable examples. The different data types and conversion symbols used in the C programs have also been elaborated. The special symbols such as escape sequences together with their applications are also discussed. A few of the functions which are commonly used in the programs such as `clrscr()`, `exit()` are described in this chapter. Input and output functions together with examples are narrated with programming example. At last, the main points for the understanding of programs are given to the readers so that they can follow them.

EXERCISES

I Fill in the blanks:

1. _____ functions provide the conversion symbol to identify the data type.
 - (a) Formatted
 - (b) Unformatted
 - (c) Library
 - (d) User defined
2. _____ functions does not convert data.
 - (a) Formatted
 - (b) Unformatted
 - (c) Library
 - (d) User defined
3. The function prints all types of data values on to the console _____.
 - (a) `printf()`
 - (b) `scanf()`
 - (c) `gets()`
 - (d) `pow()`
4. The _____ statement reads all types of data values.
 - (a) `scanf()`
 - (b) `printf()`
 - (c) `puts()`
 - (d) `abs()`
5. _____ function reads one character type data at a time till the user presses the enter key.
 - (a) `getchar()`
 - (b) `puts()`
 - (c) `accept()`
 - (d) `floor()`

II True or false:

1. Formatted functions require format string to format the data.
2. The function `gets()` is an unformatted function.
3. The `'\n'` escape sequence inserts a tab.
4. Formatted functions return values.
5. The function `gets()` is defined in `<string.h>`.
6. Any signed data type can have negative as well as positive values.

7. The functions `cgets()` and `cputs()` work with character pointer as argument.
8. While inputting values through `scanf()`, `&` is required before a variable name.
9. The `getche()` is used to read data character by character.
10. The `char` requires one byte space in the memory.
11. The `'\a'` escape sequence is for alert bell.
12. The format string `%g` is used for float type.
13. The `%s` is used to format the string.
14. The format string `%P` is used to display hexadecimal in lowercase.
15. The `%lf` is used for long integer

III Select the appropriate options from the choices given in the questions:

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

```
void main()
{
    printf("\n %d%d%d%d", 'A',
        'B', 'C', 'D');
}
```

 - (a) 65666768
 - (b) ABCD
 - (c) 91929394
 - (d) None of the above
2. What will be the values of `a` and `b` after the execution of the following program?

```
void main()
{
    int a,b;
    a=65*66;
    b='A' * 'B';
    clrscr();
    printf("a=%d b=%d", a, b);
}
```

 - (a) `a=4290 b=4290`
 - (b) `a=4290 b=AB`
 - (c) `a=4290 b=0`
 - (d) None of the above
3. What function is appropriate for accepting a string?
 - (a) `gets()`
 - (b) `getch()`
 - (c) `getche()`
 - (d) `scanf()`
4. What is the ASCII range for 0 to 9 digits?
 - (a) 48 to 57
 - (b) 65 to 90
 - (c) 97 to 122
 - (d) None of the above
5. What is the ASCII range for A to Z letters?
 - (a) 65 to 90
 - (b) 48 to 57
 - (c) 97 to 122
 - (d) None of the above
6. The escape sequence `'\t'` is a
 - (a) tab
 - (b) next line
 - (c) backspace
 - (d) None of the above
7. What would be the value of `x` on execution of the program?

```
void main()
{
    float x=2.3;
    clrscr();
    x+=.2;
    printf("%g", x);
}
```

 - (a) 2.5
 - (b) 4.3
 - (c) 4
 - (d) None of the above
8. What will be the output of the following program?

```
void main()
{
    system("");
}
```

 - (a) control goes to the DOS prompt
 - (b) syntax error
 - (c) bad command or file name
 - (d) None of the above

9. Which is the correct statement for finding the cube of 2?
- `pow(2,3);`
 - `pow(3,2);`
 - `pow(3);`
 - None of the above
10. The `abs()` function displays
- an absolute value
 - a negative value
 - a zero value
 - None of the above
11. What will be the output of the following program?
- ```
void main()
{
 printf("\n %d%d%d%d", 'a',
 'b', 'c', 'd');
}
```
- 979899100
  - 87888990
  - 90919293
  - None of them
12. What will be the output of the following program?
- ```
void main()
{
    char yourname[10]={"AJAY"};
    clrscr();
    printf("\n Welcome %s to 'C'
        Programming Course", yourname);
}
```
- Welcome AJAY to 'C' Programming Course
 - Welcome to 'C' Programming Course
 - Welcome 'C' Programming Course
 - None of them
13. What will be the values of a and b after execution of the following program?
- ```
void main()
{
 int a,b;
 a=65*66;
 b='A'*'B';
 clrscr();
 printf("a=%d b=%d", a,b);
}
```
- a=4290 b=4290
  - a=4290 b=9506
  - a=4290 b=0
  - None of the above

#### IV Attempt the following programming exercises:

- Write a program to input the rainfall of three consecutive days in CMS and find its average?
- Find the simple interest? Inputs are principal amount, period in year and rate of interest.
- Write a program to find the total number of minutes of 12 hours?
- Find the area and perimeter of (a) square and (b) rectangle. Input the side(s) through the keyboard?
- Accept any three numbers and find their squares and cubes.
- The speed of a van is 80 km/hour. Find the number of hours required for covering a distance of 500 km? Write a program in this regard.
- Write a program to convert inches into centimetres.
- Write a program to enter the name of this book and display it.
- Write a program to store and interchange two float numbers in variables a and b.
- Write a program to enter text with `gets()` and display it using `printf()` statement. Also find the length of the text.
- Write a program to ensure that the subtraction of any two-digit number and its reverse is always the multiple of nine. For example, entered number is 54 and its reverse is 45. The difference between them is 9.
- Write a program to convert kilograms into grams.
- Write a program to find the total amount when there are five notes of Rs. 100, three notes of Rs. 50 and 20 notes of Rs. 20.



14. Write a program to enter the temperature in Fahrenheit and convert it to Celsius. Formula to be used is  $tc = ((tf - 32) * 5) / 9$  where  $tc$  and  $tf$  are temperatures in Celsius and Fahrenheit, respectively.
15. Write a program to display the list of c program files and directories. Use `system()` function to execute DOS commands.

## V What will be the output/s of the following program/s?

1.
 

```
void main()
{
 clrscr();
 printf("\n %0 %hx %p", 45,
 65, 65);
}
```
2.
 

```
void main()
{
 int x=67,y=68,z=69;
 clrscr();
 printf("\n%c %c %c", x,
 y, z);
 getch();
}
```
3.
 

```
void main()
{
 char str="C Programming";
 clrscr();
 puts(str);
 getch();
}
```
4.
 

```
void main()
{
 int k='A';
 clrscr();
 while (k<='K') putchar(k++);
}
```

## VI Find the bug/s in the following program/s:

1.
 

```
void main()
{
 int x=2;
 char y='A';
 float f=2.05;
 clrscr();
 printf("%d %c %f", f, y, x);
 getch();
}
```
2.
 

```
void main()
{
 int x;
 clrscr();
 printf("Enter a
 Number :");
 scanf("%d", x);
 printf("%d", x);
}
```
3.
 

```
void main()
{
 char x, d;
 clrscr();
 printf("Enter an
 alphabet:");
 x=getchar();
}
```
4.
 

```
void main()
{
 clrscr();
 printf("I \nam /n an\t
 Indian");
}
```
5.
 

```
void main()
{
 putchar("x");
}
```

**VII Answer the following questions:**

1. What are the formatted and unformatted functions?
2. What is the difference between character I/O and string I/O?
3. What is the escape sequence? List and indicate the functions of escape sequences.
4. List any three escape sequences with their uses.
5. What is the difference between puts() and putchar()?
6. What is the difference between getch() and getche()?
7. How cgets() is different from gets()?
8. How will you execute a DOS command through C program?
9. What is the use of the exit() function?
10. What is a stream?
11. What are the tips to the design output?

**ANSWERS****I Fill in the blanks:**

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | a    | 2. | b    | 3. | a    | 4. | a    | 5. | a    |

**II True or false:**

| Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. |
|-----|------|-----|------|-----|------|-----|------|-----|------|
| 1.  | T    | 2.  | T    | 3.  | F    | 4.  | T    | 5.  | F    |
| 6.  | T    | 7.  | T    | 8.  | T    | 9.  | T    | 10. | T    |
| 11. | T    | 12. | T    | 13. | T    | 14. | F    | 15. | F    |

**III Select the appropriate options from the choices given in the questions:**

| Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q. | Ans. | Q.  | Ans. |
|-----|------|-----|------|-----|------|----|------|-----|------|
| 1.  | a    | 2.  | a    | 3.  | a    | 4. | a    | 5.  | a    |
| 6.  | a    | 7.  | a    | 8.  | a    | 9. | a    | 10. | a    |
| 11. | a    | 12. | a    | 13. | b    |    |      |     |      |

**V What will be the output/s of the following program/s?**

| Q. | Ans.        |
|----|-------------|
| 1. | 55 41 0041  |
| 2. | C D E       |
| 3. | No Output   |
| 4. | ABCDEFGHIJK |

**VI Find the bug/s in the following program/s:**

| <b>Q.</b> | <b>Ans.</b>                                                                                                                                                             |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>1.</b> | Program runs but output will be 0 because formatted data cannot be displayed through unformatted functions. After putting them in sequence output will be 2 A 2.050000. |
| <b>2.</b> | In <code>scanf( )</code> & is not prefixed before variable name.                                                                                                        |
| <b>3.</b> | Header file <code>&lt;stdio.h&gt;</code> must be included                                                                                                               |
| <b>4.</b> | No bug                                                                                                                                                                  |
| <b>5.</b> | <code>putchar( )</code> cannot be used to display string. Instead of it use <code>puts( )</code> .                                                                      |

## CHAPTER

# 5

# Decision Statements

### Chapter Outline

- 5.1 Introduction
- 5.2 The `if` Statement
- 5.3 The `if-else` Statement
- 5.4 Nested `if-else` Statements
- 5.5 The `if-else-if` Ladder Statement
- 5.6 The `break` Statement
- 5.7 The `continue` Statement
- 5.8 The `goto` Statement
- 5.9 The `switch` Statement
- 5.10 Nested `switch` case
- 5.11 The `switch` case and **nested** `ifs`

## 5.1 INTRODUCTION

A program is nothing but the execution of one or more instructions sequentially in the order in which they come into sight. This process is analogous to reading the text and figures that appear on the page of a notebook. In the monolithic program, the instructions are executed sequentially in the order in which they appear in the program. Quite often, it is desirable in a program to alter the sequence of the statements depending upon certain circumstances. In real time applications, there are a number of situations where one has to change the order of the execution of statements based on the conditions.

Decision-making statements in a programming language help the programmer to transfer the control from one part to other parts of the program. Thus, these decision-making statements facilitate the programmer in determining the flow of control. This involves a decision-making condition to see whether a particular condition is satisfied or not. On the basis of real time applications it is essential:

- (i) to alter the flow of a program.
- (ii) to test the logical conditions.
- (iii) to control the flow of execution as per the selection.

These conditions can be placed in the program using decision-making statements. C language supports the control statements as listed below:

- (i) The `if` statement
- (ii) The `if-else` statement
- (iii) The `if-else-if` ladder statement
- (iv) The `switch case` statement
- (v) The `goto` unconditional jump
- (vi) The `loop` statement

Besides, the C also supports other control statements such as `continue`, `break`.

The decision-making statement checks the given condition and then executes its sub-block. The decision statement decides which statement to execute after the success or failure of a given condition.

The conditional statements use relational operators, which have been explained in Chapter 3. The relational operators are useful for comparing the two values. They can be used to check whether they are equal to each other, unequal or one is smaller/greater than the other.

The reader or the programmer is supposed to understand the concepts as cited above. Following points are expected to be known to the programmer related to the decision-making statements.

**Sequential execution:** The statements in the program are executed one after another in a sequential manner. This is called the sequential execution.

**Transfer of control:** The C statements such as `if`, `goto`, `break`, `continue`, `switch` allow a program to transfer the control at different places in the program. This is accomplished by skipping one or more statements appearing in a sequential flow. This jumping of control is called the transfer of control.

During 1960s, programmers faced various difficulties in the program in which `goto` statement was used which allows a programmer to transfer the control anywhere in the program. Bohm and Jacopini reported that the programmer should not use the `goto` statement in their programs. Both

Bohm and Jacopini demonstrated that all the programs could be written by using only three control structures, i.e. sequence structure, selection structure and repetition structure by eliminating the `goto` statement.

## 5.2 THE `if` STATEMENT

C uses the keyword `if` to execute a set of command lines or one command line when the logical condition is true. It has only one option. The sets of command lines are executed only when the logical condition is true (see Figure 5.1).

Syntax for the simplest `if` statement :-  
`if (condition) /* no semi-colon */  
 statement;`

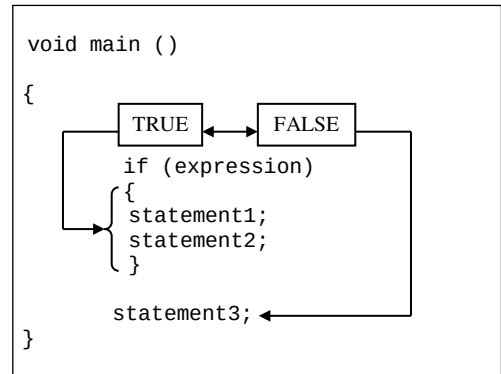


**Figure 5.1** The `if` statement

The `if` statement contains an expression which is evaluated. If the expression is true it returns 1, otherwise 0. The statement is executed when the condition is true. In case the condition is false, the compiler skips the lines within the `if` block. The condition is always enclosed within a pair of parentheses. The conditional statements should not be terminated with semi-colons (;). The statements following the `if` statement are normally enclosed within curly braces. The curly braces indicate the scope of the `if` statement. The default scope is one statement. But it is good practice to use curly braces even if a single statement is used following the `if` condition.

Given below are simple programs that demonstrate the use of the `if` statement.

- 5.1 Write a program to check whether the entered number is less than 10. If yes, display the same.

```

void main()
{
 int v;
 clrscr();
 printf("Enter the number :");
 scanf("%d", &v);
 if(v<10)
 printf("\nNumber entered is less than 10");
 sleep(2); /* process halts for given value in seconds */
}

```

### OUTPUT:

```

Enter the number : 9
Number entered is less than 10

```

**Explanation:**

In the above program, the user can enter the number. The entered number is checked with the `if` statement. If it is less than 10, a message 'Number entered is less than 10' is displayed. For the sake of understanding, Figure 5.2 is given for the above program.

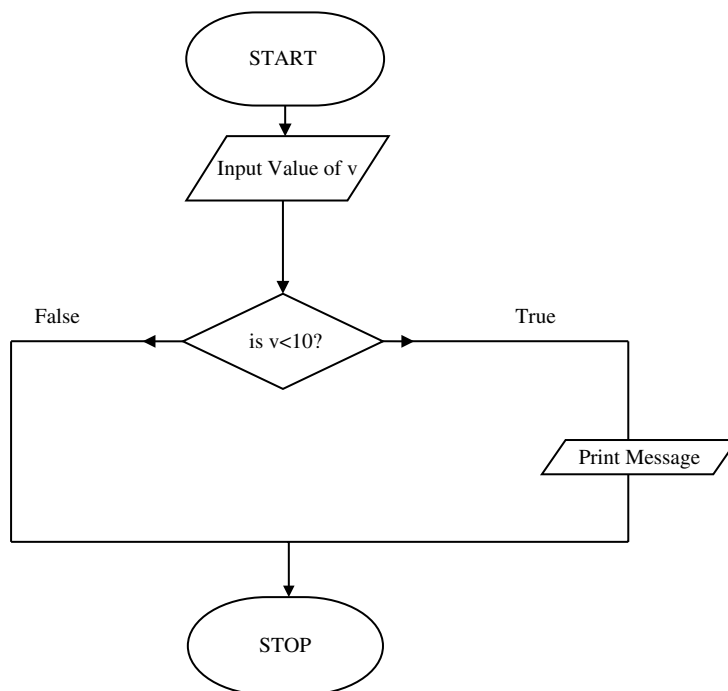


**Figure 5.2** The `if` statement (flow of control)

- 5.2 Write a program to check whether the candidate's age is greater than 17 or not. If yes, display message 'Eligible for Voting'.

```

void main()
{
 int age;
 clrscr();
 printf("\n Enter age in the years :");
 scanf("%d",&age);
 if(age>17)
 printf("\n Eligible for Voting.");
 getch();
}

```

**OUTPUT:**

```

Enter age in the years : 20
Eligible for Voting.

```

**Explanation:**

In the above program, age is entered through the keyboard. If the age is greater than 17 years, a message will be displayed as shown at the output.

- 5.3 Write a program using curly braces in the `if` block. Enter only the three numbers and calculate their sum and multiplication.

```
void main()
{
 int a,b,c,x;
 clrscr();
 printf("\nEnter Three Numbers:");
 x=scanf("%d %d %d",&a,&b,&c);
 if(x==3)
 {
 printf("\n Addition : %d",a+b+c);
 printf("\n Multiplication : %d", a*b*c);
 }
}
```

**OUTPUT:**

```
Enter Three Numbers: 1 2 4
Addition : 7
Multiplication : 8
After second time execution
Enter Three Numbers: 5 v 8
```

**Explanation:**

The variable 'x' contains the number of values correctly inputted by the user. If the value of 'x' is 3, the addition and multiplication operations are performed as per the first example in the output. In the second example, the numbers are not correctly entered. Hence, the `if` condition is false and no operation is performed. Here, the statements following the `if` condition is enclosed within curly braces.

## 5.3 THE `if-else` STATEMENT

We observed the execution of the `if` statement in the previous programs. We observed that the `if` block statements execute only when the condition in `if` is true. When the condition is false, program control executes the next statement which appears after the `if` statement.

The `if-else` statement takes care of the true and false conditions. It has two blocks. One block is for `if` and it is executed when the condition is true. The other block is of `else` and it is executed when the condition is false. The `else` statement cannot be used without `if`. No multiple `else` statements are allowed with one `if` (see Figures 5.3 and 5.4).

The flow chart for the `if-else` statement is given in Figure 5.4.



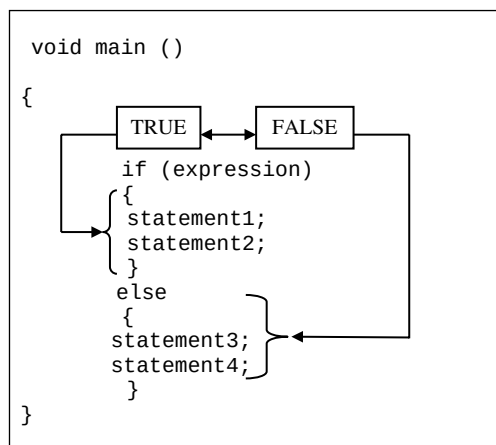


Figure 5.3 The if-else statements

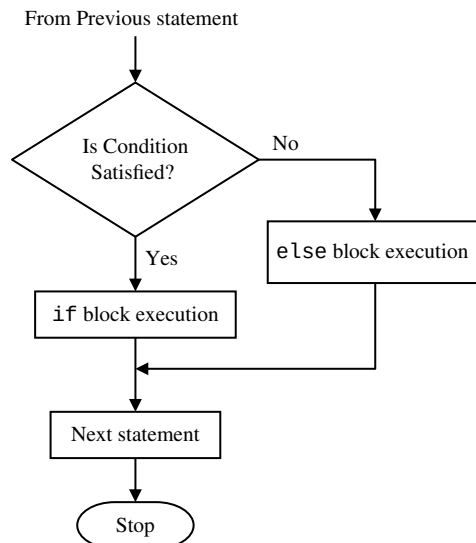


Figure 5.4 The if-else statement

The syntax of if-else statement is as follows:

```

if(the condition is true)
execute the Statement1;
else
execute the Statement2;

```

OR

Syntax of if-else statement can be given as follows:

```

if (expression is true)
{
 statement1; /* if block */
 statement2;
}
else
{
 statement3; /* else block */
 statement4;
}

```

The if-else statement is demonstrated in the following programs.

- 5.4 Write a program to find the roots of a quadratic equation by using if-else condition.

```

#include <math.h>
void main()
{
 int b,a,c;

```

```

float x1,x2;
clrscr();
printf("\n Enter Values for a,b,c :");
scanf("%d %d %d", &a,&b,&c);
if(b*b>4*a*c)
{
 x1=-b+sqrt(b*b-4*a*c)/2*a;
 x2=-b-sqrt(b*b-4*a*c)/2*a;
 printf("\n x1=%f x2=%f",x1,x2);
}
else
 printf("\n Roots are Imaginary");
getch();
}

```

**OUTPUT:**

```

Enter Values for a,b,c : 5 1 5
Roots are Imaginary

```

**Explanation:**

The user can enter the values of a, b and c in the above program. The terms  $b^2$  and  $4*a*c$  are evaluated. The `if` condition checks whether  $b^2$  is greater than  $4*a*c$ . If true, `x1` and `x2` are evaluated and printed; otherwise message displayed will be 'Roots are Imaginary'.

- 5.5 Write a program to calculate the square of those numbers only whose least significant digit is 5.

```

void main()
{
 int s,d;
 clrscr();
 printf("\n Enter a Number :");
 scanf("%d",&s);
 d=s%10;
 if(d==5)
 {
 s=s/10;
 printf("\n Square = %d%d",s*s++,d*d);
 }
 else
 printf("\n Invalid Number");
}

```

**OUTPUT:**

```

Enter a Number : 25
Square = 625

```

**Explanation:**

In the above program, a number whose square is to be computed is entered. With the modular division, operation the last digit is separated to confirm whether it is 5 or not. If yes, the body of the `if` loop is executed where 10 divides the entered number and a quotient is obtained.

The quotient and its consecutive number are multiplied and displayed. Followed by this, a square of 5 is calculated and displayed. Care is taken in the `printf()` statement to display the two results: (i) multiplication of quotient and its consecutive number and (ii) square of 5 without space. Thus, the square of a number is displayed.

- 5.6 Write a program to calculate the salary of a medical representative based on the sales. Bonus and incentive to be offered to him will be based on total sales. If the sale exceeds or equals to Rs.1,00,000, follow the particulars of Table 1, otherwise follow Table 2.

#### 1. TABLE

Basic=Rs. 3000.  
Hra=20% of basic.  
Da=110% of basic.  
Conveyance=Rs.500.  
Incentive=10% of sales.  
Bonus=Rs. 500.

#### 2. TABLE

Basic=Rs. 3000.  
Hra=20% of basic.  
Da=110% of basic.  
Conveyance=Rs.500.  
Incentive=5% of sales.  
Bonus=Rs. 200.

```
void main()
{
 float bs,hra,da,cv,incentive,bonus,sale,ts;
 clrscr();
 printf("\n Enter Total Sales in Rs.:");
 scanf("%f", &sale);
 if(sale>=100000)
 {
 bs=3000;
 hra=20 * bs/100;
 da=110 * bs/100;
 cv=500;
 incentive=sale*10/100;
 bonus=500;
 }
 else
 {
 bs=3000;
 hra=20 * bs/100;
 da=110 * bs/100;
 cv=500;
 incentive=sale*5/100;
 bonus=200;
 }
 ts=bs+hra+da+cv+incentive+bonus;
 printf("\nTotal Sales : %.2f",sale);
 printf("\nBasic Salary : %.2f",bs);
 printf("\nHra : %.2f",hra);
 printf("\nDa : %.2f",da);
 printf("\nConveyance : %.2f",cv);
 printf("\nIncentive : %.2f",incentive);
 printf("\nBonus : %.2f",bonus);
 printf("\nGross Salary : %.2f",ts);
 getch();
}
```

**OUTPUT:**

```

Enter Total Sales in Rs. 100000
Total Sales: 100000.00
Basic Salary: 3000.00
Hra: 600.00
Da: 3300.00
Conveyance: 500.00
Incentive: 10000.00
Bonus: 500.00
Gross Salary: 17900.00

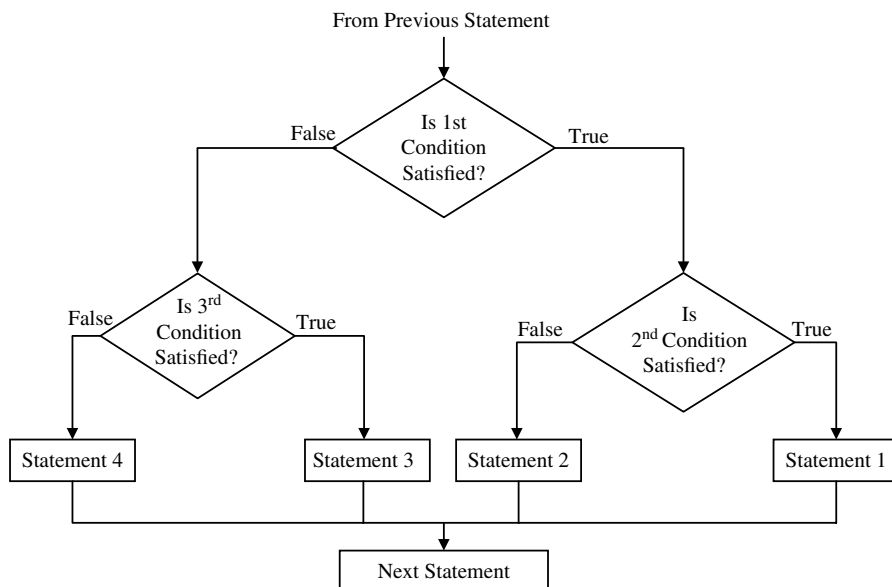
```

**Explanation:**

This program calculates the salary of a medical representative depending on sales. The basic salary is the same but other allowances and incentives change, depending on the sales. If the sale is more than Rs. 1,00,000, the rate of allowances and incentive is as per Table 1 otherwise it is as per Table 2. The `if` condition checks the given figure of sales. If sale is more than Rs. 1,00,000, the first block following the `if` statement is executed otherwise the `else` block is executed. In both the blocks, simple arithmetic operations are performed to calculate the allowances and total salary.

**5.4 NESTED if-else STATEMENTS**

In this kind of statement, a number of logical conditions are checked for executing various statements. Here, if any logical condition is true the compiler executes the block followed by `if` condition, otherwise it skips and executes the `else` block. In the `if-else` statement, the `else` block is executed by default after failure of condition. In order to execute the `else` block depending upon certain condition we can add, repetitively, `if` statements in `else` block. This kind of nesting will be unlimited. Figure 5.5 describes the nested `if-else-if` blocks.

**Figure 5.5** Nested `if-else` statements

Syntax of nested `if-else` statement can be given as follows.

```
if(condition)
{
 /*Inside first if block*/
 if(condition)
 {
 statement 1; /*if block*/
 statement 2;
 }
 else
 {
 statement 3; /*else block*/
 statement 4;
 }
}
else
{
 /*Inside else block*/
 if(condition)
 {
 statement 5; /*if block*/
 statement 6;
 }
 else
 {
 statement 7; /*else block*/
 statement 8;
 }
}
```

From the above block, following rules can be described for applying nested `if-else-if` statements.

1. Nested `if-else` can be chained with one another.
2. If the condition is true control passes to the block following first `if`. In that case, we may have one more `if` statement whose condition is again checked. This process continues till there is no `if` statement in the last `if` block.
3. If the condition is false control passes to `else` block. In that case, we may have one more `if` statement whose condition is again checked. This process continues till there is no `if` statement in the last `else` block.

## 5.5 THE `if-else-if` LADDER STATEMENT

In this kind of statement, a number of logical conditions are checked for executing various statements. Here, if the first logical condition is true the compiler executes the block followed by first `if` condition, otherwise it skips that block and checks for next logical condition followed by `else-if`, if the condition is true the block of statements followed by that `if` condition is executed. The process is continued until a true condition is occurred or an `else` block is occurred. If all `if` conditions become false, it executes the `else` block. In the `if-else-if` ladder statement, the `else` block may or may not have the `else` block.

In `if-else-if` ladder statement we do not have to pair `if` statements with the `else` statements that is we do not have to remember the number of braces opened like nested `if-else`. So it is simpler to code than nested `if-else` and having same effect as nested `if-else`.

The statement is named as **if-else-if** ladder because it forms a ladder like structure as shown in Figure 5.6.

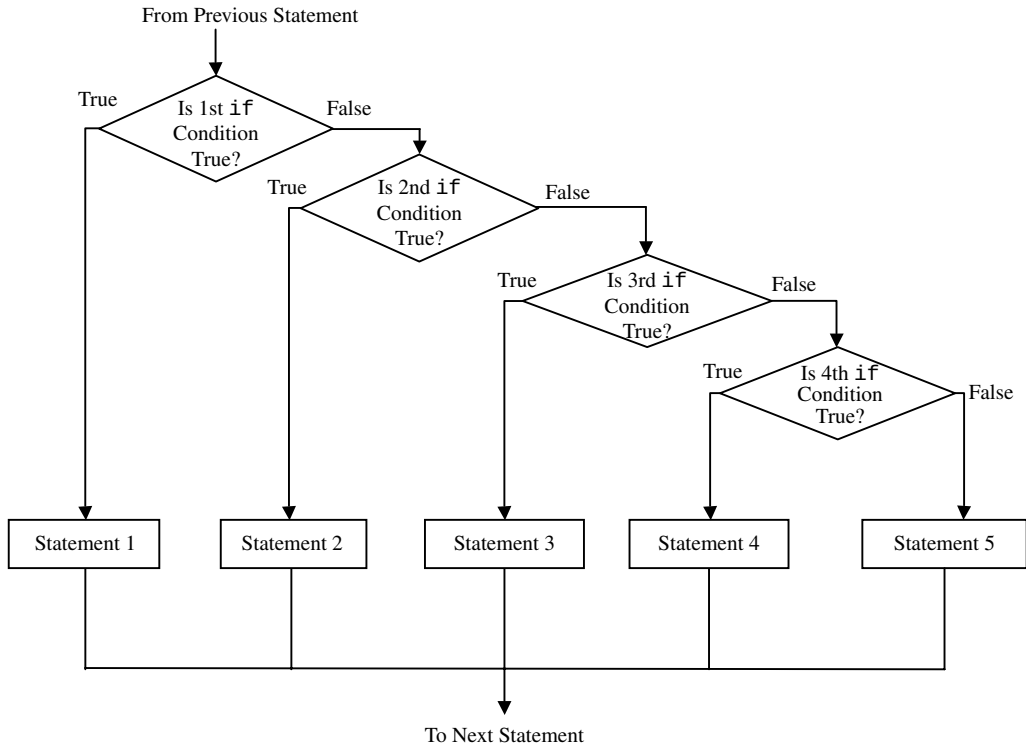


**Figure 5.6** **if-else-if** ladder statement

Syntax of **if-else-if** statement can be given as follows.

```

if(condition)
{
 statement 1; /*if block*/
 statement 2;
}
else if(condition)
{
 statement 3; /*second if block*/
 statement 4;
}
else if(condition)
{
 statement 5; /*third if block*/
 statement 6;
}
else
{
 statement 7; /*else block*/
 statement 8;
}

```

From the above block, following rules can be described for applying nested `if-else-if` statements:

1. Nested `if-else` can be chained with one another.
2. If the first `if` condition is false control passes to `else-if` block where condition is again checked with the next `if` statement. This process continues till there is no `if` statement in the last `else` block.
3. If one of the `if` statements satisfies the condition, other nested `else-if` statement will not be executed.

Given below programs are described on the bases on nested `if-else` and `if-else-if` ladder statements.

- 5.7 Write a program to calculate electricity bill. Read the starting and ending meter reading. The charges are as follows.

| No. of units consumed | Rates in (Rs.) |
|-----------------------|----------------|
| 200-500               | 3.50           |
| 100-200               | 2.50           |
| Less than 100         | 1.50           |

```
void main()
{
 int initial, final, consumed;
 float total;
 clrscr();
 printf("\n Initial & Final Readings :");
 scanf("%d %d", &initial, &final);
 consumed = final - initial;
 if (consumed >= 200 && consumed <= 500)
 total = consumed * 3.50;
 else if (consumed >= 100 && consumed <= 199)
 total = consumed * 2.50;
 else if (consumed < 100)
 total = consumed * 1.50;
 printf("Total bill for %d unit is %f", consumed, total);
 getch();
}
```

#### OUTPUT:

```
Initial & Final Readings : 800 850
Total bill for 50 unit is 75.000000
```

#### Explanation:

Initial and final readings are entered through the keyboard. Their difference is nothing but the total energy consumed. As per the table given in the example, rates are applied and total bill based on consumption of energy is calculated.

- 5.8 Write a program to find the maximum number out of six numbers invoked through the keyboard.

```

void main()
{
int a,b,c,d,e,f;
clrscr();
printf("Enter 1st number:");
scanf("\n%d",&a);
printf("Enter 2nd number:");
scanf("\n%d",&b);
printf("Enter 3 rd number:");
scanf("\n%d",&c);
printf("Enter 4th number:");
scanf("\n%d",&d);
printf("Enter 5th number:");
scanf("\n%d",&e);
printf("Enter 6th number:");
scanf("\n%d",&f);
if((a>b)&&(a>c)&&(a>d)&&(a>e)&&(a>f))
printf("Maximum out of six Numbers is : %d",a);
else if((b>c)&&(b>d)&&(b>e)&&(b>f))
printf("Maximum out of six Numbers is : %d",b);
else if((c>d)&&(c>e)&&(c>f))
printf("Maximum out of six Numbers is :%d",c);
else if((d>e)&&(d>f))
printf("Maximum out of six Numbers is : %d",d);
else if(e>f)
printf("Maximum out of six Numbers is : %d",e);
else
printf("Maximum out of six Numbers is : %d",f);
getch();
}

```

#### OUTPUT:

```

Enter 1st number:23
Enter 2nd number:45
Enter 3rd number:67
Enter 4th number:89
Enter 5th number:80
Enter 6th number:90
Maximum out of six Numbers is : 90

```

#### *Explanation:*

Six numbers are entered through the keyboard. All the values of variables are compared with one another using && (AND) in nested **if-else-if** statements. When one of the **if** statements satisfies the condition that **if** block is executed which prints the largest number otherwise the control passes to another **if-else-if** statement.



- 5.9 Write a program to find the largest number out of three numbers. Read the numbers through the keyboard.

```
void main()
{
 int x,y,z;
 clrscr();
 printf("\nEnter Three Numbers x, y, z :");
 scanf("%d %d %d", &x,&y,&z);
 printf("\nLargest out of Three Numbers is :");
 if(x>y)
 {
 if(x>z)
 printf("x=%d\n",x);
 else
 printf("z=%d\n",z);
 }
 else
 {
 if(z>y)
 printf("z=%d\n",z);
 else
 printf("y=%d\n",y);
 }
}
```

**OUTPUT:**

```
Enter Three Numbers x, y, z : 10 20 30
Largest out of Three Numbers is :z=30
```

**Explanation:**

This is also an example of the nested `if`s. When the `if` statement satisfies the condition, control passes to another `if` statement block. Three numbers are entered through keyboard. The first `if` statement compares the first number with the second number. If the condition is a true, the block followed by first `if` statement executes. Inside the block, the `if` statement checks whether the first number is larger than the third number. If yes, then the largest number is the first one and the same is displayed. Else the third number is the largest and the same is printed. In case the first `if` statement fails to satisfy the condition, the `else` block with nested `if` would be executed. The third number is compared with the second number. If it is true the third number is the largest otherwise the second number is the largest.

- 5.10 Write a program to find the smallest out of the three numbers.

```
void main()
{
 int a,b,c,smallest;
 clrscr();
 printf("\n Enter Three Numbers :");
```

```

scanf("%d %d %d", &a,&b,&c);
if(a<b)
{
 if(a<c)
 smallest=a;
 else
 smallest=c;
}
else
{
 if(b<c)
 smallest =b;
 else
 smallest =c;
}
printf("The smallest of %d %d %d is %d \n", a,b,c, smallest);
getche();
}

```

**OUTPUT:**

Enter Three Numbers : 1 5 8  
The smallest of 1 5 8 is 1

**Explanation:**

The logic in the above program is the same as the last one. Instead of > (greater than) < (less than) condition is used.

- 5.11 Write a program to calculate the gross salary for the conditions given below.

| Basic Salary (Rs.)  | DA (Rs.)      | HRA (Rs.)    | Conveyance (Rs.) |
|---------------------|---------------|--------------|------------------|
| BS>=5000            | 110% of Basic | 20% of Basic | 500              |
| BS>=3000 && bs<5000 | 100% of Basic | 15% of Basic | 400              |
| bs<3000             | 90% of Basic  | 10% of Basic | 300              |

```

void main()
{
 float bs,hra,da,cv,ts;
 clrscr();
 printf("\n Enter Basic Salary :");
 scanf("%f",&bs);
 if(bs>=5000)
 {
 hra=20 * bs/100;
 da= 110 * bs/100;
 cv=500;
 }
 else
 if(bs>=3000 && bs<5000)
 {
 hra=15*bs/100;

```

```

 da=100*bs/100;
 cv=400;
 }
 else
 {
 if(bs<3000)
 hra=10*bs/100;
 da= 90*bs/100;
 cv=300;
 }
 ts=bs+hra+da+cv;
 printf("\nBasic Salary : %5.2f",bs);
 printf("\nHra : %5.2f",hra);
 printf("\nDa : %5.2f",da);
 printf("\nConveyance : %5.2f",cv);
 printf("\nGross Salary : %5.2f",ts);
 getch();
}

```

**OUTPUT:**

```

Enter Basic Salary: 5400
Basic Salary: 5400
Hra: 1080
Da: 5940
Conveyance: 500
Gross Salary:12920

```

**Explanation:**

In the above program, the basic salary of an employee is entered through the keyboard. This entered figure is checked with different conditions as cited in the problem. The `if-else` conditions are used and on the basis of the conditions gross salary is calculated and displayed.

- 5.12 Calculate the total interest based on the following.

| PRINCIPLE AMOUNT (Rs.) | Rate of Interest (Rs.) |
|------------------------|------------------------|
| >=10000                | 20%                    |
| >=8000 && <=9999       | 18%                    |
| <8000                  | 16%                    |

```

void main()
{
 float princ,nyrs,rate,interest;
 clrscr();
 printf("\n Enter Loan & No. of years :-");
 scanf("%f %f", &princ, &nyrs);
 if(princ>=10000)
 rate=20;
}

```

```

else
if(princ>=8000 && princ<=9999)
rate=18;
else
if(princ<8000)
rate=16;
interest = princ * nyrs * rate/100;
printf("\nYears : %6.2f",nyrs);
printf("\nLoan : %6.2f",princ);
printf("\nRate : %6.2f",rate);
printf("\nInterest : %6.2f",interest);
getche();
}

```

**OUTPUT:**

```

Enter Loan & No. of years :- 5000 3
Loan : 5000.00
Years : 3.00
Rate : 16.00
Interest : 2640.00

```

**Explanation:**

In the above program, the loan and the number of years are entered through the keyboard. The entered principal amount is checked with the `if` statement. Based on the principal amount, the rate of interest is charged. The interest is calculated by considering different factors such as loan amount, the number of years and the rate of interest as per the table.

- 5.13 Write a program to find the average of six subjects and display the results as follows.

| AVERAGE                              | RESULT          |
|--------------------------------------|-----------------|
| >=35 & <50                           | Third Division  |
| >=50 & <60                           | Second Division |
| >=60 & <75                           | First Division  |
| >=75 & <=100                         | Distinction     |
| If marks in any subject less than 35 | Fail            |

```

void main()
{
 int a,b,c,d,e,f;
 float sum=0;
 float avg;
 clrscr();
 printf("\nEnter Marks:\n");
 printf("P C B M E H\n");
 scanf("%d %d %d %d %d %d",&a,&b,&c,&d,&e,&f);

 if(a<35 || b<35 || c<35 || d<35 || e<35 || f<35)
 {
 printf("\nResult: Fail");
 exit();
 }
}

```

```

sum=a + b + c + d + e + f;
avg=sum/6;
printf("Total : %g \nAverage : %g", sum,avg);

if(avg>=35 && avg<50)
printf("\n Result: Third Division");
else
if(avg>=50 && avg <60)
printf("\n Result: Second Division");
else
if(avg>=60 && avg<75)
printf("\n Result: First Division");
else
if(avg>75 && avg <=100)
printf("\nResult : Distinction");
getche();
}
Enter Marks:
P C B M E H
56 57 56 89 78 45
Total : 381
Average : 63.5
Result: First Division

```

**Explanation:**

In the above program, marks of six subjects are entered through the keyboard. Their sum and average are calculated. The first **if** statement checks the condition whether the marks in individual subjects are less than 35. If so, message displayed will be 'Result: Fail' and the program terminates. The logical OR (| |) is used here.

The average marks obtained are checked with different conditions. The **if-else** blocks are used. Based on the conditions the statements are executed.

## 5.6 THE break STATEMENT

The keyword **break** allows the programmers to terminate the loop. The **break** skips from the loop or block in which it is defined. The control then automatically goes to the first statement after the loop or block. The **break** can be associated with all conditional statements.

We can also use the **break** statements in the nested loops. If we use the **break** statement in the innermost loop, then the control of the program is terminated only from the innermost loop.

The difference between the **break** and **exit()** is provided in Table 5.1.

**Table 5.1** Difference between **break** and **exit()**

| Sr. No | break                               | exit()                                               |
|--------|-------------------------------------|------------------------------------------------------|
| 1.     | It is a keyword.                    | It is a function.                                    |
| 2.     | No header file is needed.           | Header file <code>process.h</code> must be included. |
| 3.     | It stops the execution of the loop. | It terminates the program.                           |

## 5.7 THE `continue` STATEMENT

The `continue` statement is exactly opposite of the `break` statement. The `continue` statement is used for continuing the next iteration of the loop statements. When it occurs in the loop, it does not terminate, but it skips the statements after this statement. It is useful when we want to continue the program without executing any part of the program. Table 5.2 gives the differences between `break` and `continue`.

**Table 5.2** *Difference between `break` and `continue`*

| <b>break</b>                          | <b>continue</b>                              |
|---------------------------------------|----------------------------------------------|
| Exits from current block or loop.     | Loop takes the next iteration.               |
| Control passes to the next statement. | Control passes at the beginning of the loop. |
| Terminates the loop.                  | Never terminates the program.                |

## 5.8 THE `goto` STATEMENT

This statement does not require any condition. This is an unconditional control jump. This statement passes control anywhere in the program, i.e. control is transferred to another part of the program without testing any condition. User has to define the `goto` statement as follows:

```
goto label;
```

where, the label name must start with any character.

Here, the label is the position where the control is to be transferred. A few examples are described for the sake of understanding.

- 5.14 Write a program to detect the entered number as to whether it is even or odd. Use the `goto` statement.

```
include <stdlib.h>
void main()
{
 int x;
 clrscr();
 printf("Enter a Number :");
 scanf("%d",&x);
 if(x%2==0)
 goto even;
 else
 goto odd;
 even :
 printf("\n %d is Even Number.",x);
 return;
 odd:
 printf("\n %d is Odd Number.",x);
}
```

### OUTPUT:

```
Enter a Number : 5
5 is Odd Number.
```

**Explanation:**

In the above program, a number is entered. The number is checked for even or odd with modules division operator. When the number is even, the `goto` statement transfers the control to the label `even`. Similarly, when the number is odd the `goto` statement transfers the control to the label `odd` and respective message will be displayed.

- 5.15 Write a program to calculate the telephone bill. Transfer controls at different places according to the number of calls and calculate the total charges. Follow rates as per given in the table.

| Telephone Call       | Rate in Rs. |
|----------------------|-------------|
| <100      No Charges |             |
| >99 & <200           | 1           |
| >199 & <300          | 2           |
| >299                 | 3           |

```
void main()
{
 int nc;
 clrscr();
 printf("\nEnter Number of Calls :");
 scanf("%d",&nc);
 if(nc<100)
 goto free;
 else if(nc>99 && nc<200)
 goto charge1;
 else if(nc>199 && nc<300)
 goto charge2;
 else
 goto charge3;
 free :
 printf("\n No charges.");
 return;
 charge1:
 printf("\n Total Charges : %d Rs.",nc*1);
 return;
 charge2:
 printf("\n Total Charges : %d Rs.",nc*2);
 return;
 charge3:
 printf("\n Total Charges : %d Rs.",nc*3);
}
```

**OUTPUT:**

```
Enter Number of Calls : 500
Total Charges: 1500 Rs
```

**Explanation:**

The execution process of the above program is the same as that of the previous one. The difference is that in this program control is transferred to various labels.

- 5.16 Write a program to check if the entered year is a leap year or not. Use the `goto` statement.

```
#include <stdio.h>
#include <conio.h>
int main()
{
 int year;
 clrscr();
 printf("\nEnter Year :");
 scanf("%d",&year);
 if(year%4==0 && year%100!=0 || year%400==0)
 goto leap;
 else
 goto noleap;
 leap:
 printf("%d is a leap year.",year);
 return 0;
 noleap:
 printf("%d is not leap year.",year);
 getch();
 return 0;
}
```

**OUTPUT:**

```
Enter Year : 2012
2012 is a leap year.
```

**Explanation:**

The entered year is divided using the modulus division operator 400. The condition is satisfied hence the year 2012 is the leap year.

## 5.9 THE `switch` STATEMENT

The `switch` statement is a multi-way branch statement. In the program, if there is a possibility to make a choice from a number of options, this structured selection is useful. The `switch` statement requires only one argument of any data type, which is checked with the number of `case` options. The `switch` statement evaluates expression and then looks for its value among the case constants. If the value matches with `case` constant, this particular `case` statement is executed. If not, `default` is executed. Here, `switch`, `case` and `default` are reserved keywords. Every `case` statement terminates with `'.'`. The `break` statement is used to exit from current `case` structure. The `switch` statement is useful for writing the menu-driven program.

The syntax of the `switch case` statement is as follows.

```
switch(variable or expression)
{
 case constant A :
 statement;
```



```

 break;
 case constant B :
 statement;
 break;
 default :
 statement ;
}

```

(a) The **switch** expression

In the block, the variable or expression can be a character or an integer. The integer expression following the keyword **switch** will yield an integer value only. The integer may be any value 1, 2, 3, and so on. In case a character constant, the values may be given in the alphabetic order such as 'x', 'y', 'z'.

(b) The **switch** organization

The **switch** expression should not be terminated with a semi-colon and/or with any other symbol. The entire case structure following **switch** should be enclosed with curly braces. The keyword **case** is followed by a constant. Every constant terminates with a colon. Each **case** statement must contain different constant values. Any number of **case** statements can be provided. If the case structure contains multiple statements, they need not be enclosed with curly braces. Here, the keywords **case** and **break** perform the job of opening and closing curly braces, respectively.

(c) The **switch** execution

When one of the cases satisfies, the statements following it are executed. In case, there is no match, the **default** case is executed. The **default** can be put anywhere in the **switch** expression. The **switch** statement can also be written without the **default** statement. The **break** statement used in **switch** causes the control to go outside the **switch** block. By mistake, if no **break** statements are given all the cases following it are executed (see Figure 5.7).

➤ 5.17 Write a program to print lines by selecting the choice.

```

void main()
{
 int ch;
 clrscr();
 printf("\n[1]");
 printf("\n[2] _____");
 printf("\n[3] *****");
 printf("\n[4] =====");
 printf("\n[5] EXIT");
 printf("\n\n ENTER YOUR CHOICE :");
 scanf("%d", &ch);
 switch(ch)
 {
 case 1 :
 printf("\n");

```

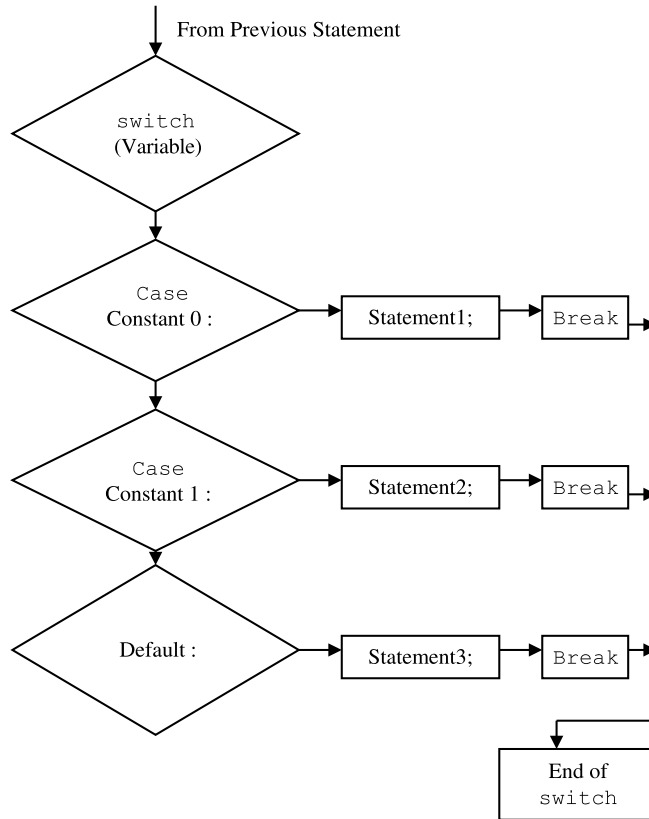


Figure 5.7 switch statement

```

break;
case 2 :
printf("\n _____");
break;
case 3 :
printf("\n *****");
break;
case 4 :
printf("\n =====");
break;
case 5 :
printf("\n Terminated by choice");
exit();
break;
default :
printf("\n Invalid Choice");

```

```

 }
 getch();
}

```

**OUTPUT:**

```

[1]
[2] _____
[3] *****
[4] =====
[5] EXIT
ENTER YOUR CHOICE : 1
.....

```

**Explanation:**

In this program, a menu appears with five options and it requests the users to enter their choice. The choice entered by the user is then passed to `switch` statement. In the `switch` statement, the value is checked with all the case constants. The matched `CASE` statement is executed in which the line is printed of the user's choice. If the user enters a non-listed value, then no match occurs and `default` is executed. The `default` warns the user with a message 'Invalid Choice'.

- 5.18 Write a program to calculate (1) Addition, (2) Subtraction, (3) Multiplication, (4) Division, (5) Remainder calculation, (6) Larger out of two numbers by using `switch` statements.

```

void main()
{
 int a,b,c,ch;
 clrscr();
 printf("\t =====");
 printf("\n\t MENU");
 printf("\n\t =====");
 printf("\n\t[1] ADDITION");
 printf("\n\t[2] SUBTRACTION");
 printf("\n\t[3] MULTIPLICATION");
 printf("\n\t[4] DIVISION");
 printf("\n\t[5] REMAINDER");
 printf("\n\t[6] LARGER OUT OF TWO");
 printf("\n\t[0] EXIT");
 printf("\n\t=====");
 printf("\n\n\t ENTER YOUR CHOICE :");
 scanf("%d", &ch);
 if(ch<=6 & ch>0)
 {
 printf("Enter Two Numbers :");
 scanf("%d %d",&a,&b);
 }

 switch (ch)
 {

```

```

 case 1 :
 c=a+b;
 printf("\n Addition : %d",c);
 break;
 case 2 :
 c=a-b;
 printf("\n Subtraction : %d",c);
 break;
 case 3 :
 c=a*b;
 printf("\n Multiplication : %d",c);
 break;
 case 4 :
 c=a/b;
 printf("\n Division : %d",c);
 break;
 case 5 :
 c=a%b;
 printf("\n Remainder : %d",c);
 break;
 case 6 :
 if(a>b)
 printf("\n\t %d (a) is larger than %d (b).",a,b);
 else
 if(b>a)
 printf("\n\t %d (b) is larger than %d (a).",b,a);
 else
 printf("\n\t %d (a) & %d (b) are same.",a,b);
 break;
 case 0 :
 printf("\n Terminated by choice");
 exit();
 break;
 default :
 printf("\n Invalid Choice");
 }
 getch();
}

```

**OUTPUT:**

MENU

=====

```

[1] ADDITION
[2] SUBTRACTION
[3] MULTIPLICATION
[4] DIVISION
[5] REMAINDER
[6] LARGER OUT OF TWO
[0] EXIT

```

=====

ENTER YOUR CHOICE : 6

Enter Two Numbers : 8 9

9 (b) is larger than 8 (a).

**Explanation:**

In this program also, a menu appears with different arithmetic operations. It requests the user to enter the required choice number. The choice entered by the user is checked with the `if` statement. If it is in between 1 and 6 the `if` block is executed which prompts the user to enter two numbers. After this, the choice entered by the user is passed to the `Switch` statement and it performs relevant operations.

- 5.19 Write a program that converts number of years into (1) minutes, (2) hours, (3) days, (4) months and (5) seconds using `Switch` statements.

```
void main()
{
 long ch,min,hrs,ds,mon,yrs,se;
 clrscr();
 printf("\n[1] MINUTES");
 printf("\n[2] HOURS");
 printf("\n[3] DAYS");
 printf("\n[4] MONTHS");
 printf("\n[5] SECONDS");
 printf("\n[0] EXIT");
 printf("\n Enter Your Choice :");
 scanf("%ld", &ch);
 if(ch>0 && ch<6)
 {
 printf("Enter Years :");
 scanf("%ld",&yrs);
 }
 mon=yrs*12;
 ds=mon*30;
 ds=ds+yrs*5;
 hrs=ds*24;
 min=hrs*60;
 se=min*60;
 switch(ch)
 {
 case 1 :
 printf("\n Minutes : %ld",min);
 break;
 case 2 :
 printf("\n Hours : %ld",hrs);
 break;
 case 3 :
 printf("\n Days : %ld",ds);
 break;
 case 4 :
 printf("\n Months : %ld",mon);
 break;
 case 5 :
 printf("\n Seconds: %ld",se);
 break;
 case 0 :
 printf("\n Terminated by choice");
 }
```

```

 exit();
 break;
 default :
 printf("\n Invalid Choice");
 }
 getch();
}

```

**OUTPUT:**

```

[1] MINUTES
[2] HOURS
[3] DAYS
[4] MONTHS
[5] SECONDS
[0] EXIT
Enter Your Choice : 4
Enter Years : 2
Months : 24

```

**Explanation:**

In this program, the number of years is entered and according to the user's choice **switch case** structure performs the operation.

- 5.20 Write a program to perform the following operations:
1. Display any numbers or stars on the screen by using **for** loop.
  2. Display the menu containing the following: (a) whole screen, (b) half screen, (c) the top three lines and (d) the bottom three lines.

```

void main()
{
 int i,c;
 clrscr();
 for (i=0;i<=700;i++)
 printf("%d",i);
 printf("\n\n\t\tCLRAR SCREE MENU\n");
 printf("\t\t1] Whole screen\n");
 printf("\t\t2] Half Screen\n");
 printf("\t\t3] Top 3 Lines \n");
 printf("\t\t4] Bottom 3 Lines\n");
 printf("\t\t5] Exit \n Enter Your Choice :");
 scanf("%d",&c);
 switch(c)
 {
 case 1:
 clrscr();
 break;
 case 2 :
 for (i=0;i<=189;i++)
 {
 gotoxy(i,1);
 printf("\t");
 }

```

```

 }
 break;
 case 3 :
 for (i=1;i<=99;i++)
 {
 gotoxy(i,1);
 printf("\t");
 }
 break;
 case 4 :
 for (i=1;i<120;i++)
 {
 gotoxy(i,21);
 printf("\t");
 }
 default :
 break;
 }
 getch();
}

```

**Explanation:**

In the above program, the **for** loop is used for displaying numbers on the screen. The screen will be covered with numbers. The screen as a whole or part or top or bottom portion can be cleared by using **switch cases**. While using **switch cases** for loops are used. The programmer can execute the program and see its effect by entering the different choices.

- 5.21 Write a program to display the following files of current directory by using system the DOS command: (1) .exe files, (2) .bat files, (3) .obj files and (4) .bak files.

```

void main()
{
 int c;
 clrscr();
 printf("\n FILE LISTING MENU");
 printf("\n 1] .EXE");
 printf("\n 2] .BAT");
 printf("\n 3] .obj");
 printf("\n 4] .bak\n Enter Your Choice -:");
 scanf("%d",&c);
 switch(c)
 {
 case 1 :
 system("dir .exe");
 break;
 case 2:
 system("\dir .c");
 break;
 case 3:
 system("\dir .obj");
 break;
 }
}

```

```

 case 4:
 system("\\dir .bak");
 break;
 default :
 break;
 }
 getch();
}

```

**Explanation:**

In the above program, a menu is displayed. The user can give different choices. The effect can be observed by selecting one of the choices. The user can view all .exe, .bat, .obj, .bak files. The effect is the same as the DOS 'dir' command. The System function is used to call the operating system command.

- 5.22 Write a program to display days in a calendar format of an entered month of year 2015.

```

void main()
{
 int m,h,i=1,a,j,b=1;
 clrscr();
 printf ("\n Enter Month Number of the Year 2015 : ");
 scanf ("%d",&m); /* Program for finding days of any month of 2015*/
 switch(m)
 {
 case 1 :
 a=5;
 j=31;
 break;

 case 2 :
 a=1;
 j=28;
 break;

 case 3 :
 a=1;
 j=31;
 break;

 case 4 :
 a=4;
 j=30;
 break;

 case 5 :
 a=6;
 j=31;
 break;

 case 6 :
 a=2;
 j=30;
 break;
 }
}

```



```

 case 7 :
 a=4;
 j=31;
 break;

 case 8 :
 a=7;
 j=31;
 break;

 case 9 :
 a=3;
 j=30;
 break;

 case 10 :
 a=5;
 j=31;
 break;

 case 11 :
 a=1;
 j=30;
 break;

 case 12 :
 a=3;
 j=31;
 break;

 default :
 printf ("\a\a Invalid Month");
 exit();
}
/* starting day is to be shown/adjusted as per calendar */
printf ("\n\n\n");
printf ("\t\t\tMonth - %d - 2015\n\n",m);
printf (" SUN MON TUE WED THU FRI SAT\n\n");

 switch (a)
 {
 case 1 :
 printf ("\t%d",i);
 break;

 case 2 :
 printf ("\t\t%d",i);
 break;

 case 3 :
 printf ("\t\t\t%d",i);
 break;

 case 4 :
 printf ("\t\t\t\t%d",i);
 break;

 case 5 :
 printf ("\t\t\t\t\t%d",i);
 break;

```



## ➤ 5.23 Write a program to convert decimal to hexadecimal number.

```
include <process.h>
void main()
{
 int x,y=30,z;
 clrscr();
 printf("\nEnter a number :");
 scanf("%d",&x);
 printf("\n Conversion of Decimal to Hexadecimal Number \n");
 for(;;)
 {
 if(x==0)
 exit(1);
 z=x%16;
 x=x/16;
 gotoxy(y--,5);
 switch(z)
 {
 case 10 :
 printf("A");
 break;
 case 11 :
 printf("%c",'B');
 break;
 case 12 :
 printf("%c","C");
 break;
 case 13 :
 printf("D");
 break;
 case 14 :
 printf("E");
 break;
 case 15 :
 printf("F");
 break;
 default :
 printf("%d",z);
 }
 }
}
```

**OUTPUT:**

```
Enter a number : 31
Conversion of Decimal to Hexadecimal Number
1F
```

**Explanation:**

In the above program, an infinite **for** loop is used. In the **for** loop, the **switch case** statement is used for printing letters A to F. In case entered numbers are in between 1 to 9, the same will be displayed. But if the remainders obtained are greater than 9, in **for** loop, the appropriate case statement is executed for displaying hexadecimal symbols A to F.

## 5.10 NESTED switch case

The C supports nested `switch` statements. The inner `switch` can be a part of an outer `switch`. The inner and outer `switch case` constants may be the same. No conflicts arise even if they are the same. A few examples are given below on the basis of nested `switch` statements.

- 5.24 Write a program to detect if the entered number is even or odd. Use nested `switch case` statements.

```
void main()
{
 int x,y;
 clrscr();
 printf("\n Enter a Number :");
 scanf("%d",&x);
 switch(x)
 {
 case 0 :
 printf("\n Number is Even.");
 break;
 case 1 :
 printf("\n Number is Odd .");
 break;
 default :
 y=x%2;
 switch(y)
 {
 case 0:
 printf("\n Number is Even.");
 break;
 default :
 printf("\n Number is Odd.");
 }
 }
 getch();
}
```

### OUTPUT:

```
Enter a Number : 5
Number is Odd.
```

### *Explanation:*

In the above-given program, the first `switch` statement is used for displaying the message such as even or odd numbers when the entered numbers are 0 and 1, respectively.

When the entered number is other than 0 and 1, its remainder is calculated with modulus operator and stored in the variable 'y'. The variable 'y' is used in the inner `switch` statement. If the remainder is '0' the message displayed will be 'Number is Even', otherwise for non-zero it will be 'Number is Odd'. Here, the constants used for inner and outer `switch` statements are the same.

- 5.25 Write a program to count the number of 1s, 0s, blank spaces and others using nested switch statement.

```
void main()
{
 static int x,s,a,z,o;
 char txt[20];
 clrscr();
 printf("\nEnter Numbers :");
 gets(txt);
 while(txt[x]!='\0')
 {
 switch(txt[x])
 {
 case ' ' :
 s++;
 break;
 default :
 switch(txt[x])
 {
 case '1':
 a++;
 break;
 case '0':
 z++;
 break;
 default :
 o++;
 }
 }
 x++;
 }
 printf("\n Total Spaces : %d", s);
 printf("\n Total 1s : %d",a);
 printf("\n Total 0s : %d",z);
 printf("\n Others : %d",o);
 printf("\n String Length: %d",s+a+z+o);
}
```

**OUTPUT:**

```
Enter Numbers : 1110022 222
Total Spaces : 1
Total 1s : 3
Total 0s : 2
Others : 5
String Length:11
```

**Explanation:**

In the above-mentioned program, the outer switch counts only spaces in a given text. For other than spaces, the inner switch statement is used. The inner switch is for counting 1s, 0s and others. The string length is printed at the end of the program, which is the addition of all the counts.

## 5.11 THE switch case AND nested ifs

The distinction between the `switch case` and the nested `ifs` is narrated in Table 5.3.

**Table 5.3** *Distinction between the switch case and nested ifs*

| switch case                                                                                   | nested ifs                                                                |
|-----------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| The <code>switch</code> can only test for equality, i.e. only constant values are applicable. | The <code>if</code> can evaluate relational or logical expressions.       |
| No two case statements have identical constants in the same <code>switch</code> .             | Same conditions may be repeated for the number of times.                  |
| Character constants are automatically converted to integers.                                  | Character constants are automatically converted to integers.              |
| In <code>switch case</code> statement nested <code>if</code> can be used.                     | In nested <code>if</code> statement <code>switch case</code> can be used. |

- 5.26 Write a program to convert integer to character using `if` condition.

```
void main()
{
 int x;
 clrscr();
 printf("\n Enter a Number :");
 scanf("%d",&x);
 if(x=='A')
 printf("%c",x);
}
```

**OUTPUT:**

```
Enter a Number : 65
A
```

**Explanation:**

In this program, the variable 'x' is declared as an integer variable. Its value is entered through the keyboard. The ASCII value of entered number is checked with the `if` statement. If there is a match, the ASCII, value is displayed.

- 5.27 Write a program to use nested `if-else` statements in the `switch` statement. Also show the effect of conversion of integer to character.

```
void main()
{
 int i;
 clrscr();
 printf("\n Enter Any ASCII Number :");
 scanf("%d",&i);
 clrscr();
```

```

switch(i)
{
 case 'A':
 printf("Capital A\n");
 break;
 case 'B':
 printf("Capital B\n");
 break;
 case 'C':
 printf("Capital C\n");
 break;
 default:
 if(i>47 && i<58)
 printf("\n Digit :[%c]",i);
 else if(i>=58 && i<=64)
 printf("\nSymbol :[%c]",i);
 else if(i>64 && i<91)
 printf("\nCapital :[%c]",i);
 else if(i>96 && i<123)
 printf("\n Small :[%c]",i);
 else
 printf("\n Invalid Choice",i);
 getche();
}
}

```

**OUTPUT:**

Enter Any ASCII Number : 65  
Capital A

***Explanation:***

An ASCII number is entered. In the outer `switch` if its ASCII value is equivalent to 'A', 'B' and 'C' then relevant message is displayed. If the ASCII values are other than these three values, then the inner `switch` statement is used to determine its equivalent ASCII symbols. The symbols may be any character including special symbols and digits. If we enter 65, Capital A is displayed.

## SUMMARY

The reader is made aware about the decision-making statements such as `if` and the `if-else` statements in the C programming. Also, the multi-way-decision statement `switch case` is also discussed. How nested `if-else` and `switch case` statements are to be used in programs are also illustrated in a simple and easy way. Ample simple examples have been explained on the decision-making statements. To change the flow of the program, the programmer can use statements such as `break`, `continue` and `goto`. The reader is expected to execute all programs given in this chapter so as to achieve the expertise in handling decision-making statements in the programs. By writing programs for solving more real-life problems, a programmer benefits a lot.

## EXERCISES

### I Fill in the blanks:

1. The `switch` can only test for \_\_\_\_\_.  
 (a) equality  
 (b) non-equality  
 (c) None of them
2. Only \_\_\_\_\_ values are applicable in the `switch` structure.  
 (a) constants  
 (b) variables  
 (c) objects
3. No two case statements have \_\_\_\_\_ in the same switch.  
 (a) identical constants  
 (b) different constants  
 (c) variables
4. In `switch`, character constants are automatically converted to \_\_\_\_\_.  
 (a) floats  
 (b) integers  
 (c) bool values
5. Same conditions may be repeated for number of times in \_\_\_\_\_ structure.  
 (a) nested ifs  
 (b) switch case  
 (c) None of the them

### II True or false:

1. The `else` is associated with `if`.
2. The `else` block is executed when condition is false.
3. The `if` statement can have multiple `else` statements
4. The statement `if (1)` executes `if` block.
5. One `switch` statement can have multiple case statements but only one default statement.
6. The default statement can be written anywhere in the `switch` block.
7. The `break` statement is used to separate two case statements.
8. The `case` statement is always terminated by a semi-colon.
9. The `switch` statement accepts only constant as an argument.
10. Like `if-else` the `switch` statement can be nested.
11. The default statement is compulsory for `switch` statement
12. The `switch` statement testes for equality only.
13. The `if` statement can evaluate relational and logical expressions.
14. The `if` statement works with expression as well as constants.
15. The `default` is a keyword.
16. The `default` statement can be placed at the beginning of `switch`, without any statement in its block
17. The `switch` block should be terminated by a semi-colon.

### III Select the appropriate option from the multiple choices given below:

1. The `switch` statement is used to  
 (a) switch between functions in a program  
 (b) switch from one variable to another variable  
 (c) choose from multiple possibilities which may arise due to different values of a single variable  
 (d) use switching variables
2. The default statement is executed when  
 (a) all the case statements are false  
 (b) one of the case is true  
 (c) one of the case is false  
 (d) None of the above



3. Each case statement in `switch` is separated by

- (a) `break`
- (b) `continue`
- (c) `exit()`
- (d) `goto`

4. The keyword `else` can be used with

- (a) `if` statement
- (b) `switch` statement
- (c) `do...while()` statement
- (d) None of the above

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

```
void main()
{
 char x='H';
 clrscr();
 switch(x)
 {
 case 'H': printf("%c", 'H');
 case 'E': printf("%c", 'E');
 case 'L': printf("%c", 'L');
 case 'l': printf("%c", 'L');
 case 'O': printf("%c", 'O');
 }
}
```

- (a) HELLO
- (b) HELlo
- (c) H
- (d) None of the above

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

```
void main()
{
 char x='G';
 switch(x)
 {
```

```
if(x=='B')
{
 case 'd': printf("%",
'o');
 case 'B': printf("%s",
"Bad");
}
else
case 'G':
printf("%s", "Good");
default : printf("%s",
"Boy");
}
}
```

- (a) Good Boy
- (b) bad boy
- (c) boy
- (d) None of the above

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

```
void main()
{
 char x='d';
 clrscr();
 switch(x)
 {
 case 'b':
 puts("0 1 001");
 break;
 default :
 puts("1 2 3");
 break;
 case 'R' :
 puts("I II III");
 }
}
```

- (a) 1 2 3
- (b) 0 1 001
- (c) I II III
- (d) None of the above

#### IV What is/are the output/s of the following programs?

1.

```
void main()
{
 int number=7;
 clrscr();
 if(number %2==0)
 printf("\n The entered number
 %d is even",number);
 else
```

```
printf("\n The Entered number
%d is odd",number);
getche();
}
```

2.

```
void main()
{
 int b=4,a=1,c=2;
```

```

float x1,x2;
clrscr();
if(b*b>4*a*c)
{
 x1=-b+sqrt(b*b-4*a*c)/2*a;
 x2=-b-sqrt(b*b-4*a*c)/2*a;
 printf("\n x1=%.1f\n x2=%.1f", x1,x2);
}
else
 printf("\n Roots are Imaginary");
getch();
}

```

3.

```

void main()
{
 float bs,hra,da,cv,incentive,
 bonus,sale=110000,ts;
 clrscr();
 if(sale>=100000)
 {
 bs=3000;
 hra=20 * bs/100;
 da=110 * bs/100;
 cv=500;
 incentive=sale*10/100;
 bonus=500;
 }
 else
 {
 bs=3000;hra=20 *
 bs/100;
 da=110 * bs/100;
 cv=500;
 incentive=sale*5/100;
 }
 ts=bs+hra+da+cv+incentive+
 bonus;
 printf("\n %.2f",ts);
 getch();
}

```

4.

```

void main()
{
 int x=6,y=5,z=10;
 clrscr();
 printf("\nAnswer is :");
 if(x > y)
 {
 if(x > z)
 printf("%d\n",x);
 else

```

```

 printf("%d\n",z);
 }
 else
 {
 if(z > y)
 printf("%d\n",z);
 else
 printf("%d\n",y);
 }
 getch();
}

```

5.

```

void main()
{
 int initial=1000,final=1300,
 consumed;
 float total;
 clrscr();
 consumed = final-initial;
 if(consumed>=200)
 total=consumed * 2.50;
 else
 total=consumed * 1.50;
 printf("Total bill
 for %d units is Rs
 %.2f",consumed,total);
 getch();
}

```

6.

```

void main()
{
 int a=2,b=4,c=1,temp;
 clrscr();
 if
 (a<b)
 {
 if(a<c)
 temp=a;
 else
 temp=c;
 }
 else
 {
 if(b<c)
 temp=b;
 else
 temp=c;
 }
 printf("Answer out of three
 numbers (%d %d %d) is %d \n",
 a,b,c, temp);
 getch();
}

```

**V Find the bug/s in the following program/s?**

1.
 

```
void main()
{
 int cet ;
 clrscr();
 printf("\n Enter marks obtained
 in CET examination :");
 scanf("%d",&cet);
 if(cit>120)
 printf("\n Eligible for
 admission in autonomous
 Institute.");
 getch();
}
```
2.
 

```
void main()
{
 clrscr();
 if(0)
 printf("False");
 else
 printf("True");
}
```

3.
 

```
void main()
{
 clrscr();
 if(3>2)
 printf("2 is smaller than 3");
 printf("and");
 else
 printf("3 is greater");
}
```
4.
 

```
void main()
{
 clrscr();
 if(3>2)
 printf("2 is smaller than 3");
 else
 printf(" Numbers are equal");
 else
 printf("3 is greater");
}
```

**VI Attempt the following programs:**

1. Write a program to check whether the blood donor is eligible or not for donating blood. The conditions laid down are given below. Use `if` statement.
  - (a) Age should be greater than 18 years but not more than 55 years.
  - (b) Weight should be more than 45 kg.
2. Write a program to check whether the voter is eligible for voting or not. If his/her age is equal to or greater than 18, display message 'Eligible' otherwise 'Non- Eligible'. Use the `if` statement.
3. Write a program to calculate bill of a job work done as follows. Use `if-else` statement.
  - (a) Rate of typing Rs. 3/page.
  - (b) Printing of 1st copy Rs. 5/page and later every copy Rs. 3/page.
 User should enter the number of pages and print out copies he/she wants.
4. Write a program to calculate the amount of the bill for the following jobs.
  - (a) Scanning and hardcopy of a passport photo Rs. 5.
  - (b) Scanning and hardcopies (more than 10) Rs. 3.
5. Write a program to calculate bill of Internet browsing. The conditions are given below.
  - (a) 1 Hour – 20 Rs.
  - (b) ½ Hour – 10 Rs.
  - (c) Unlimited hours in a day – 90 Rs.
 Owner should enter number of hours spent by customer.
6. Write a program to enter a character through keyboard. Use `switch` case structure and print appropriate message. Recognize the entered character whether it is vowel, consonants or symbol?
7. The table given below is a list of gases, liquids and solids. By entering one by one substances through the keyboard, recognize their state (gas, liquid and solid).

|        |         |
|--------|---------|
| WATER  | OZONE   |
| OXYGEN | PETROL  |
| IRON   | ICE     |
| GOLD   | MERCURY |

8. Write a program to calculate the sum of remainders obtained by dividing with modular division operations by 2 on 1 to 9 numbers.

## VII Answer the following questions:

- Is it possible to use multiple `else` with `if` statement?
- Is it possible to use multiple `default` statements in `switch` statement?
- Write the use of `else` and `default` statements in `if-else` and `switch` statements, respectively.
- Why `goto` statement is avoided?
- Why the `break` statement is essential in the `switch` statement?
- Which other functions or keywords can be used in place of the `break` statement?
- Is it possible to use the `else` statement in place of `default` or vice versa?
- Can we put `default` statement anywhere in the `switch case` structure?
- What are the limitations of the `switch case` statement?
- What is a sequential execution?
- What is the transfer of control?

## ANSWERS

### I Fill in the blanks:

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | a    | 2. | a    | 3. | a    | 4. | b    | 5. | a    |

### II True or false:

| Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. |
|-----|------|-----|------|-----|------|-----|------|-----|------|
| 1.  | T    | 2.  | T    | 3.  | F    | 4.  | T    | 5.  | T    |
| 6.  | T    | 7.  | T    | 8.  | F    | 9.  | T    | 10. | T    |
| 11. | T    | 12. | T    | 13. | T    | 14. | T    | 15. | T    |
| 16. | T    | 17. | F    |     |      |     |      |     |      |

### III Select the appropriate option from the multiple choices given below:

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | a    | 2. | a    | 3. | a    | 4. | a    | 5. | a    |
| 6. | a    | 7. | a    |    |      |    |      |    |      |

**IV What is/are the output/s of the following programs?**

| Q. | Ans.                                      |
|----|-------------------------------------------|
| 1. | The Entered number 7 is odd               |
| 2. | x1=-2.0 x2=-6.0                           |
| 3. | 18900.00                                  |
| 4. | Answer is :10                             |
| 5. | Total bill for 300 units is Rs 750.00     |
| 6. | Answer out of three numbers ( 2 4 1) is 1 |

**V Find the bug/s in the following program/s?**

| Q. | Ans.                                                         |
|----|--------------------------------------------------------------|
| 1. | Replace cet instead of cit in <code>if</code> statement.     |
| 2. | <code>if (0)</code> is always false statement.               |
| 3. | scope of <code>if</code> block is not defined.               |
| 4. | <code>if</code> should not have multiple <code>else</code> . |

# CHAPTER

# 6

# Loop Control

## Chapter Outline

- 6.1** Introduction
- 6.2** The for Loop
- 6.3** Nested for Loops
- 6.4** The while Loop
- 6.5** The do-while Loop
- 6.6** The while Loop within the do-while Loop
- 6.7** Bohm and Jacopini's Theory

## 6.1 INTRODUCTION

In our life, we need to perform tasks which are repetitive in nature in numerous applications. It is a tedious process to perform such kind of tasks repeatedly with pen/pencil and paper. But with computer programming languages and packages, the task becomes easy, accurate and fast. For example, salary of labours of a factory can be calculated simply by a formula (number of hours work carried  $\times$  wage rate per hour). The accounts department of every organization does this calculation every month/week. Such a type of repetitive deeds can easily be done using a loop in a program.

### 6.1.1 | What is a Loop?

A loop is defined as a block of statements, which are repeatedly executed for a certain number of times. The loops are of two types.

- (i) *Counter-controlled repetition*: This is also called the definite repetition action, because the number of iterations to be performed is defined in advance in the program itself. The steps for performing counter-controlled repetitions are as follows.

#### *Steps in Loop*

*Loop variable* : It is a variable used in the loop.

*Initialization*: It is the first step in which starting and final values are assigned to the loop variable. Each time the updated value is checked by the loop itself.

*Incrimination/decrimination*: It is the numerical value added or subtracted to the variable in each round of the loop. The updated value is compared with the final value and if it is found less than the final value the steps in the loop are executed.

The above steps are implemented in numerous programs in this chapter.

- (ii) *Sentinel-controlled repetition*: This is also called the indefinite repetition. One cannot estimate how many iterations are to be performed. In this type, loop termination happens on the basis of certain conditions using the decision-making statement.

The C language supports three types of loop control statements and their syntaxes are described in Table 6.1 (also see Figures 6.1 and 6.2).

**Table 6.1** Loops in C

| <i>for</i>                                                         | <i>while</i>                                                                     | <i>do-while</i>                                                                    |
|--------------------------------------------------------------------|----------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| for(expression -1;<br>expression-2;<br>expression-3)<br>statement; | expression -1;<br>while(expression -2)<br>{<br>statement;<br>expression -3;<br>} | Expression -1; do<br>{<br>statement;<br>expression-3;<br>}<br>while(expression-2); |

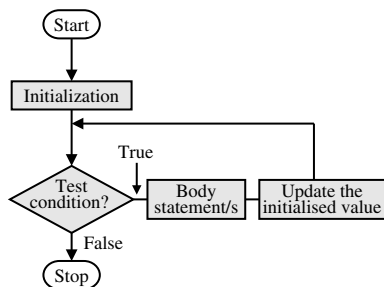


**Figure 6.1** The while statement

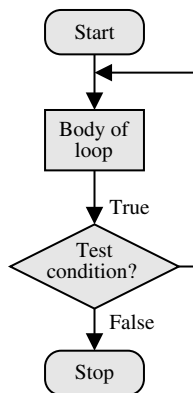


**Figure 6.2** The do-while statement

The `for` loop statement comprises three actions. These actions are placed in the `for` statement itself. The three actions are `initialize counter`, `test condition` and `Re-evaluation parameters`, which are included in one statement. The expressions are separated by semi-colons (;). This leads the programmer to visualize the parameters easily. The `for` statement is equivalent to `while` and `do-while` statements. The only difference between `for` and `while` is that in the latter case logical condition is tested and then the body of the loop gets executed. However, in the `for` statement, test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition fails at the beginning.

```
for(a=10;a<10;a--)
printf("%d", a);
```

For example, in the above two-line program will never execute because the test condition is not proper at the beginning, hence statement following to `for` loop does not execute.

The `do-while` loop executes the body of the loop at least once regardless of the logical condition.

## 6.2 THE `for` LOOP

The `for` loop allows to execute a set of instructions until a certain condition is satisfied. Condition may be predefined or open-ended. The general syntax of the `for` loop will be as given in Table 6.2 (see also Figures 6.3 and 6.4).

### **Explanation:**

The `for` statement contains three expressions which are separated by semi-colons. Following actions are to be performed in the three expressions.

**Table 6.2** Syntax of `for` loop

|                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------|
| <pre>for(initialize counter; test condition; re-evaluation parameter) {     statement1;     statement2; }</pre> |
|-----------------------------------------------------------------------------------------------------------------|



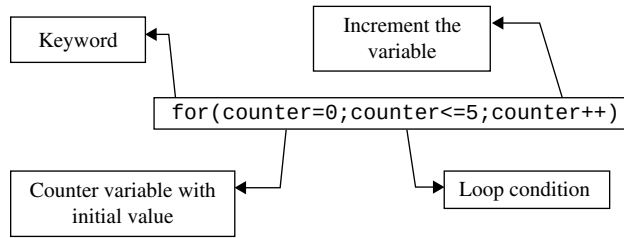


**Figure 6.3** The `for` loop

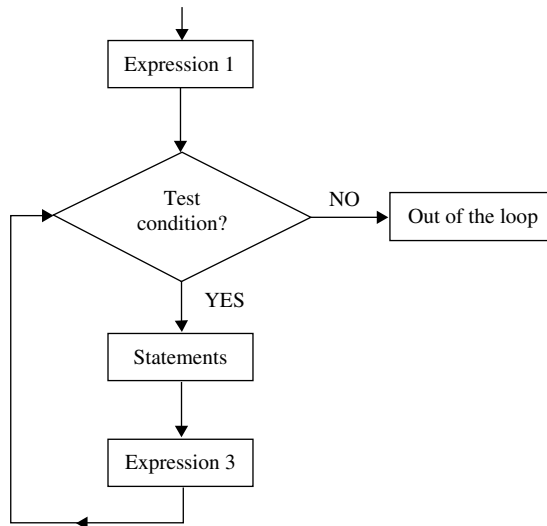


**Figure 6.4** Execution of the `for` loop

1. The initialize counter sets to an initial value. This statement is executed only once.
2. The test condition is a relational expression, that determines the number of iterations desired or determines when to exit from the loop. The '`for`' loop continues to execute as long as conditional test is satisfied. When the condition becomes false the control of the program exits from the body of the '`for`' loop and executes next statement after the body of the loop.
3. The re-evaluation parameter decides how to make changes in the loop (increment or decrement operations are to be used quite often). The body of the loop may contain either a single statement or multiple statements. In case, there is only one statement after the `for` loop, braces may not be necessary. In such a case, only one statement is executed till the condition is satisfied. It is good practice to use braces even for single statement following the `for` loop.

The syntax of the '`for`' loop with only one statement is shown in the third row of Table 6.3. The `for` loop can be specified by different ways and it is as per Table 6.3.

**Table 6.3** Various formats of 'for' Loop

| <i>Syntax</i>                                                | <i>Output</i>               | <i>Remarks</i>                                                                                                                      |
|--------------------------------------------------------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| 1. <code>for( ; ; )</code>                                   | Infinite loop               | No arguments                                                                                                                        |
| 2. <code>for(a=0; a&lt;=20; )</code>                         | Infinite loop               | 'a' is neither incremented nor decremented.                                                                                         |
| 3. <code>for(a =0;a&lt;=10; a++)<br/>printf("%d", a);</code> | Displays value from 0 to 10 | 'a' is incremented from 0 to 10. Curly braces are not necessary. Default scope of the for loop is one statement after the for loop. |
| 4. <code>for(a =10;a&gt;=0;a--)<br/>printf("%d", a)</code>   | Displays value from 10 to 0 | 'a' is decremented from 10 to 0.                                                                                                    |

➤ 6.1 Print the first five numbers starting from 1 together with their squares.

```
void main()
{
 int i;
 clrscr();
 for(i=1;i<=5;i++)
 printf("\n Number: %5d it's Square: %5d",i,i*i);
}
```

**OUTPUT:**

```
Number: 1 it's Square: 1
Number: 2 it's Square: 4
Number: 3 it's Square: 9
Number: 4 it's Square: 16
Number: 5 it's Square: 25
```

**Explanation:**

Let us now examine how the program executes.

1. The value of *i* sets to one for the first time when the program execution starts in the `for` loop.
2. The condition  $i \leq 5$  is specified and tested in each iteration. Initially, the condition is true since the value of *i* is 1. The statements following the `for` loop gets executed.
3. Upon execution of the `printf()` statement, compiler sends control back to the `for` loop where the value of *i* is incremented by one. This is repeated till the value of '*i*' is less than or equal to 5.
4. If the new updated value of *i* exceeds 5, the control exits from the loop.
5. The `printf()` statement executes as long as the value of *i* reaches 5.

The `for` loop can be used by different ways. Various examples using `for` loops are given below.

- 6.2 Display numbers from 1 to 15 using **for** loop. Use **i++**.

```
void main()
{
 int i;
 clrscr();
 for(i=1;i<=15;i++)
 printf("%5d",i);
 getch();
}
```

**OUTPUT:**

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

**Explanation:**

In the above program counter is initialized with a variable  $i = 1$ . Testing and incrementation of counter value are done in the **for** statement itself. Instead of **i++** we can also use **i = i + 1**. Here, we are illustrating a program using **i = i + 1**.

- 6.3 Display numbers from 1 to 15 using **for** loop. Use **i = i + 1**.

```
void main()
{
 int i;
 clrscr();
 for(i=1;i<=15;i=i+1)
 printf("%5d",i);
 getch();
}
```

**OUTPUT:**

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

**Explanation:**

In the above program also, the value of counter is incremented at each time in the **for** statement itself. Instead of **i++**, **i = i + 1** is used in the **for** statement. The same result can be observed as found in the previous example. The incrementation of a counter can be done anywhere in the body of **for** loop and for infinite times. An example is stated below where in the counter value is incremented in the body of the loop and not in the **for** statement.

- 6.4 Write a program to display numbers from 1 to 16. Use incrementation operation in the body of the loop for more than one time.

```
void main()
{
 int i,c=0;
 clrscr();
 for(i=0;i<=15;)
 {
 i++;
```

```

 printf("%5d",i);
 i=i+1;
 printf("%5d",i);
 c++;
 }
 printf("\n\n The Body of the loop is executed for %d times.",c);
}

```

**OUTPUT:**

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
The Body of the loop is executed for 8 times.

```

**Explanation:**

The body of the loop is executed for eight times. Incrementation operation is done following the `for` statement. Hence, the counter which is initialized in the `for` statement is incremented first, i.e. the value of `i` after incrementation is one and the same is displayed with the first `printf()` statement. After this operation, once again in the loop `i` is incremented by 1 and the values are displayed using the second `printf()` statement. The body of the loop is executed till `i=15` and for eight times. In the example given below, the declaration of counter is done before the `for` statement.

- 6.5 Write a program to display even numbers from 0 to 14. Declare the initial counter value before the `for` loop statement.

```

void main()
{
 int i=0;
 clrscr();
 for(;i<=15;)
 {
 printf("%5d",i);
 i+=2;
 }
}

```

**OUTPUT:**

```

0 2 4 6 8 10 12 14

```

**Explanation:**

In this program, the `for` statement contains only test condition. The semi-colon (;) is essential before and after the condition. The `for` statement must contain two semi-colons even though no parameters are provided.

- 6.6 Write a program to display the numbers in ascending (1 to 10) and descending (10 to 1) orders.

```

void main()
{
 int i=0;
 clrscr();
 printf("Numbers in Ascending Order :");
}

```

```

for(++i<=10;)
printf("%3d",i);
printf("\n\n");
printf("Numbers in Descending Order :");
for(i-->1;)
printf("%3d",i);
}

```

**OUTPUT:**

```

Numbers in Ascending Order : 1 2 3 4 5 6 7 8 9 10
Numbers in Descending Order: 10 9 8 7 6 5 4 3 2 1

```

**Explanation:**

In the above example, incrementation and comparison are done in the first `for` loop. The decrementation operation is in the second `for` loop. With `++i<=10`, firstly, `i` is incremented and comparison is made and result is displayed through the `printf()` statement. With `i-->1`, (counter) value is compared first. After satisfying decrementation operation is performed and the value is displayed.

- 6.7 Write a program to display letters from a to j using infinite `for` loop. Use `goto` statement to exit from the loop.

```

void main()
{
 char i=97;
 clrscr();
 for(;;)
 {
 printf("%5c",i++);
 if(i==107)
 goto stop;
 }
 stop;
 getch();
}

```

**OUTPUT:**

```

a b c d e f g h i j

```

**Explanation:**

The `for(;;)` statement is used in the above program. This statement allows the execution of the body of the loop infinite times. To exit from the infinite `for` loop, `goto` statement is used. After satisfying the test condition, `goto` statement is executed which terminates the program. The output is displayed in alphabets from a to j.

- 6.8 Write a program to count numbers between 1 to 100 not divisible by 2, 3 and 5.

```

void main()
{
 int x,c=0;
 clrscr();

```

```

printf("\n Numbers from 1 to 100 not divisible by 2,3 & 5\n\n");
for(x=0 ;x<=100;x++)
{
 if(x%2!=0 && x%3!=0 && x%5!=0)
 {
 printf("%d\t",x);
 c++;
 }
}
printf("\nTotal Numbers : %d",c);
}

```

**OUTPUT:**

```

Numbers from 1 to 100 not divisible by 2,3 & 5
1 7 11 13 17 19 23 29 31 37
41 43 47 49 53 59 61 67 71 73
77 79 83 89 91 97
Total Numbers : 26

```

**Explanation:**

In the above program, **for** loop executes from 1 to 100. Each time mod operation is performed with 2, 3 and 5 with the value of the loop variable. If the remainder is non-zero then the counter 'c' is incremented. Thus, 1 to 100 numbers are checked. At the end, the variable 'c' displays the total number.

- 6.9 Write a program to display the numbers in increasing and decreasing order using the infinite **for** loop.

```

void main()
{
 int n,a,b;
 clrscr();
 printf("Enter a number :");
 scanf("%d",&n);
 a=b=n;
 printf("(++) (--)\n");
 printf("=====");
 for(; ; (a++,b--))
 {
 printf("\n%d\t%d",a,b);
 if(b==0)
 break;
 }
}

```

**OUTPUT:**

```

Enter a number : 5
(++) (--)
=====
5 5
6 4

```

|    |   |
|----|---|
| 7  | 3 |
| 8  | 2 |
| 9  | 1 |
| 10 | 0 |

**Explanation:**

The infinite `for` loop can also be specified as shown in the above program. The initial counter value, which is set to a given number, can be continuously incremented or decremented. In order to terminate from the infinite `for` loop, a condition (`b==0`) is tested. After satisfying break statement allows to exit from the loop.

- 6.10 Create an infinite `for` loop. Check each value of the `for` loop. If the value is even, display it otherwise continue with iterations. Print even numbers from 1 to 21. Use `break` statement to terminate the program.

```
void main()
{
 int i=1;
 clrscr();
 printf("\n\t\t Table of Even numbers from 1 to 20");
 printf("\n\t\t ===== == ===== == = == ==\n");
 for(;;)
 {
 if(i==21)
 break;
 else if(i%2==0)
 {
 printf("%d\t",i);
 i++;
 continue;
 }
 else
 {
 i++;
 continue;
 }
 }
 getch();
}
```

**OUTPUT:**

```
Table of Even numbers from 1 to 20
===== == ===== == = == ==
2 4 6 8 10 12 14 16 18
```

**Explanation:**

In the above program, both `break` and `continue` statements are in use. The program displays only even numbers from 1 to 21. An infinite loop is created and in the loop variable '`i`' is incremented. The value of '`i`' is checked every time. If it is even, it is printed otherwise `continue` statement is executed, which passes control at the beginning of the `for` loop. When the value of '`i`' reaches to 21, the `break` statement is executed and it terminates the loop.

- 6.11 Write a program to display numbers from 1 to 9 and their square roots.

```
include <math.h>
void main()
{
 int i;
 float a;
 clrscr();
 for(i=1;i<=9;i++)
 printf("\n\t %d %.2f ", i, a=sqrt(i));
 getch();
}
```

**OUTPUT:**

```
1 1.00
2 1.41
3 1.73
4 2.00
5 2.24
6 2.45
7 2.65
8 2.83
9 3.00
```

**Explanation:**

In the above program, the `for` loop executes nine times from 1 to 9. The `sqrt()` functions return square root of a number. Each time value of variable '`i`' is passed to `sqrt()` and it returns square root of that number. The header file `math.h` is essential.

- 6.12 Write a program to find the number in between 7 and 100 which is exactly divisible by 4 and if divided by 5 and 6 remainders obtained should be 4.

```
include <process.h>
void main()
{
 int x;
 clrscr();
 for(x=7;x<100;x++)
 {
 if(x%4==0 && x%5==4 && x%6==4)
 {
 printf("\n Number : %d", x);
 }
 }
 getch();
}
```

**OUTPUT:**

```
Number : 64
```



**Explanation:**

In the above program, the `for` loop is initialized from 7 to 100. The `if` statement is used for dividing with modular division operations by 4, 5 and 6. The remainders obtained are checked. If they are 0, 4 and 4 respectively, then the number is displayed.

- 6.13 Write a program to evaluate the series given in comments.

```
/* x=1/1+1/4+1/9...1/n² */
/* y=1/1+1/8+1/27...1/n³ */
include <math.h>
void main()
{
 int i,n;
 float x=0,y=0;
 clrscr();
 printf("Enter Value of n:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 x=x+(1/pow(i,2));
 y=y+(1/pow(i,3));
 }
 printf("Value of x = %f",x);
 printf("\nValue of y = %f",y);
 getch();
}
```

**OUTPUT:**

```
Enter Value of n: 2
Value of x = 1.2500
Value of Y = 1.12500
```

**Explanation:**

Initially, it is assumed that the sum of the two series 1 and 2 is zero. So, the variables '`x`' and '`y`' are set to zero. The denominators of the first and second equations are squares and cubes of numbers from 1 to '`n`' respectively. The user enters the value of '`n`', which determines how long the series should be continued. Every time in the loop square and cube of variable '`i`' are calculated using the library function `pow()`. They are added to variables '`x`' and '`y`', respectively.

- 6.14 Write a program to generate the triangular number.

Note: Triangular number is nothing but summation of 1 to given number.

For example, when entered number is 5, its triangular number would be  $(1+2+3+4+5)=15$ .

```
void main()
{
 int n,j,tri_num=0;
 clrscr();
 printf("What Triangular number do you want :");
```

```

scanf("%d",&n);
for(j=1;j<=n;++j)
tri_num=tri_num+j;
printf("Triangular number of 5 is %d",tri_num);
}

```

**OUTPUT:**

```

What Triangular number do you want? : 5
Triangular number of 5 is 15

```

**Explanation:**

In the above program, a number is entered whose triangular number is to be calculated. The `for` loop is initialized from 1 to 'n'. In each iteration of `for` loop, the value of 'j' is added to the 'tri\_num' variable. When the loop terminates the 'tri\_num' contains triangular number of the entered number.

➤ 6.15 Write a program to find the sum of the following series.

```

/* 1. 1+2+3+..n */
/* 2. 12+22+32+..n2 */
void main()
{
 int sum=0,ssum=0,i,j;
 clrscr();
 printf("Enter Number :");
 scanf("%d", &j);
 clrscr();
 printf(" Numbers:");
 for(i=1;i<=j;i++)
 printf("%5d",i);
 printf("\n\nSquares:");
 for(i=1;i<=j;i++)
 {
 printf("%5d",i*i);
 sum=sum+i;
 ssum=ssum+i*i;
 }
 printf("\n\nSum of Numbers from 1 to %d :%d",j,sum);
 printf("\nSum of Squares of 1 to %d Numbers :%d",j,ssum);
}

```

**OUTPUT:**

```

Enter Number: 5
Numbers: 1 2 3 4 5
Squares: 1 4 9 16 25
Sum of Numbers from 1 to 5: 15
Sum of Squares of 1 to 5 Numbers: 55

```

**Explanation:**

In the above program, sum and square of sum are assumed to be zero. The first **for** loop prints number from 1 to the entered number. The second **for** loop does sum and squares of the numbers from 1 to the entered number. The variable 'sum' and 'ssum' are used for displaying the final results.

- 6.16 Write a program to find the perfect squares from 1 to 500.

```
#include <math.h>
void main()
{
 int i,count,x;
 float c;
 clrscr();
 printf("\n\n");
 printf(" Perfect squares from 1 to 500\n");
 count=0;
 for(i=1;i<=500;i++)
 {
 c=sqrt(i);
 x=floor(c); /* For rounding up floor() is used. */
 if(c==x)
 {
 printf("\t%5d",i);
 count++;
 }
 }
 printf("\n\n Total Perfect Squares =%d\n",count);
 getch();
}
```

**OUTPUT:**

Perfect squares from 1 to 500

|     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | 4   | 9   | 16  | 25  | 36  | 49  | 64  | 81  | 121 | 144 |
| 169 | 196 | 225 | 256 | 289 | 324 | 361 | 400 | 441 | 484 |     |

Total Perfect Squares = 22

**Explanation:**

Numbers like 2, 4, 9, 25, . . . are perfect squares. In the above program a **for** loop is used from 1 to 500. We can use the **sqrt()** function for finding the square root of numbers. The square root obtained is stored in variable 'c', and it is rounded off and stored in another variable 'x'. The value of 'c' remains unchanged. Now the comparison between 'x' and 'c' is made. If both of them are the same, then the number becomes a perfect square root. Perfect square roots are only integers and not floats. Hence, **floor()** function does not affect the integer values.

- 6.17 Write a program to detect the largest number out of five numbers and display it.

```
#include<process.h>
void main()
exit(0);
{
```

```

int a,b,c,d,e,sum=0,i;
clrscr();
printf("\nEnter Five numbers :");
scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
sum=a+b+c+d+e;
for(i=sum; i<=sum;i--)
{
 if(i==a || i==b || i==c || i==d || i==e)
 {
 printf("The Largest Number : %d",i);
 exit(0);
 }
}
}

```

**OUTPUT:**

```

Enter Five numbers : 5 2 3 7 3
The Largest Number : 7

```

**Explanation:**

Through the keyboard, five numbers are entered and their sum is stored in the variable 'sum'. The **for** loop is used up to the value of 'sum' in the reverse order (--). While decrementing the value of sum, the same is tested each time as to whether decremented sum is equal to one of the entered numbers. In case the condition is satisfied, the largest number is displayed and **exit()** terminates the program. The largest number entered through the keyboard appears first while decrementing the value of 'sum' (sum=a+b+c+d+e).

- 6.18 Write a program to print the five entered numbers in the ascending order.

```

void main()
{
 int a,b,c,d,e,sum=0,i;
 clrscr();
 printf("\nEnter Five numbers :");
 scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
 printf("\n Numbers in ascending order :");
 sum=a+b+c+d+e;
 for(i=1; i<=sum;i++)
 {
 if(i==a || i==b || i==c || i==d || i==e)
 {
 printf("%3d",i);
 }
 }
}

```

**OUTPUT:**

```

Enter Five numbers : 5 8 7 4 1
Numbers in ascending order : 1 4 5 7 8

```

**Explanation:**

The logic used is the same as in the previous program. Here, after satisfying the `if` condition the program does not terminate and searches for the next number. This procedure is continued till the value of '`a`' reaches to '`sum`'. The numbers are sorted and displayed in the increasing order.

For descending order the `for` should be '`for(i=sum; i<=sum;i--)`'.

- 6.19 Perform multiplication of two integers by using the negative sign.

```
void main()
{
 int a,b,c,d=0;
 clrscr();
 printf("\n Enter two numbers :");
 scanf("%d %d", &a,&b);
 for(c=1;c<=b;c++)
 d=(d)-(-a);
 printf("Multiplication of %d * %d :%d",a,b,d);
 getch();
}
```

**OUTPUT:**

```
Enter two numbers : 5 5
Multiplication of 5 * 5 : 25
```

**Explanation:**

The multiplication is nothing but repetitive addition. For example, multiplication of  $5 \times 5$  is 25 and the result is obtained by adding 5, five times. Here, the two numbers are entered and the 1st number ('`a`') is repeatedly added to '`b`' (2nd number) times and result is stored in '`d`'. Instead of using '+' sign double negative is used. Multiplication of two -ve signs is +ve ( $- * - = +$ ).

- 6.20 Perform multiplication of two integers by using repetitive addition.

```
void main()
{
 int a,b,c=1,d=0;
 clrscr();
 printf("\n Enter two numbers:");
 scanf("%d %d", &a,&b);
 for(;;)
 {
 d=d+a;
 if(c==b)
 goto stop;
 c++;
 }
 stop:
 printf("Multiplication of %d * %d :%d",a,b,d);
 getch();
}
```

**OUTPUT:**

```
Enter two numbers : 8 4
Multiplication of 8 * 4 : 32
```

**Explanation:**

Here, also the two numbers are entered through the keyboard. The infinite `for` loop is used and counter 'c' is initialized to 1 and 'd' to 0. Variable 'd' is used for storing the result after repetitive addition. After adding the value of 'a' in 'd' each time the counter 'c' is tested with 'b'. If there is no match, counter is incremented and program goes back in the loop and the addition is performed again. When 'c' equals to 'b' (repetitive addition of the first number till the counter is equal to the second number), program terminates using the `goto` statement and the result of multiplication is displayed.

- 6.21 Calculate the sum and average of five subjects.

```
void main()
{
 int a,b,c,d,e,sum=0,i;
 float avg;
 clrscr();
 printf("\nEnter The Marks of Five Subjects");
 for(i=1;i<=5;i++)
 {
 printf("\n[%d] Student:",i);
 if(scanf("%d %d %d %d %d",&a,&b,&c,&d,&e)==5)
 {
 sum=a+b+c+d+e;
 avg=sum/5;
 printf("\n Total Marks of Student[%d] %d",i,sum);
 printf("\n Average Marks of Student[%d] %f\n",i,avg);
 }
 else
 {
 clrscr();
 printf("\n Type Mismatch");
 }
 }
}
```

**OUTPUT:**

```
Enter The Marks of Five Subjects
[1] Student: 58 52 52 56 78
 Total Marks of Student[1] 296
 Average Marks of Student[1] 59.000000
[2] Student:
```

**Explanation:**

Here, the `for` loop is used for entering marks of five students. The marks of five subjects are entered through the keyboard. They are assigned to variables 'a', 'b', 'c', 'd' and 'e'. Their 'sum' and 'average' are calculated. If by mistake, the user enters marks other than integers the program

reports the error message 'Type Mismatch'. The `scanf()` is compared with the number of arguments entered correctly. If it is less than the given argument the program displays 'Type Mismatch'. The marks of the second student onwards can be entered and the results can be observed.

- 6.22 Write a program to enter a character and display its position in alphabets.

```
include <ctype.h>
void main()
{
 int c=1;
 char i, ch;
 clrscr();
 printf("Enter a character :");
 scanf("%c",&ch);
 ch=toupper(ch);
 for(i=65;i<=90;i++,c++)
 if(ch ==i)
 printf("\n%c'is%d [st/nd/rd/th] Character in Alphabetic.",i,c);
}
```

**OUTPUT:**

Enter a character : U

'U' is 21 [st//nd/rd/th] Character in Alphabetic.

**Explanation:**

In this program, a character is entered. It is converted into capital. The ASCII values of A to Z are from 65 to 90. The `for` loop is taken from 65 to 90 and these values are compared with the entered character. Certainly, there will be a match somewhere. For this, statement `(ch==i)` is used. Till the match is achieved counter is incremented. Once the match occurs, we get the position of character from the first alphabetic with the help of a counter.

- 6.23 Write a program to enter mantissa and exponent. Calculate the value of  $m^x$ . Where 'm' is a mantissa and 'x' is an exponent.

```
void main()
{
 int i,number,power;
 long ans;
 clrscr();
 printf("\n Enter Number :");
 scanf("%d",&number);
 printf("\n Enter Power :");
 scanf("%d",&power);
 ans=1;
 for(i=1;i<=power;i++)
 ans*=number;
 printf("\n The Power of %d raised to %d is %ld\n", number,
 power,ans);
 getch();
}
```

**OUTPUT:**

```
Enter Number : 5
Enter Power : 3
The Power of 5 raised to 3 is 125.
```

**Explanation:**

In the above program, mantissa and exponents are entered. The first number is mantissa and the second is its power. The variable 'ans' stores repetitive multiplication of mantissa till the loop is active. Finally, it displays the power of a given number.

**6.3 NESTED for LOOPS**

We can also use loop within loops. In nested **for** loops one or more **for** statements are included in the body of the loop. In other words C allows multiple **for** loops in nested forms. The numbers of iterations in this type of structure will be equal to the number of iteration in the outer loop multiplied by the number of iterations in the inner loop. Given below examples are based on the nested **for** loops.

➤ 6.24 Write a program to perform subtraction of two loop variables. Use nested **for** loops.

```
void main()
{
 int a,b,sub;
 clrscr();
 for(a=3;a>=1;a--)
 {
 for(b=1;b<=2;b++)
 {
 sub=a-b;
 printf("a=%d\t b=%d\t a-b=%d\n",a,b,sub);
 }
 }
}
```

**OUTPUT:**

```
a=3 b=1 a-b = 2
a=3 b=2 a-b = 1
a=2 b=1 a-b = 1
a=2 b=2 a-b = 0
a=1 b=1 a-b = 0
a=1 b=2 a-b = -1
```

**Explanation:**

In the above program, the outer loop variable and inner loop variables are 'a' and 'b', respectively. For each value of 'a' the inner loop is executing twice. The inner loop terminates when the value of 'b' exceeds 2 and the outer loop terminates when the value of 'a' is 0. Thus, the outer loop executes thrice and for each outer loop, the inner loop executes twice. Thus, the total iterations are  $3 \times 2 = 6$  and six output lines are shown in the output.



- 6.25 Write a program to illustrate an example based on the nested `for` loops.

```
void main()
{
 int i,j;
 clrscr();
 for(i=1;i<=3;i++) /* outer loop */
 {
 for(j=1;j<=2;j++) /* inner loop */
 printf("\n i*j : %d",i*j);
 }
}
```

**OUTPUT:**

```
i*j = 1
i*j = 2
i*j = 2
i*j = 4
i*j = 3
i*j = 6
```

**Explanation:**

Here, in the above example for each value of 'i' the inner loop's *j* value varies from 1 to 2. Outer loop executes three times whereas the inner loop executes six times. The inner loops terminate when the value of *j* = 2 and the outer loop terminates when the value of *i* = 3. The total number of iterations = outer loop iterations (3) \* inner loop iterations (2) = 6.

- 6.26 Write a program using nested `for` loops. Print values and messages when any loop ends.

```
void main()
{
 int a,b,c;
 clrscr();
 for(a=1;a<=2;a++) /* outer loop */
 {
 for(b=1;b<=2;b++) /* middle loop */
 {
 for(c=1;c<=2;c++) /* inner loop */
 printf("\n a=%d + b=%d + c=%d : %d", a, b, c, a+b+c);
 printf("\n Inner Loop Over.");
 }
 printf("\n Middle Loop Over.");
 }
 printf("\n Outer Loop Over.");
}
```

**Explanation:**

The execution of the above program will be done by the C compiler in sequence as per Table 6.4. The total number of iterations is equal to  $2 * 2 * 2 = 8$ . The final output provides eight results.

**Table 6.4** Output of program 6.27

| Values of Loop Variables |        |       | Output  |
|--------------------------|--------|-------|---------|
| Outer                    | Middle | Inner |         |
| 1. a=1                   | b=1    | c=1   | a+b+c=3 |
| 2. a=1                   | b=1    | c=2   | a+b+c=4 |
| Inner loop over          |        |       |         |
| 3. a=1                   | b=2    | c=1   | a+b+c=4 |
| 4. a=1                   | b=2    | c=2   | a+b+c=5 |
| Inner loop over          |        |       |         |
| Middle loop over         |        |       |         |
| 5. a=2                   | b=1    | c=1   | a+b+c=4 |
| 6. a=2                   | b=1    | c=2   | a+b+c=5 |
| Inner loop over          |        |       |         |
| 7. a=2                   | b=2    | c=1   | a+b+c=5 |
| 8. a=2                   | b=2    | c=2   | a+b+c=6 |
| Inner loop over          |        |       |         |
| Middle loop over         |        |       |         |
| Outer loop over          |        |       |         |

➤ 6.27 Write a program to find perfect cubes up to a given number.

*/\* 1, 8, 27, 64 are perfect cubes of 1, 2, 3 and 4 \*/.*

```
include<math.h>
void main()
{
 int i, j, k;
 clrscr();
 printf("Enter a Number :");
 scanf("%d",&k);
 for(i=1;i<k;i++)
 {
 for(j=1;j<=i;j++)
 {
 if(i==pow(j,3))
 printf("\nNumber : %d & it's Cube : %d",j,i);
 }
 }
}
```

#### OUTPUT:

```
Enter a Number : 100
Number : 1 & it's Cube : 1
Number : 2 & it's Cube : 8
Number : 3 & it's Cube : 27
Number : 4 & it's Cube : 64
```

**Explanation:**

In the above program, the programmer has to enter the number up to which the perfect cubes are to be obtained. For example, if the user enters 100, the perfect cubes up to 100 are 1, 8, 27 and 64. Here, the two **for** loops are used. The outer **for** loop varies up to the entered number. The inner **for** loop varies up to value of '*i*'. For each value of '*j*', its cube is calculated and checked with the value of '*i*'. If the **if** condition is true, the perfect cubes are printed, otherwise the loop continues. Thus, all the perfect cubes up to a given number are displayed.

- 6.28 Write a program to display numbers 1 to 100 using ASCII values from 48 to 57. Use the nested loops.

```
void main()
{
 int i,j=0,k= -9;
 clrscr();
 printf("\t Table of 1 to 100 Numbers Using ASCII Values \n");
 printf("\t ===== \n");
 for(i=48;i<=57;i++,k++)
 {
 for(j=49;j<=57;j++)
 printf("%5c%c", i, j);
 if(k!=1)
 printf("%5c%c", i+1, 48);
 if(k==0)
 printf("\b\b%d%d%d", k+1, k, k);
 printf("\n\n");
 }
 getch();
}
```

**OUTPUT:**

Table of 1 to 100 Numbers Using ASCII Values

=====

```
01 02 03 04 05 06 07 08 09 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
```

**Explanation:**

The first **for** loop is varying from 48 to 57. These numbers are used for printing the first digit of every number from 1 to 100. 48 to 57 are ASCII codes for 0 to 9. The second **for** loop is varying from ASCII 49 to 57 whose equivalent numerical numbers are 1 to 9. These ASCII codes are used for printing the second digit. In each output line the common digit like 0 in the first line is decided by the variable of the outer loop. It varies from 0 to 9. The second digit of each line is decided by the variable of inner loop, which varies for all lines from 1 to 9. The value of  $k = -9$  for displaying 10 rows. For displaying numbers from 01 to 99, '*i*' and

'j' are printed without using space in one `printf()` statement. The first `if` statement is used for printing 10, 20, 30 up to 90 numbers. The second `if` is used to print 100.

- 6.29 Write a program to count number of votes secured by 'A' & 'B'. Assume three voters are voting them. Also count the invalid votes.

```
include <ctype.h>
void main()
{
 int a=0,b=0,o=0,i;
 char v;
 clrscr();
 printf("\tPress A or B\n");
 for(i=1;i<=3;i++)
 {
 printf("\n\nVoter no. %d",i);
 printf("Enter Vote :");
 v=getche();
 v=toupper(v);
 if(v=='A')
 a++;
 else if(v=='B')
 b++;
 else
 o++;
 }
 printf("\n\n Status of vote(s) \n");
 printf("\nA secures %d vote(s).",a);
 printf("\nB secures %d vote(s).",b);
 printf("\nInvalid votes %d.",o);
}
```

**OUTPUT:**

```
Press A or B
Voter no. 1 Enter Vote :A
Voter no. 2 Enter Vote :A
Voter no. 3 Enter Vote :A
Status of vote(s)
A secures 3 vote(s).
B secures 0 vote(s).
Invalid votes 0.
```

**Explanation:**

In the above program, the `for` loop is used for the number of voters who will be voting to either 'A' or 'B'. The body of the loop is executed for three times. The voter can give his/her choice by pressing either 'A' or 'B'. Apart from these two characters, if the user enters any other character, the vote will be invalid. The results are displayed at the output.

- 6.30 Write a program to simulate a digital clock.

```
include <dos.h>
void main()
{
```

```

int h,m,s;
clrscr();
for(h=1;h<=12;h++)
{
 clrscr();
 for(m=1;m<=59;m++)
 {
 clrscr();
 for(s=1;s<=59;s++)
 {
 gotoxy(8,8);
 printf("hh mm ss\n");
 gotoxy(8,9);
 printf("%d %d %d",h,m,s);
 sleep(1);
 }
 }
}

```

**OUTPUT:**

```

hh mm ss
1 1 1

```

**Explanation:**

The above program uses three **for** loops for hour, minute and second. The inner most loop is for second's operation, followed by middle loop for minutes and the outer most for hours. When the second's loop executes 60 times, the minutes loop increments by 1. Similarly, the hours loop increments after completing minute's loop. The hours loop increments only up to 12. Thereafter, it resets from 1. The **gotoxy()** function is used for selecting the position where hh mm ss is to be marked.

- 6.31 Write a program to count the occurrence of 0 to 9 digits between 1 and the given decimal number.

```

void main()
{
 int t,k,n,l,i;
 static int st[10];
 clrscr();
 printf("\n Enter a Decimal Number :");
 scanf("%d",&n);
 for(l=1;l<=n;l++)
 {
 t=l;
 while(t!=0)
 {
 k=t%10;
 t=t/10;
 st[k]++;
 }
 }
}

```

```

printf("\noccurrence of 0-9 digits between 1 to %d Numbers.",n);
printf("\n===== = == === ===== ===== = =====\n");
for(i=0;i<10;i++)
if(st[i]>0)
printf("\n %d Occurs %8d Times.",i,st[i]);
getch();
}

```

**OUTPUT:**

```

Enter a Decimal Number: 15
Occurrence of 0-9 digits between 1 to 15 Numbers.
0 Occurs 1 Times.
1 Occurs 8 Times.
2 Occurs 2 Times.
3 Occurs 2 Times.
4 Occurs 2 Times.
5 Occurs 2 Times.
6 Occurs 1 Times.
7 Occurs 1 Times.
8 Occurs 1 Times.
9 Occurs 1 Times.

```

**Explanation:**

A decimal number is entered through a keyboard with the variable *n*. The **for** loop is used from 1 to *n* (up to the given number). The value of '*t*' is copied to another variable '*t*'. In the body of the **while** loop **mod** operation is performed on variable '*t*' for separating individual digits, and the separated digit is stored in the variable '*k*'. Further, the '*t*' is divided by 10 for obtaining remainders and is assigned to '*t*'. The array `st[10]` is initialized to 0 using static storage class. The variable '*k*' is used as an argument with `st[]`, and it is incremented. The value of '*k*', whatever it contains when used in arrays `st[]` that element number will be incremented. Thus, we get the occurrence of digits from 0 to 9 by printing this array.

- 6.32 Write a program to accept a number and find sum of its individual digits repeatedly till the result is a single digit.

```

void main()
{
 int n,s=0;
 clrscr();
 printf("\n Enter a Number :");
 scanf("%d",&n);
 printf("\n Sum of Digits till a single digit\n %d",n);
 for(; n!=0 ;)
 {
 s=s+n%10;
 n=n/10;
 if(n==0 && s>9)
 {

```

```

 printf("\n %2d",s);
 n=s;
 s=0;
 }
}
printf("\n %2d",s);
}

```

**OUTPUT:**

```

Enter a Number :4687
Sum of Digits till a single digit
4687
25
7

```

**Explanation:**

In the above program, a number is entered through the keyboard. The **for** loop executes till the value of variable 'n' is non-zero. The sum of digits obtained is assigned to variable 'n'. If 'n' is found zero and sum is greater than 9 the for loop continues. Thus, repeatedly the sum of digits is calculated till the result of sum becomes less than 10.

- 6.33 Write a program to display octal numbers in binary. Attach a parity bit with "1" if number of 1s are even otherwise "0".

OR

Generate odd parity to octal numbers 0 to 7. Express each number in binary and attach the parity bit.

```

void main()
{
 int x,y,a,b,c=0,j=12,k=2;
 clrscr();
 printf(" Binary Bits Parity Bits\n");
 printf(" =====");
 for(x=0;x<8;x++)
 {
 k++;
 j=12;
 y=x;
 for(a=0;a<3;a++)
 {
 b=y%2;
 gotoxy(j-=3,k);
 printf("%d",b);
 y=y/2;
 if(b==1)
 c++;
 }
 if(c%2==0)

```

```

 {
 gotoxy(25,k);
 printf("%d",1);
 }
 else
 {
 gotoxy(25,k);
 printf("%d",0);
 }
 c=0;
 }
}

```

**OUTPUT:**

| Binary Bits<br>===== | Parity Bits<br>===== |
|----------------------|----------------------|
| 0 0 0                | 1                    |
| 0 0 1                | 0                    |
| 0 1 0                | 0                    |
| 0 1 1                | 1                    |
| 1 0 0                | 0                    |
| 1 0 1                | 1                    |
| 1 1 0                | 1                    |
| 1 1 1                | 0                    |

**Explanation:**

In the above program, two **for** loops are used. The inner **for** loop generates three binary bits for each octal number. The 0 to 7 octal numbers are taken from the outer **for** loop. If the bit is 1 counter 'c' is incremented. The value of 'c' is checked for even or odd condition. If 'c' is odd, the parity bit '0' is displayed otherwise '1' will be displayed.

- 6.34 Write a program to evaluate the series given in comments.

*/\* sum of series of  $x - x^3/3! + x^5/5! - x^7/7! + \dots x^n/n!$  \*/*

```

include <math.h>
void main()
{
 int n,i,x,c=3,f=1,l;
 float sum;
 clrscr();
 printf("Enter x & n :");
 scanf("%d %d",&x,&n);
 sum=x;
 for(i=3;i<=n;i+=2)
 {
 f=1;
 if(c%2!=0)
 {
 for(l=1;l<=i;l++)
 f=f*l;
 sum=sum-pow(x,i)/f;
 }
 }
}

```



```

 else
 {
 for(l=1;l<=i;l++)
 f=f*l;
 sum=sum+pow(x,i)/f;
 }
 c++;
 }
 printf("\nSum of series Numbers :%f",sum);
 getch();
}

```

**OUTPUT:**

```

Enter x & n : 2 5
Sum of series Numbers :0.9333

```

**Explanation:**

Through the keyboard, the values of 'x' and 'n' are entered, where numerical constant values are assigned to 'x' and 'n'. Throughout the series, 'n' decides how long the series should continue. The series contains alternate + and – terms. The two **for** loops within **if** are used for calculating the factorials. The **if** statement decides the positive or negative term depending upon odd or even value of 'c'.

The following are the programs based on a series. The logic of the programs is simple for understanding.

- 6.35 Write a program to evaluate the series given in comment.

```

/* x+x2/2!+x4/4!+x6/6!+...xn/n! */

```

```

include <math.h>
void main()
{
 int f=1,l,i,x,y;
 float sum;
 clrscr();
 printf("Enter the value of x & y :");
 scanf("%d %d",&x,&y);
 sum = x;
 for(i=2;i<=y;i+=2)
 {
 f=1;
 for(l=1;l<=i;l++)
 f=f*l;
 sum = sum + pow(x,i)/f;
 }
 printf("\n Sum of Series : %f", sum);
 getch();
}

```

**OUTPUT:**

```

Enter the value of x & y : 4 4
Sum of Series :22.6666

```

- 6.36 Write a program to evaluate the series given in comment.

/\* 1-1/1!+2/2!-3/3!...n/n! \*/

```
include <math.h>
void main()
{
 int n,i,c=3,l;
 float sum=0,f=1,k;
 clrscr();
 printf("Enter value of n :");
 scanf("%d",&n);
 sum=1;
 for(i=1;i<=n;i++)
 {
 f=1;
 if(c%2!=0)
 {
 for(l=1;l<=i;l++)
 f=f*l;
 k=i/f;
 sum=sum-k;
 }
 else
 {
 for(l=1;l<=i;l++)
 f=f*l;
 sum=sum+(i/f);
 }
 c++;
 }
 printf("\nSum of series Numbers :%f",sum);
}
```

**OUTPUT:**

```
Enter value of n : 3
Sum of series Numbers : 0.5000
```

- 6.37 Write a program to detect the Armstrong numbers in three digits from 100 to 999. If the sum of cubes of each digits of the number is equal to the number itself, then the number is called an Armstrong number (e.g.  $153 = 1^3 + 5^3 + 3^3 = 153$ ).

```
include <math.h>
void main()
{
 int n,d,x;
 int k,i,cube=0;
 clrscr();
 printf("\n The following numbers are armstrong numbers.");
 for(k=100;k<=999;k++)
 {
```

```

 cube=0;
 x=1;
 d=3;
 n=k;
 while(x<=d)
 {
 i=n%10;
 cube=cube+pow(i,3);
 n=n/10;
 x++;
 }
 if(cube==k)
 printf("\n\t\ %d",k);
}

```

**OUTPUT:**

The Following Numbers are Armstrong numbers.

```

153
370
371
470

```

**Explanation:**

The program separates individual digit of number and calculates cubes of each digit. If the sum of the cubes of individual digits of a number is equal to that number, then the number is called Armstrong numbers.

- 6.38 Write a program to display the stars as shown below.

```

*
* *
* * *
* * * *
* * * * *

```

```

void main()
{
 int x,i,j;
 clrscr();
 printf("How many lines stars (*) should be printed ? :");
 scanf("%d",&x);
 for(i=1;i<=x;i++)
 {
 for(j=1;j<=i;j++)
 {
 printf("*");
 }
 printf("\n");
 }
}

```

**OUTPUT:**

```
How many lines stars (*) should be printed ? : 5
*
* *
* * *
* * * *
* * * * *
```

**Explanation:**

Here in the above example, the two `for` loops are used for displaying as per the format shown in the output. Variables 'x', 'i' and 'j' are used according to the requirement. On the execution of inner loop star/stars are displayed. To display them in different lines the second `printf()` statement is used.

- 6.39 Write a program to generate the pattern of numbers as given below.

```
5 4 3 2 1 0
4 3 2 1 0
3 2 1 0
2 1 0
1 0
0
```

```
void main()
{
 int i,c=0;
 clrscr();
 printf("Enter a Number :");
 scanf("%d",&i);
 for(;i>=0;i--)
 {
 c=i;
 printf("\n");
 for(; ;)
 {
 printf("%3d",c);
 if(c==0)
 break;
 c--;
 }
 }
}
```

**OUTPUT:**

```
Enter a number: 6
6 5 4 3 2 1 0
5 4 3 2 1 0
4 3 2 1 0
3 2 1 0
2 1 0
1 0
0
```

**Explanation:**

An integer value is read for variable 'i'. The first **for** loop decrements the value of 'i'. The value of 'i' is assigned to variable 'c'. In the second infinite **for** loop, the value of 'c' is printed and decremented. The **if** condition terminates the infinite **for** loop when it finds the value of 'c' is 0.

- 6.40 Write a program to display the series of numbers as given below.

```
1
1 2
1 2 3
1 2 3 4
4 3 2 1
3 2 1
2 1
1
```

```
void main()
{
 int i,j,x;
 printf("\nEnter Value of x :");
 scanf("%d",&x);
 clrscr();
 for(j=1;j<=x;j++)
 {
 for(i=1;i<=j;i++)
 printf("%3d",i);
 printf("\n");
 }
 printf("\n");
 for(j=x;j>=1;j--)
 {
 for(i=j;i>=1;i--)
 printf("%3d",i);
 printf("\n");
 }
}
```

**OUTPUT:**

```
Enter Value of x : 4
1
1 2
1 2 3
1 2 3 4
4 3 2 1
3 2 1
2 1
1
```

**Explanation:**

The first two **for** loops are used to display the first four lines of the output. The next two **for** loops are used to display the last four lines of the output. The outputs of the first four lines are in ascending order whereas in the last four lines the numbers are in descending order.

- 6.41 Write a program to generate the pyramid structure using numerical.

```
void main()
{
 int k,i,j,x,p=34;
 printf("\n Enter A number :");
 scanf("%d",&x);
 clrscr();
 for(j=0;j<=x;j++)
 {
 gotoxy(p,j+1);
 /* position curosr on screen (x cordinate,y cordinate) */
 for(i=0-j;i<=j;i++)
 printf("%3d",abs(i));
 p=p-3;
 }
}
```

**OUTPUT:**

```
Enter a number : 3
 0
 1 0 1
 2 1 0 1 2
3 2 1 0 1 2 3
```

**Explanation:**

Here, in the above program 'p' is equated to 34. This number decides the 'x' co-ordinate on the screen from where numbers are to be displayed. The 'y' co-ordinate is decided by j+1 where 'j' is varying from 0 to the entered number. The value of 'i' is negative towards the left of zero. Hence, its absolute value is taken. The inner for loop is executed for displaying digits towards the left and right of zero.

- 6.42 Write a program to convert a binary number to a decimal number.

```
void main()
{
 int x[5]={1,1,1,1,1};
 int y=x[0],i;
 clrscr();
 printf("\n Values in different Iterations. \n");
 printf("===== == ===== \n");
 for(i=0;i<4;i++)
 {
 y=y*2+x[i+1];
 printf("[%d] %d\t",i+1,y);
 }
 printf("\nEquivalent of [");
 for(i=0;i<5;i++)
 printf("%d",x[i]);
 printf("] in Decimal Number is :");
 printf("%d\t",y);
}
```

```
 getch();
}
```

**OUTPUT:**

Values in different Iterations.

=====

[1] 3 [2] 7 [3] 15 [4] 31

Equivalent of [1 1 1 1 1] in Decimal Number is : 31

**Explanation:**

A binary number is initialized in an array. The first element of the array is assigned to variable 'y'. The logic used here is to multiply by 2 the variable 'y' and add the next successive bit. For this the equation  $y = y * 2 + x[i+1]$  is used. The number of iterations to be carried out is equal to the number of binary bits minus 1. Here, five bits are initialized, hence the number of iterations will be four.

- 6.43 Write a program to add a parity bit with four binary bits such that the total number of one's should be even.

```
void main()
{
 int bit[5], j, c=0;
 clrscr();
 printf("\n Enter four bits :");
 for(j=0; j<4; j++)
 {
 scanf("%d", &bit[j]);
 if(bit[j]==1)
 c++;
 else
 if(bit[j]>1 || bit[j]<0)
 {
 j--;
 continue;
 }
 }
 if(c%2==0)
 bit[j]=0;
 else
 bit[j]=1;
 printf("\n Message bits together with parity bit :");
 for(j=0; j<5; j++)
 printf("%d", bit[j]);
}
```

**OUTPUT:**

Enter four bits : 1 1 1 1

Message bits together with parity bit : 11110

**Explanation:**

In the above program, the four binary bits are entered through the keyboard. If the number of one's entered is even, then the fifth bit is set to 0 otherwise to 1. This is checked by the `if` statement.

- 6.44 Write a program to convert a binary to decimal number. Enter the binary bits by using the for loop.

```
#include <process.h>
void main()
{
 int a,b,z[10],i;
 clrscr();
 printf("\nEnter the number of bits:-");
 scanf("%d",&b);
 printf("\nEnter the binary bits:-");
 for(i=0;i<b;i++)
 scanf("%d",&z[i]);
 a=z[0];
 for(i=0;i<(b-1);i++)
 {
 a=a*2+z[i+1];
 printf("\n%d",a);
 }
 printf("\nDecimal Number is : %d",a);
 getch();
}
```

**OUTPUT:**

```
Enter the number of bits:- 5
Enter the binary bits:- 1 0 0 0 1
2
4
8
17
Decimal Number is : 17
```

**Explanation:**

In the above program, the number of binary bits is entered. Using this number, individual bits are entered through the keyboard. For this for loop is used. The first element of the array is assigned to variable 'a'. The logic used here is to multiply by 2 to the variable 'a' and add the next successive bit. For this the equation  $a = a * 2 + z[i+1]$ ; is used. The number of iterations to be carried out is equal to the number of binary bits minus one. Here, five bits are initialized, hence the number of iterations will be four.

Table 6.5 gives the truth table of logic gates AND, OR and EX-OR.

**Table 6.5** Logic gates truth table of AND, OR and EX-OR

| Inputs |   | Outputs  |         |            |
|--------|---|----------|---------|------------|
|        |   | AND Gate | OR Gate | EX-OR Gate |
| A      | B | C        | C       | C          |
| 0      | 0 | 0        | 0       | 0          |
| 0      | 1 | 0        | 1       | 1          |
| 1      | 0 | 0        | 1       | 1          |
| 1      | 1 | 1        | 1       | 0          |



- 6.45 Write a program to verify the truth table of AND Gate. Assume AND Gate has two input bits A and B and one output bit C.

```
void main()
{
 int x,a[4],b[4],c[4];
 clrscr();
 printf("Enter Four Bits :");
 for(x=0;x<4;x++)
 {
 scanf("%d",&a[x]);
 if(a[x]>1 ||a[x]<0)
 {
 x--;
 continue;
 }
 }
 printf("Enter Four Bits :");
 for(x=0;x<4;x++)
 {
 scanf("%d",&b[x]);
 if(b[x]>1 ||b[x]<0)
 {
 x--;
 continue;
 }
 }
 printf("\nA B C");
 for(x=0;x<4;x++)
 {
 if(a[x]==1 && b[x]==1)
 c[x]=1;
 else
 c[x]=0;
 printf("\n%d %d %d",a[x],b[x],c[x]);
 }
}
```

**OUTPUT:**

```
Enter Four Bits: 1 0 1 0
Enter Four Bits: 1 0 0 1
A B C
1 1 1
0 0 0
1 0 0
0 1 0
```

**Explanation:**

The four binary bits for A and B are entered. The `if` statement checks the bits of A and B. If both of them are 1 then the output 'C' must be 1, otherwise for all other conditions the output is 0.

➤ 6.46 Write a program to verify the truth table of OR Gate.

```
void main()
{
 int x,a[4],b[4],c[4];
 clrscr();
 printf("Enter Four Bits :");
 for(x=0;x<4;x++)
 {
 scanf("%d",&a[x]);
 if(a[x]>1 || a[x]<0)
 {
 x--;
 continue;
 }
 }
 printf("Enter Four Bits :");
 for(x=0;x<4;x++)
 {
 scanf("%d",&b[x]);
 if(b[x]>1 || b[x]<0)
 {
 x--;
 continue;
 }
 }
 for(x=0;x<4;x++)
 {
 if(a[x]==0 && b[x]==0)
 c[x]=0;
 else
 c[x]=1;
 printf("\n%d %d %d",a[x],b[x],c[x]);
 }
}
```

#### OUTPUT:

```
Enter Four Bits: 1 1 1 0
Enter Four Bits: 1 0 0 0
A B C
1 1 1
1 0 1
1 0 1
0 0 0
```

#### *Explanation:*

The four binary bits for A and B are entered. The `if` statement checks the bits of A and B. If one of them is at logic 1, then the output 'C' must be 1. When the inputs are 0, the output is also 0.

- 6.47 Write a program to verify the truth table of EX-OR GATE.

```

void main()
{
 int x,a[4],b[4],c[4];
 clrscr();
 printf("Enter Four Bits :");
 for(x=0;x<4;x++)
 {
 scanf("%d",&a[x]);
 if(a[x]>1 ||a[x]<0)
 {
 x--;
 continue;
 }
 }
 printf("Enter Four Bits :");
 for(x=0;x<4;x++)
 {
 scanf("%d",&b[x]);
 if(b[x]>1 ||b[x]<0)
 {
 x--;
 continue;
 }
 }
 printf("\nA B C");
 for(x=0;x<4;x++)
 {
 if(a[x]==0 && b[x]==1)
 c[x]=1;
 else
 if(a[x]==1 && b[x]==0)
 c[x]=1;
 else
 c[x]=0;
 printf("\n%d %d %d",a[x],b[x],c[x]);
 }
}

```

#### OUTPUT:

```

Enter Four Bits: 1 1 1 0
Enter Four Bits: 1 0 0 0
A B C
1 1 0
1 0 1
1 0 1
0 0 0

```

**Explanation:**

The four binary bits for A and B are entered. The `if` statement checks the bits of A and B. If the input bits are dissimilar the output 'C' must be 1. When the inputs are similar, the output is also 0. In the two extreme cases, the output is zero.

- 6. 48 Write a program to find the Hamming code for the entered binary code. Assume the binary code of four bits in length. The Hamming code should be seven bits.

Tip: R.W. Hamming developed error correcting and detecting codes in communication the four bits data which is to be transmitted containing additional three check bits. The word that is to be transmitted, its format, will be D7, D6, D5, P4, D3, P2 and P1. Where D bits are data bits and the P bits are parity bits. P1 is set so that it provides even parity over bits P1, D3, D5 and D7. P2 is set for even parity over bits P2, D3, D6 and D7. P4 is set for even parity over bits P4, D5, D6 and D7

```
void main()
{
 static int c[7];
 int x,b[4];
 clrscr();
 printf("\n Read the Binary Numbers :");
 for(x=0;x<4;x++)
 scanf("%d",&b[x]);
 /* piece copy operation */
 c[0]=b[0];
 c[1]=b[1];
 c[2]=b[2];
 c[4]=b[3];
 printf("\n Before XOR operation :");
 for(x=0;x<=6;x++)
 printf("%3d",c[x]);

 c[6]= c[0]^c[2]^c[4];
 c[5]= c[0]^c[1]^c[4];
 c[3]= c[0]^c[1]^c[2];
 printf("\n Hamming code after XOR operation :");
 for(x=0;x<=6;x++)
 printf("%3d",c[x]);
 getch();
}
```

**OUTPUT:**

```
Read the Binary Numbers : 1 0 1 0
Before XOR operation : 1 0 1 0 0 0 0
Hamming code after XOR operation : 1 0 1 0 0 1 0
```

**Explanation:**

The hamming code that contains seven bits is initially assumed as zeros. This is declared with `static int c[7]`. The four bit data is read through the keyboard using the first `for` loop. The data bits are placed at the appropriate positions using piece copy operations. The three parity bits are evaluated using XOR bitwise operations. Ultimately, the seven bits are displayed using the last `for` loop.

- 6.49 Write a program to show the results of students who appeared in the final examination. Assume that the students have to appear in six subjects. The result declared should be as per the following table.

| <b>Total Marks</b> | <b>Result</b>   |
|--------------------|-----------------|
| >=420              | Distinction     |
| >=360              | First Division  |
| >=240              | Second Division |
| Otherwise          | Fail            |

```
#define DISTINCTION 420
#define FIRST 360
#define SECOND 240
void main()
{
 int number,i,j,roll_no,marks,total;
 clrscr();
 printf("\n Enter number of Students :");
 scanf("%d",&number);
 printf("\n");
 for(i=1;i<=number;++i)
 {
 printf("Enter Roll Number :");
 scanf("%d",&roll_no);
 total=0;
 printf("\n Enter Marks of 6 Subjects for Roll no %d :
 \n",roll_no);
 for(j=1;j<=6;j++)
 {
 scanf("%d",&marks);
 total=total+marks;
 }
 printf("TOTAL MARKS =%d",total);
 if(total>= DISTINCTION)
 printf("\n(Distinction)\n\n");
 else if(total>=FIRST)
 printf("\n(First Division)\n\n");
 else if(total>=SECOND)
 printf("\n(Second Division)\n\n");
 else
 printf("\n(***Fail***)");
 }
}
```

#### OUTPUT:

```
Enter number of Students : 1
Enter Roll Number : 1
Enter Marks of 6 Subjects for Roll no 1 :
42 52 62 72 82 92
TOTAL MARKS = 402
(First Division)
```

**Explanation:**

In the above program, the number of students whose result is to be calculated is entered. After this, the marks of six subjects with their roll numbers are entered. The sum of six subjects is calculated. The sum is compared with distinction, first and second macros using `if-else` ladder and appropriate result is displayed.

In this program, macros are used. During preprocessing, the preprocessors replace every occurrence of `distinction` with 420, similarly, `first` and `second` are replaced with 360 and 240, respectively. `Distinction`, `first` and `second` in the above program are 'macro templates', whereas 420, 360 and 240 are called their corresponding macro expansions. For details of macros refer to Chapter 12.

## 6.4 THE `while` LOOP

Another kind of loop structure in C is the `while` loop. The `while` loop is frequently used in programs for the repeated execution of statement/s in a loop. Until a certain condition is satisfied the loop statements are executed.

The syntax of `while` loop is

```
while(test condition)
{
 body of the loop
}
```

The test condition is indicated at the top and it tests the value of the expression before processing the body of the loop. The test condition may be any expression. The loop statements will be executed till the condition is true, i.e. the test condition is evaluated and if the condition is true, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.

The execution of the loop can be followed by the following flow chart given in Figure 6.5.

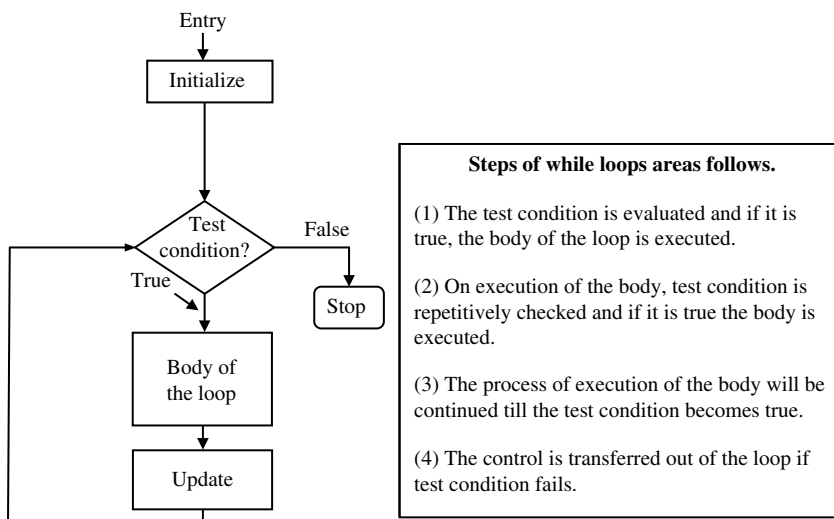


**Figure 6.5** The flow chart showing the execution of the loop

Here, the block of the loop may contain either a single statement or a number of statements. The same block can be repeated.

The braces are needed only if, the body of the loop contains more than one statement. However, it is good practice to use braces even if the body of the loop contains only one statement.

- 6.50 Write a program to print the string 'You have learnt C program' nine times using while loop.

```
void main()
{
 int x=1;
 while(x<10)
 {
 printf("\n You have learnt C program");
 x++;
 }
}
```

**OUTPUT:**

```
You have learnt C program
You have learnt C program
You have learnt C program
You have learnt C program
You have learnt C program
You have learnt C program
You have learnt C program
You have learnt C program
You have learnt C program
```

**Explanation:**

The parentheses after 'while' contains a condition. As long as the condition remains true, all the statements within the body of the loop get executed repeatedly. The variable 'x' is initialized to 1. The compiler checks the condition and after satisfying it, the body of the loop is executed. The control then goes to the while loop. Next time, the value of 'x' is incremented by 1. Now if 'x' is 2 again it satisfies the condition and the body of the loop gets executed. This process is continued till the value of 'x' reaches 9. When the condition becomes false (after 9), the control passes to the first statement that follows the body of the while loop and the program is now terminated. The output of the program is shown above.

- 6.51 Write a program to add 10 consecutive numbers starting from 1. Use the while loop.

```
void main()
{
 int a=1,sum=0;
 clrscr();
 while(a<=10)
 {
 printf("%3d",a);
 sum=sum+a;
 a++;
 }
 printf("\n Sum of 10 numbers :%d",sum);
}
```

**OUTPUT:**

```
1 2 3 4 5 6 7 8 9 10
Sum of 10 numbers : 55
```

**Explanation:**

In the above program, integer variable 'a' is initialized to 1 and variable 'sum' to 0. The while loop checks the condition for  $a \leq 10$ . The variable 'a' is added to variable 'sum' and each time 'a' is incremented by 1. In each while loop 'a' is incremented and added to 'sum'. When the value of 'a' reaches 10, the condition given in while loop is false. At last the loop is terminated. The sum of the number is displayed.

- 6.52 / 6.53 Write a program to calculate factorial of a given number. Use while loop.

```
void main()
{
 int a, fact=1;
 clrscr();
 printf("\n Enter The Number :");
 scanf("%d", &a);
 while(a>=1)
 {
 printf(" %d *", a);
 fact=fact*a;
 a--;
 }
 printf(" = %d", fact);
 printf("\n Factorial of Given number is %d", fact);
}
```

**OUTPUT:**

```
Enter The Number: 5
5* 4 * 3 * 2 * 1 * = 120
Factorial of Given number is 120
```

**Explanation:**

In the above program, the working of the while loop is the same as that in the previous one. The only difference is that variable 'a' is decremented. Factorial of a number means a product from 1 to that number. Here, variable 'fact' is initialized to 1. For each iteration of while loop the entered number is multiplied with the previous value of the variable 'fact' and 'a' is decremented. When the entered number 'a' reaches 1, the while loop terminates and 'fact' variable contains the product of 1 to the entered number. Here, the entered number is 5 and its factorial value is 120, and the same is displayed.

OR

```
void main()
{
 int a, b=1, fact=1;
 clrscr();
 printf("\n Enter The Number :");
```



```

scanf("%d", &a);
while(b<=a)
{
 printf(" %d *", b);
 fact=fact*b;
 b++;
}
printf(" = %d", fact);
printf("\n Factorial of %d is %d", b-1, fact);
}

```

**OUTPUT:**

```

Enter the Number: 4
1 * 2 * 3 * 4 * = 24
Factorial of 4 is 24.

```

**Explanation:**

The working of the above program is the same as the last one. Here, we declared and initialized one more variable 'b' to 1. Here, we are keeping the variable 'a' unchanged. We require an extra variable to count the number of loops completed by the `while` loop. Once the value of the variable 'b' matches with the entered number 'a', the `while` loop terminates. The factorial of a number in variable 'fact' is displayed.

- 6.54 Write a program to convert decimal number to binary number.

```

void main()
{
 int x,y=40;
 clrscr();
 printf("Enter a Number :");
 scanf("%d",&x);
 printf("\n Binary Number :");
 while(x!=0)
 {
 gotoxy(y--,3);
 printf("%d",x%2);
 x=x/2;
 }
}

```

**OUTPUT:**

```

Enter a Number : 25
Binary Number : 11001

```

**Explanation:**

In the above program, the number, which is to be converted into binary, is entered through the keyboard. The `while` loop is executed till the value of 'x' becomes non-zero. The body of the `while` loop contains the `mod (%)` and `divide` operations. With these operations, remainders are obtained and the value of 'x' is reduced to half. Binary bits corresponding to the decimal number which are obtained are taken in the reverse order by using the `gotoxy()` function. In the statement `gotoxy(y--, 3)`, the first argument (`y--`)

provides the column position and the second provides the row number where the output is to be displayed.

Similarly, for conversion of decimal to octal instead of 2 one can use 8 for mod as well as division operations.

- 6.55 Write a program to convert decimal number to the user-defined number system. The base of the number system may be taken up to 9.

```
void main()
{
 int m,b,y=35;
 clrscr();
 printf("Enter the Decimal Number :");
 scanf("%d",&m);
 printf("Enter base of number System :");
 scanf("%d",&b);
 printf("The Number Obtained :");
 while(m!=0)
 {
 gotoxy(y--,3);
 printf("%d",m%b);
 m=m/b;
 }
 getch();
}
```

**OUTPUT:**

```
Enter the Decimal Number : 50
Enter base of number System : 5
The Number Obtained : 200
```

**Explanation:**

The logic of the above program is the same as explained in the previous program. The only difference is that the user has a choice to enter the base of any number system. This program is more flexible than the previous one.

- 6.56 Write a program to convert binary number to equivalent decimal number.

```
include <math.h>
include <process.h>
void main()
{
 long n;
 int x,y=0,p=0;
 clrscr();
 printf("\n Enter a Binary Number :");
 scanf("%ld",&n);
 while(n!=0)
 {
```

```

 x=n%10;
 if(x>1 || x<0)
 {
 printf("\n Invalid Digit.");
 exit(1);
 }
 y=y+x*pow(2,p);
 n=n/10;
 p++;
 }
 printf("\n Equivalent Decimal Number is %d.",y);
 getch();
}

```

**OUTPUT:**

```

Enter a Binary Number : 1111
Equivalent Decimal Number is 15
Enter a Binary Number : 120
Invalid Digit.

```

**Explanation:**

In the above program, binary number is entered. The equation  $y = y + x \cdot \text{pow}(2, p)$  is used in each iteration where 'y' is initialized to 0. The variable 'x' is obtained by mod operation which separates individual digit of entered number, and the variable 'p' acts as an exponent for base 2. It is initially zero and incremented by one in each iteration, where p varies from 0 to the number of digits -1. The product ( $x \cdot \text{pow}(2, p)$ ) is added with the previous value of 'y'.

- 6.57 Write a program to read a positive integer number 'n' and generate the numbers in the following way. If entered number is 5, the output will be as follows. OUTPUT: 5 4 3 2 1 0 1 2 3 4 5.

```

#include <math.h>
void main()
{
 int n,i,k=0;
 clrscr();
 printf("Enter a number :");
 scanf("%d",&n);
 i=n+1;
 k=k-n;
 while(i!=k)
 {
 printf("%3d",abs(k));
 k++;
 }
}

```

**OUTPUT:**

```

Enter a number : 3
3 2 1 0 1 2 3

```

**Explanation:**

The value of ' $n$ ' is entered through the keyboard. The variable ' $k$ ' is initially declared zero. The equation  $k=k-n$  i.e.  $k=0-n=-n$ . The value of ' $k$ ' is negative, hence `abs()` function is used to print the positive (absolute) value. The `while` loop checks the condition for ( $i \neq k$ ). Variable ' $k$ ' is printed and incremented till its value becomes  $n+1$ . In this way, the numbers are printed as asked in the problem.

- 6.58 Write a program to enter a number through keyboard and find the sum of its digits.

```
void main()
{
 int n,k=1,sum=0;
 clrscr();
 printf("Enter a Number :");
 scanf("%d",&n);
 while(n!=0)
 {
 k=n%10;
 sum=sum+k;
 k=n/10;
 n=k;
 }
 printf("Sum of digits %d ",sum);
}
```

**OUTPUT:**

```
Enter a Number : 842
Sum of digits 14
```

**Explanation:**

In the above program, a number is entered by the user. It is assigned to variable ' $n$ '. In the `while` loop `mod` and `division` operations are done on the entered number. At each rotation of `while` loop, the remainder is obtained and added to the variable ' $sum$ '. The original number is also reduced by the division operation. Due to continuous division, at one stage the entered number reduces to zero. At this stage, the `while` loop condition mismatches and is terminated. The variable ' $sum$ ' displays the ' $sum$ ' of all digits of the entered number.

- 6.59 Write a program to enter few numbers and count the positive and negative numbers together with their sums. When 0 is entered program should be terminated.

```
void main()
{
 int j=1,p=0,n=0,s=0,ns=0;
 clrscr();
 printf("\n Enter Numbers (0) Exit :");
 while(j!=0)
 {
```

```

scanf("%d",&j);
if(j>0)
{
 p++;
 s=s+j;
}
else if(j<0)
{
 n++;
 ns=ns+j;
}
}
printf("\n Total Positive Numbers : %d",p);
printf("\n Total Negative Numbers : %d",n);
printf("\n Sum of Positive Numbers : %d",s);
printf("\n Sum of Negative Numbers : %d",ns);
}

```

**OUTPUT:**

```

Enter Numbers (0) Exit :1 2 3 4 5 -5 -4 -8 0
Total Positive Numbers : 5
Total Negative Numbers : 3
Sum of Positive Numbers : 15
Sum of Negative Numbers : -17

```

**Explanation:**

Numbers are entered through the keyboard. The `if` statement checks whether the entered numbers are greater or less than zero. If the numbers are greater than zero the `if` block is executed otherwise the `else` block is executed. The output shows the number of positive and negative numbers and their sums. When `0` is entered `while` loop terminates.

- 6.60 Read an integer through the keyboard. Sort odd and even numbers by using `while` loop. Write a program to add the sum of odd and even numbers separately and display the results.

```

void main()
{
 int a,c=1,odd=0,even=0;
 float b;
 clrscr();
 printf("Enter a Number :");
 scanf("%d", &a);
 printf("ODD EVEN");
 printf("\n");
 while(c<=a) /* A
 {
 b=c%2;
 while(b==0){/* B
 printf("\t%d ",c);
 even=even+c;
 b=1;}
 }
}

```

```

 b=c%2;
 while(b!=0){ /*c
 printf("\n%d", c);
 odd=odd+c;
 b=0;}
 c++;
 }
 printf("\n=====");
 printf("\n%d %d", odd, even);
}

```

**OUTPUT:**

Enter a Number: 10

| ODD   | EVEN |
|-------|------|
| 1     | 2    |
| 3     | 4    |
| 5     | 6    |
| 7     | 8    |
| 9     | 10   |
| ===== |      |
| 25    | 30   |

**Explanation:**

The above program is an example of nested **while** loops. When the entered number is even the first loop (B loop) is executed and number is added to variable '**even**', otherwise the **while** loop (C loop) is executed and the same job of addition is done with variable '**odd**'. Thus, variables '**odd**' and '**even**' give the sum of even and odd numbers.

- 6.61 Write a program to print the entered number in the reversed order.

```

void main()
{
 int n,d,x=1;
 int i;
 clrscr();
 printf("Enter the number of digits :-");
 scanf("%d",&d);
 printf("\nEnter the number which is to be reversed:-");
 scanf("%d",&n);
 printf("\nThe Reversed Number is :-");
 while(x<=d)
 {
 i=n%10;
 printf("%d",i);
 n=n/10;
 x++;
 }
 getch();
}

```

**OUTPUT:**

```
Enter the number of digits :- 4
Enter the number which is to be reversed:- 5428
The Reversed Number is :- 8245
```

**Explanation:**

The statements following the while loop are  $i=n\%10$  and  $n=n/10$ . They provide the remainder and quotient values, respectively. By taking repeatedly remainders and quotients, we get the number in the reverse order. For repeating the loop the 'x' is to be incremented.

- 6.62 Write a program to enter a statement entering a combination of capital, lower case, symbols and numerical. Carry out separation of capitals, lower case, symbols and numerical by using ASCII values from 48 to 122.

```
include <ctype.h>
void main()
{
 static char scan[40],cap[20],small[20],num[20],oth[20];
 int i=0,c=0,s=0,h=0,n=0;
 clrscr();
 puts("Enter Text Here :\n");
 gets(scan);
 while(scan[i]!='\0')
 {
 if(scan[i]>=48 && scan[i]<=57)
 num[++n]=scan[i];
 else
 if(scan[i]>=65 && scan[i]<=90)
 cap[++c]=scan[i];
 else
 if(scan[i]>=97 && (scan[i]<=122))
 small[++s]=scan[i];
 else
 oth[++h]=scan[i];
 i++;
 }
 printf("\nCapital Letters :[");
 for(i=0;i<20;i++)
 printf("%c",cap[i]);
 printf("\nSmall Letters :[");
 for(i=0;i<20;i++)
 printf("%c",small[i]);
 printf("\nNumaric Letters :[");
 for(i=0;i<20;i++)
 printf("%c",num[i]);
 printf("\nOther Letters :[");
 for(i=0;i<20;i++)
```

```

 printf("%c", oth[i]);
 printf("]");
 getch();
}

```

**OUTPUT:**

```

Enter Text Here: HAVE A NICE DAY, contact me on 51606.
Capital Letters: [HAVEANICEDAY]
Small Letters : [contactmeon]
Numaric Letters : [28669]
Other Letters: [,]

```

**Explanation:**

The array `scan[40]` contains the string. The string is to be entered by the user. The arrays `small[20]`, `num[20]` and `oth[20]` are used at run time in the program. The `while` loop reads the array `scan[]` character by character. The `if-else` ladder statements check the characters, whether the character is small, capital or symbol, and assigns it to the different arrays according to its case or type.

- 6.63 /6.64 Write a program to sort numbers 0 to 9, alphabets in upper and lower case using equivalent ASCII values. The following table can be used.

| ASCII values | Corresponding Symbols |
|--------------|-----------------------|
| 48 to 57     | 0 to 9                |
| 65 to 90     | A to Z                |
| 97 to 120    | a to z                |

```

void main()
{
 char a;
 int i=48;
 clrscr();
 printf("NUMBERS \n");
 while(i<=57)
 {
 printf("%c", i);
 i++;
 }
 i+=7;
 printf("\n\n CAPITAL ALPHABETS\n");
 while(i<=90)
 {
 printf(" %c", i);
 i++;
 }
 i+=6;
 printf("\n\n SMALL ALPHABETS \n");
 while(i<=122)
 {

```



```

 printf("%c",i);
 i++;
 }
}

```

**OUTPUT:**

NUMBERS

0 1 2 3 4 5 6 7 8 9

CAPITAL ALPHABETS

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SMALL ALPHABETS

a b c d e f g h i j k l m n o p q r s t u v w x y z

***Explanation:***

In the above program, three `while` loops are used. These three `while` loops are used for sorting numbers, capital letters and lowercase letters. The variable '`i`' is initialized to 48 because the ASCII 48 is '`0`'. The three `while` loops check ASCII values as per the ranges given in the table. The number, capital and small letters are not continuous in ASCII. Some symbols are also present between these values. To ignore these symbols, variable '`i`' is increased twice with 7 and 6.

OR

```

#include <process.h>
void main()
{
 int i=48;
 clrscr();
 printf("Numbers :");
 while(i<124)
 {
 if(i<58)
 printf("%2c",i);
 else
 {
 if(i==58)
 {
 printf("\nCapital Letters :");
 i+=7;
 }
 if(i>64 && i<91)
 printf("%2c",i);
 if(i==90)
 {
 printf("\n Small Letters :");
 i+=7;
 }
 if(i>96 && i<123)

```

```

 printf("%2c", i);
 }
 i++;
}
}

```

**Explanation:**

The output of the above program is the same as for the previous one. Instead of nested while loop, nested if-else statements are used.

- 6.65 Write a program to use three **while** nested loops. Print numbers after each iteration and messages after termination of each loop.

```

void main()
{
 int i=1,j=1,k=1;
 clrscr();
 while(i<4)
 {
 while(j<3)
 {
 while(k<2)
 {
 printf("\n\n i=%d j=%d k=%d",i,j,k);
 k++;
 }
 printf("\n Inner Loop (k) Completed.");
 k=1;
 j++;
 }
 printf("\n Middle Loop (j) Completed.");
 j=1;
 i++;
 }
 printf("\n Outer Loop (i) Completed.");
}

```

**OUTPUT:**

```

i=1 j=1 k=1
Inner Loop (k) Completed.
i=1 j=2 k=1
Inner Loop (k) Completed.
Middle Loop (j) Completed.
i=2 j=1 k=1
Inner Loop (k) Completed.
i=2 j=2 k=1
Inner Loop (k) Completed.
Middle Loop (j) Completed.
i=3 j=1 k=1
Inner Loop (k) Completed.
i=3 j=2 k=1

```

```

Inner Loop (k) Completed.
Middle Loop (j) Completed.
Outer Loop (i) Completed.

```

### **Explanation:**

In the above program, variables **i**, **j** and **k** are declared and initialized to 1. The inner most loop is 'k', the middle is 'j' and the outer most is 'i'. The execution of loop starts from outer to inner and the completion will be from inner most to the outer most. Here, for example, the values of **i**, **j** and **k** are printed and messages are printed to understand the termination of loops.

## 6.5 THE do-while LOOP

The syntax of do-while loop in C is as follows.

```

do
{
statement/s;
}
while(condition);

```

The difference between the **while** and **do-while** loop is the place where the condition is to be tested. In the **while** loops the condition is tested following the **while** statement, and then the body gets executed, whereas in the **do-while** the condition is checked at the end of the loop. The **do-while** loop will execute at least one time even if the condition is false initially. The **do-while** loop executes until the condition becomes false. The comparison between the **while** and **do-while** loop is given in Table 6.6

Some programs are given on **do-while** loop.

**Table 6.6** Comparison of the **while** and **do-while** loop

| Sr. No. | <b>while</b> Loop                                             | <b>do-while</b> Loop                                        |
|---------|---------------------------------------------------------------|-------------------------------------------------------------|
| 1       | Condition is specified at the top.                            | Condition is mentioned at the bottom.                       |
| 2       | Body statement/s is/are executed when condition is satisfied. | Body statement/s executes even when the condition is false. |
| 3       | No brackets for a single statement.                           | Brackets are essential even when a single statement exists. |
| 4       | It is an entry-controlled loop.                               | It is an exit-controlled loop.                              |

- 6.66 Use the **do-while** loop and display a message 'this is a program of **do-while** loop' five times.

```

void main()
{
 int i=1;
 clrscr();
 do
 {

```

```

 printf("\n This is a program of do while loop.");
 i++;
 }
 while(i<=5);
}

```

**OUTPUT:**

```

This is a program of do while loop.
This is a program of do while loop.
This is a program of do while loop.
This is a program of do while loop.
This is a program of do while loop.

```

**Explanation:**

The body of the loop is executed and the value of 'i' is incremented. The incremented value is tested with the condition specified at the outside of the loop. If the condition is true the statement within the loop gets executed. In the above program, the statement within the loop gets executed five times and the result is as shown above.

- 6.67 Write a program to print the entered number in the reversed order. Use do-while loop. Also perform sum and multiplication with their digits.

```

void main()
{
 int n,d,x=1,mul=1,sum=0;
 int i;
 clrscr();
 printf("Enter the number of digits :-");
 scanf("%d",&d);
 printf("\nEnter the number which is to be reversed:-");
 scanf("%d",&n);
 printf("\n Reversed Number :-");
 do
 {
 i=n%10;
 printf("%d",i);
 sum=sum+i;
 mul=mul*i;
 n=n/10;
 x++;
 }
 while(x<=d);
 printf("\n Addition of digits :- %4d",sum);
 printf("\n Multiplication of digits :- %4d", mul);
 getch();
}

```

**OUTPUT:**

```

Enter the number of digits : 4
Enter the number which is to be reversed:- 4321

```

Reversed Number :- 1234  
 Addition of digits :- 10  
 Multiplication of digits :- 24

**Explanation:**

In the above program, the length of the number and a number are entered. Using repetitive mod operations, digits are separated and displayed. The separated digits are repeatedly added and multiplied with variables 'sum' and 'mul', respectively. Initially 'sum' is 0 and 'mul' is 1. After the termination of loop 'sum' and 'mul' display addition and multiplication of individual digits of the entered number.

- 6.68 Write a program to find the cubes of 1 to 10 numbers using do-while loop.

```
/* This is a program of cube of a given number */
#include "math.h"
void main()
{
 int y,x=1;
 clrscr();
 printf("\n Print the numbers and their cubes");
 printf("\n");
 do
 {
 y=pow(x,3);
 printf("%4d %27d\n",x,y);
 x++;
 }
 while(x<=10);
}
```

**OUTPUT:**

Print the numbers and their cubes

|    |      |
|----|------|
| 1  | 1    |
| 2  | 8    |
| 3  | 27   |
| 4  | 64   |
| 5  | 125  |
| 6  | 216  |
| 7  | 343  |
| 8  | 512  |
| 9  | 729  |
| 10 | 1000 |

**Explanation:**

Here, the mathematical function pow (x, 3) is used. Its meaning is to calculate the third power of x. With this function, we get the value of  $y=x^3$ . For the use of the pow() function, we have to include math.h header file.

- 6.69 Write a program to check whether the given number is prime or not?

```
void main()
{
 int n,x=2;
 clrscr();
 printf("Enter The number for testing (prime or not) :");
 scanf("%d",&n);
 do
 {
 if(n%x==0)
 {
 printf("\n The number %d is not prime.",n);
 getch();
 exit(0);
 }
 x++;
 }
 while(x<n);
 printf("\n The number %d is prime.",n);
 getch();
}
```

**OUTPUT:**

```
Enter The number for testing (prime or not) : 5
The number 5 is prime.
```

**Explanation:**

The number is said to be a prime number, if it is not divisible by any number starting from 2 onwards up to  $n-1$ , where ' $n$ ' is the entered number by the user. Here, in this example, the entered number is 5. The mod operation is performed using the divisor from 2 to  $n-1$  (5-1).

If the remainder is 0 then it is not a prime number and controls exit from the program. Otherwise, if the remainder is non-zero then the number is prime.

- 6.70 Write a program to count the number of students having age less than 25 and weight less than 50 kg out of five.

```
void main()
{
 int age,count=0,x=1;
 float wt;
 clrscr();
 printf("\nEnter data of 5 boys\n");
 printf("\nAge Weight\n");
 do
 {
 scanf("%d %f", &age, &wt);
 if(age<25 && wt<50)
 {
```

```

 count++;
 }
 x++;
}
while(x<=5);
printf("\n Number of boys with age <25");
printf("and weight <50 Kgs =%d\n",count);
getch();
}

```

**OUTPUT:**

Enter data of 5 boys

Age Weight

24 51

20 45

25 51

20 35

24 54

Number of boys with age <25 and weight <50 kgs = 2

**Explanation:**

In the above-given program, age and weight of five boys are entered. The `if` condition checks the age and weight and after satisfying the condition counter variable `count` is increased by one. Thus, loop executes five times. The program displays the number of students having ages less than 25 years and weight less than 50 kg.

- 6.71 Compute the factorial of a given number using the `do-while` loop.

```

void main()
{
 int a,fact=1;
 clrscr();
 printf("\n Enter The Number :");
 scanf("%d", &a);
 do
 {
 printf(" %d *",a);
 fact=fact*a;
 a- -;
 }
 while(a>=1);
 printf(" = %d",fact);
 printf("\n Factorial of Given number is %d",fact);
}

```

**OUTPUT:**

Enter The Number : 5

5 \* 4 \* 3 \* 2 \* 1 \* = 120

Factorial of Given number is 120.

**Explanation:**

The logic of the program is self-explanatory.

- 6.72 Write a program to evaluate the series such as  $1+2+3+\dots+i$ . Use **do-while** loop. Where  $i$  can be a finite value. Its value should be read through the keyboard.

```
void main()
{
 int i,a=1;
 int s=0;
 clrscr();
 printf("\n Enter a number:");
 scanf("%d",&i);
 do
 {
 printf("%d +",a);
 s=s+a;
 a++;
 }
 while(a<=i);
 printf("\b\b s=%d",s);
}
```

**OUTPUT:**

```
Enter a number: 5
1 + 2 + 3 + 4 +5 s= 15
```

**Explanation:**

In the above program, variable ' $a$ ' is initially 1 and ' $s$ ' is 0. The value of variable ' $i$ ' is read which determines the final step of the series. In the **do-while** loop variable ' $a$ ' is printed, added to variable ' $s$ ' and then incremented. This loop is continuously executed till ' $a$ ' equals the entered number ' $i$ '. The output provides the summation of 1 to the entered number. Here, the entered number is 5. Hence, the sum from 1 to 5 is 15.

## 6.6 THE **while** LOOP WITHIN THE **do-while** LOOP

The syntax of **do-while** with multiple **while** statement loop is as follows:  
do-while(condition)

```
{
 statement/s;
}
while(condition);
```

- 6.73 Write a program to use **while** statement in **do-while** loop and print values from 1 to 5.

```
void main()
{
 int x=0;
 clrscr();
 do while(x<5)
 {
```



```

 x++;
 printf("\t %d", x);
 }
 while(x<1);
}

```

**OUTPUT:**

```
1 2 3 4 5
```

***Explanation:***

The specialty of using the second `while` loop in the above program is to know that the programmer can take the second `while` loop.

## 6.7 BOHM AND JACOPINI'S THEORY

The structured programming provides simplicity. Bohm and Jacopini showed the following three forms of controls:

- (i) sequence
- (ii) selection
- (iii) repetition

Sequence means statements appearing one after another. Recall that in the last chapter, we learned that selection can be implemented using `if-else` and `switch` statements. Now in this chapter, we learned that repetition can be implemented using `for`, `while` and `do-while` statements. `goto` instructions can be used by using `if` and `while`, `for` and `do-while`, i.e., C language includes `if` and other loop control structures which are equivalent to the same with `goto`.

All the above controls can be combined by two ways: (i) stacking and (ii) nesting. Thus, the structured programming supports simplicity.

### SUMMARY

This chapter deals with the loops that are to be used in the C programs. The scope of the various three loops such as (i) the `for` loop, (ii) the `while` loop and (iii) the `do-while` has been narrated in detail in this chapter. In this chapter the numbers of solved programs on these loops have been given for the benefit of the programmers. Also examples on nested loops have been given in a simple language and in depth. The information on breaking the loop and continuing the same is also elaborated in detail together with examples.

## EXERCISES

### I True or false:

1. A loop repeatedly executes a block of statements for certain number of times.
2. The loop `for ( ; ; )` is a non-working loop.
3. The `for ( ; ; )` loop with no arguments can be executed.
4. The loop `for (a=1; a<20; a++)` will be executed for 20 times.

5. The `while(1)` is an infinite loop.
6. The loop cannot be nested.
7. Even if the condition is false the `do-while` loop executes once.
8. The `do-while` loop must be terminated by a semi-colon.
9. The `{ }` defines the block of the statement.
10. In case `{ }` is not defined, the default scope is one statement

## II Match the following correct pairs given in Group A with Group B:

1

| Group A |                                      | Group B |                       |
|---------|--------------------------------------|---------|-----------------------|
| Sr. No. | Loop                                 | Sr. No. | No. of Execution      |
| 1       | <code>for ( ; ; )</code>             | A       | Loop executes 5 times |
| 2       | <code>for (j=1; j&lt;=5; j++)</code> | B       | Non-working loop      |
| 3       | <code>for ( ; ; ; )</code>           | C       | Infinite loop         |

2

| Group A |                                                             | Group B |                                    |
|---------|-------------------------------------------------------------|---------|------------------------------------|
| Sr. No. | Loop                                                        | Sr. No. | No. of Execution                   |
| 1       | <code>while(0)</code>                                       | A       | Wrong syntax of <code>while</code> |
| 2       | <code>while(1)</code>                                       | B       | Non-working loop                   |
| 3       | <code>while (a&gt;10);</code><br>(where <code>a=15</code> ) | C       | Infinite loop                      |

## III Select the appropriate option from the multiple choices given below:

1. What will be the last value of 'c' after the execution of following program?
 

```
void main()
{
 int c=1,d=0;
 clrscr();
 while(d<=9)
 {
 printf("\n %d %d",++d,++c);
 }
}
```

  - (a) 11
  - (b) 10
  - (c) 12
  - (d) 9
2. What will be the value of 'x' after the execution of following program?
 

```
void main()
{
 int k;
 float x=0;
 clrscr();
 for(k=0;k<10;k++)
 x+=.1;
 printf("\nx=%g",x);
}
```

  - (a) `x=1`
  - (b) `x=0`
  - (c) `x=1.1`
  - (d) None of the above
3. What will be the value of 'f' after the execution of following program?
 

```
void main()
{
 char k;
 float f=65;
 clrscr();
 for(k=1;k<=10;k++)
 {
 f-=.1;
 }
 printf("\nf=%g",f);
}
```

  - (a) `f=64`
  - (b) `f=-65`
  - (c) `f=66`
  - (d) None of the above

4. What would be the final value of 'x' after the execution of the following program?

```
void main()
{
 int x=1;
 clrscr();
 do while(x<=10)
 {
 x++;
 }
 while(x<=5);
 printf("\n x=%d", x);
}
```

- (a) x=11
- (b) x=6
- (c) x=2
- (d) None of the above

5. How many while statements are possible in the do-while loop?

- (a) 2
- (b) 1

- (c) 3
- (d) None of the above

6. What will be the final values of x and y?

```
void main()
{
 int x=1,y=1;
 clrscr();
 do while(x<=8)
 {
 x++,y++;
 }
 while(y<=5);
 printf("\n x=%d y=%d", x,y);
}
```

- (a) x=9 y=9
- (b) x=9 y=6
- (c) x=6 y=6
- (d) None of the above

#### IV Attempt the following programs:

- Write a program to display alphabets as given below.  
**Az by cx dw ev fu gt hs Ir jq kp lo mn nm ol pk qj ri sh tg uf ve wd xc yb za.**
- Write a program to display count values from 0 to 100 and flash each digit for one second. Reset the counter after it reaches to hundred. The procedure is to be repeated. Use **for** loop.
- Develop a program to simulate seconds in a clock. Put the 60 dots on the circle with equal distance between each other and mark them 0 to 59. A second's pointer is to be shown with any symbol. Also print the total number of revolution made by second's pointer.
- Write a program to simulate analog watch (1 to 12 numbers to be arranged in circular fashion with all the three pointers for seconds, minutes, and hours) on the screen. Use nested **for** loops.
  - (a) Use (.) dot for second's pointer.
  - (b) Use (\*) star for minute's pointer.
  - (c) Use (#) hash for hour's pointer.
- Write a program to calculate the sum of first and last number from 1 to 10.  
(Example 1+10, 2+9, 3+8 sums should be always 11.)
- Write a program to find the total number of votes in favour of persons 'A' and 'B'. Assume 100 voters will be casting their votes to these persons. Count the number of votes gained by 'A' and 'B'. User can enter his/her choices by pressing only 'A' or 'B'.
- Write a program to pass the resolution in a meeting comprising of five members. If three or more votes are obtained the resolution is passed otherwise rejected.
- Assume that there are 99 voters voting to a person for selecting chairman's candidature. If he secures more than 2/3 votes he should be declared as chairman otherwise his candidature will be rejected.
- Write a program to display the numbers of a series 1, 3, 9, 27, 81. . . n by using the **for** loop.
- Write a program to check that entered input data for the following. Whenever input is non-zero or positive display numbers from 1 to that number, otherwise display message 'negative or zero'. The program is to be performed for 10 numbers.
- Write a program to check entered data types for 10 times. If a character is entered print 'Character is entered' otherwise 'Numeric is entered' for numerical values.

12. Write a program to find the sum of the first hundred natural numbers.  $(1+2+3+\dots+100)$ .
13. Write a program to display numbers 11, 22, 33, ..., 99 using ASCII values from 48 to 57 in loops.
14. Create an infinite **for** loop. Check each value of the **for** loop. If the value is odd, display it otherwise continue with iterations. Print even numbers from 1 to 100. Use **break** statement to terminate the program.
15. Write a program to show the display as a rectangle of characters as shown below.
- ```

Z
YZY
XYZYX
RXYZYXR
XYZYX
YZY
Z

```
16. Write a program to read 10 numbers through the keyboard and count number of positive, negative and zero numbers.
17. Write a nested **for** loop that prints a 5×10 pattern of 0s.
18. Is it possible to create a loop using the **goto** statement? If yes write the code for it.
19. Write a program to find the triangular number of a given integer. For example triangular of 5 is $(1+2+3+4+5)$ 15. Use **do-while** loop.
20. Write a program to display all ASCII numbers and their equivalent characters numbers and symbols. Use **do-while** loop. User should prompt every time to press 'Y' or 'N'. If user presses 'Y' display next alphabet otherwise terminate the program.
21. Accept any five two numbers. If the first number is smaller than the second then display sum of their squares, otherwise sum of cubes.
22. Evaluate the following series. Use **do-while** loop.
- $1+3+5+7 \dots n$
 - $1+4+25+36 \dots n$
 - $x+x^2/2!+x^3/3!+\dots n$
 - $1+x+x^2+x^3+\dots x^n$
23. Write a program to display the following, using **do-while** loop.
- $a+1+b+2+c+3 \dots n$, where n is an integer.
 - $z+y+x \dots a$.
 - $za+yb+xc \dots az$.
24. Enter the 10 numbers through the keyboard and sort them in ascending and descending order, using **do-while** loop.
25. Enter text through the keyboard and display it in the reverse order. Use **do-while** loop.
26. Print multiplication of digits of any number. For example number 235, multiplication to be $5*3*2 = 30$. Use **do-while** loop.
27. Print square roots of each digit of any number. Consider each digit as perfect square. For example, for 494 the square roots to be printed should be 2 3 2.
28. Write a program to read a positive integer number 'n' and generate the numbers in the following way. If entered number is 3 the output will be as follows.
- 9 4 1 0 1 4 9
 - 9 4 1 0 1 2 3
29. Write a program to enter two integer values through the keyboard. Using **while** loop, perform the product of two integers. In case product is zero (0), loop should be terminated otherwise loop will continue.
30. Write a program to enter a single character either in lower or uppercase. Display its corresponding ASCII equivalent number. Use the **while** loop for checking ASCII equivalent numbers for different characters. When capital 'E' is pressed, the program should be terminated.
31. Write a program to read a positive integer number 'n' and generate the numbers in the following way. If entered number is 4 the output will be as follows. OUTPUT: 4! 3! 2! 1! 0! 1! 2! 3! 4! 5!.
32. Write a program to read a positive integer number 'n' and generate the numbers in the different ways as given below. If the entered number is 4 the output will be as follows.
- 2 4 6 8 10 ... n (provided n is even).
 - 1 3 5 7 9 ... n (provided n is odd).
33. Write a program to read a positive integer number 'n' and perform the squares of individual digits. For example $n=205$ then the output will be 25 0 4.

34. What would be the output of the given below programs?

- (a)
- ```
void main()
{
 while(0)
 {
 printf("\n Hello");
 }
}
```
- (b)
- ```
void main()
{
    while(!0)
    {
        printf("\n Hello");
    }
}
```
- (c)
- ```
void main()
{
 while(!1)
 {
 printf("\n Hello");
 }
}
```

- (d)
- ```
void main()
{
    while(! NULL)
    {
        printf("\n %s", "Hello");
    }
}
```
- (e)
- ```
void main()
{
 while(" ")
 {
 printf("\n %s", "Hello");
 }
}
```

35. What will be the output of the below given program? Attempt this program with other keywords and functions. List the names of keywords and functions, which can be used as arguments in the while loop.

```
void main()
{
 while(main)
 {
 printf("\n %d", main);
 }
}
```

## V Find the bug/s in the following program/s:

- 1.
- ```
void main()
{
    int n=1;
    clrscr();
    while(n<10);
    {
        printf("%d", n);
        n=n+3;
    }
    getch();
}
```
- 2.
- ```
void main()
{
 int n;
 clrscr();
 printf("Enter the number up to 8:-");
 scanf("%d", n);
 while(n<9)
 {
```

- ```
        printf("%d", );
        n=n+1;
    }
    getch();
}
```
- 3.
- ```
void main()
{
 int k=1;
 clrscr();
 do;
 {
 printf("\n %d", k);
 k=k+1;
 }
 while(k<=5);
 getch();
}
```
- 4.
- ```
void main()
{
```

```

    chr i=65;
    clrscr();
    do
    {
        printf(" %c",i)
        i=i+1;
    }
    while(i<=90);
    getche();
}

```

5.

```

void main()
{
    int num=678,sum=0,rev=0,digit;
    clrscr();
    do
    {
        digit=num-num\10*10;
        num=num/10;
        rev=rev*10+digit;
        sum=sum+digit;
    }
    while(num>0);
    printf("\n %d",rev);
    printf("\n");
    printf("\n %d",sum);
    getche();
}

```

6.

```

void main()
{
    int i,j=1,n=4;
    float sum=1.0;
    clrscr();

```

```

    for(i=1;i<=n;i++)
    {
        j=j*i;
        sum=sum+i/j;
    }
    printf("\n %.2f",sum);
    getche();
}

```

7.

```

void main()
{
    int i=10;
    clrscr();
    do
    {
        printf("%d",i);
        i=i-1;
    }
    while(i>0 || i=5);
    getche();
}

```

8.

```

void main()
{
    int x=7,y=0,z=7,i;
    clrscr();
    for(i=0;i<=x;i++);
    {
        y=x+i*z;
        printf("%d",y);
    }
    getche();
}

```

VI Answer the following questions:

1. What happens if you create a loop that never ends?
2. Is it possible to create a **for** loop that is never executed?
3. What is a loop? Why it is necessary in a program?
4. Is it possible to nest **while** loop within **for** loops?
5. How do you choose between **while** and **for** loops?
6. What is the difference between (!0) and (!1). How **while** loop works with these values?
7. What is the difference between (1= != 1) and (1!=1)?
8. What is the difference between (! 0) and (! 0)?
9. What are the values of **NULL** and **! NULL**?
10. Is it possible to use multiple **while** statements with **DO** statement?
11. Explain Bohm and Jacopini's theory.

ANSWERS

I True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	T	2.	F	3.	T	4.	F	5.	T
6.	F	7.	T	8.	T	9.	T	10.	T

II Match the following correct pairs given in Group A with Group B:

1.

Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	C	2.	A	3.	B

2.

Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	B	2.	C	3.	A

III Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	a	4.	a	5.	a
6.	a								

V Find the bug/s in the following program/s:

Q.	Ans.
1.	Semi-colon at the end of <code>while</code> loop is not expected program runs with this semi-colon, but nothing appears on screen on execution.
2.	<code>&</code> is missing in <code>scanf</code> and <code>n</code> variable in <code>printf</code> statements.
3.	Semi-colon is not expected in <code>do</code> .
4.	<code>chr</code> should replace with <code>char</code> & semi-colon is missing in <code>printf</code> .
5.	Instead of “\” division “/” is expected in <code>digit</code> statement.
6.	<code>(float)i/j</code> is expected in <code>sum</code> statement, if correct result is required otherwise the program gives runs but result given will be wrong.
7.	<code>i= =5</code> is expected.
8.	Omit semi-colon from <code>for</code> statement if correct result is required.

CHAPTER

7

Data Structure: Array

Chapter Outline

- 7.1 Introduction
- 7.2 Array Declaration
- 7.3 Array Initialization
- 7.4 Array Terminology
- 7.5 Characteristics of an Array
- 7.6 One-Dimensional Array
- 7.7 One-Dimensional Array and Operations
- 7.8 Operations with Arrays
- 7.9 Predefined Streams
- 7.10 Two-Dimensional Array and Operations
- 7.11 Three- or Multi-Dimensional Arrays
- 7.12 The **sscanf()** and **sprintf()** Functions
- 7.13 Drawbacks of Linear Arrays

7.1 INTRODUCTION

An array is a very popular and useful data structure used to store data elements in successive memory locations. More than one element is stored in a sequence, so it is also called a composite data structure. An array is a linear and homogeneous data structure. An array permits homogeneous data. It means that similar types of elements are stored contiguously in the memory and that too under one variable name. It can be combined with a non-homogeneous structure, and a complex data structure can be created. We know that an array of structure objects can also be useful. An array can be declared of any standard or custom data type. The array of character (strings) type works somewhat differently from an array of integers, floating numbers.

Consider the following example. A variable *a* having data-type integer is initially assigned some value and later on its value is changed. Later assigned value to the variable can be displayed.

```
void main()
{
    int a=2;
    a=4;
    printf("%d",a);
}
```

OUTPUT:

4

In the above example, the value of *a* printed is 4. 2 is assigned to '*a*' before assigning 4 to it. When we assign 4 to *a* then the value stored in '*a*' is replaced with the new value. Hence, ordinary variables are capable of storing one value at a time. This fact is the same for all the data types. But in numerous applications variables must be assigned more than one value. This can be obtained with the help of arrays. An array variable allows the storing of more similar data type elements/values at a time.

7.2 ARRAY DECLARATION

Declaration of a one-dimensional array can be done with data type first, followed by the variable name and lastly the array size is enclosed in square brackets. Array size should be integer constant and it must be greater than zero and data type should be valid C data type.

For example, declaration of one-dimensional array is as follows:

`int a[5];` It tells the compiler that '*a*' is an integer type of an array and its size is five integers. The compiler reserves 2 bytes of memory for each integer array element, i.e. 10 bytes are reserved for storing five integers in the memory. In the same way, an array of different data types is declared as follows:

```
char ch[10];
float real[10];
long num[5];
```

When we declare a variable, for example:

```
int x;
```

the variable *x* is declared and the memory location of two bytes is allocated to it and later a single value can be stored in it as shown in Figure 7.1

```
x=4;
```

Every variable has a name, a value assigned to it and it is to be stored in memory location. Hence, from the above, we can say that only one value is assigned to a variable or stored.

The way, we declare one-dimensional array in the same way and can also declare two-dimensional array. Two-dimensional array is a table that contains rows and columns. For Example, `int a[3][3]`; It informs the compiler that 'a' is an integer type of an array and its size is 9 integers. This array is a three-by-three matrix. Its details are given in the two-dimensional array and operations.

Similarly, one can declare the multi-dimensional array as follows:
`datatype arrayname[size1][size2][size3]-----[sizeN];`

For example, the following array declares three-dimensional array:
`int a[3][3][3];`

Its details are given in this chapter under three- or multi-dimensional array title.

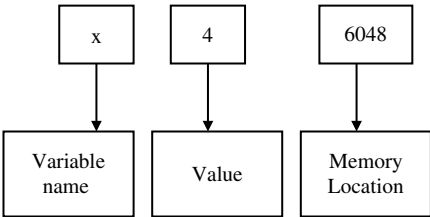


Figure 7.1 Variable mechanisms

7.3 ARRAY INITIALIZATION

The array initialization can be done as under:

```
int a[5]={1,2,3,4,5};
```

Here, five elements are stored in an array 'a'. List of elements initialized is shown within the braces. The array elements are stored sequentially in separate locations. Then, the question arises how to call individually to each element from this bunch of integer elements. Reading of the array elements begins from '0'. By indicating the position of elements, one can retrieve any element of an array. Array elements are called with array names followed by the element numbers. Table 7.1 explains the same.

Table 7.1 Calling array elements

<code>a[0]</code> refers to 1st element i.e. 1
<code>a[1]</code> refers to 2nd element i.e. 2
<code>a[2]</code> refers to 3rd element i.e. 3
<code>a[3]</code> refers to 4th element i.e. 4
<code>a[4]</code> refers to 5th element i.e. 5

Example:

To store more than one value the programming languages have an in-built data structure called an array.

1. `int num[5];`

In the above declaration, an integer array of five elements is declared. Memories for five integers, i.e. successive 10 bytes, are reserved for the num array. To initialize the num array following syntax can be used.

2. `int num[5] = {1,2,4,2,5};`

In the above statement, all elements are initialized. It is also possible to initialize individual element by specifying the subscript number in the square bracket following the array name. Array elements are accessed as follows:

```
num[0]=1;
num[1]=2;
num[2]=4;
```

```
num[3]=2;
num[4]=5;
```

The initialization can be done at the compile time or dynamically at the run time. The above is an example of compile time initialization. In the statement (2), declaration and initialization are done at once; in such type of declaration the number of elements (five) is not necessary to mention in the square bracket []. The compiler automatically counts the value initialized and assumes the number of elements initialized as the array size.

In the above array, the element `num[0]` i.e. 1 is the lowest bound and `num[4]` i.e. 5 is the last element. In C and C++, there is no bound checking. Hence, the programmer has to check it while accessing or storing elements. Once the array is declared, its lowest bound cannot be changed but the upper bound can be expanded. The array name itself is a constant pointer, and therefore we cannot modify it. Storing elements in contiguous memory locations can expand the upper bound.

The array name itself is a pointer. The array `num` is pointer to the first element i.e. `num` contains address of memory location where element 1 is stored. The address stored in the array name is called the base address. Figure 7.2 shows the pictorial representation. To access individual elements, the following syntax is used.

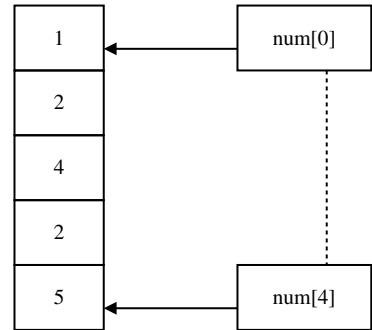


Figure 7.2 Array of integers

```
num[0] refers to the 1
num[1] refers to the 2
num[2] refers to the 4
num[3] refers to the 2
num[4] refers to the 5
```

Thus, an array is a collection of elements of the same data type, stored in unique and successive memory locations.

7.4 ARRAY TERMINOLOGY

Size: Number of elements or capacity to store elements in an array is called its size. It is always mentioned in brackets ([]).

Type: Types refer to data type. It decides which type of element is stored in the array. It also instructs the compiler to reserve memory according to data type.

Base: The address of the first element (0th) is a base address. The array name itself stores address of the first element.

Index: The array name is used to refer to the array element. For example, `num[x]`, `num` is array name and `x` is index. The value of `x` begins from 0 to onwards depending on the size of the array. The index value is always an integer value.

Range: Index of an array i.e. value of `x` varies from lower bound to upper bound while writing or reading elements from an array. For example, in `num[100]`, the range of index is 0 to 99.

Word: It indicates the space required for an element. In each memory location, computer can store a data piece. The space occupation varies from machine to machine. If the size of element is more than word (one byte) then it occupies two successive memory locations. The variables of data type `int`, `float`, `long` need more than one byte in memory.

7.5 CHARACTERISTICS OF AN ARRAY

1. The Declaration `int a[5]` is nothing but creation of five variables of integer types in memory. Instead of declaring five variables for five values, the programmer can define them in an array.
2. All the elements of an array share the same name, and they are distinguished from one another with the help of the element number.
3. The element number in an array plays a major role for calling each element.
4. Any particular element of an array can be modified separately without disturbing the other elements.

```
int a[5]={1,2,3,4,8};
```

If a programmer needs to replace 8 with 10, then it need not require changing all other numbers except 8. To carry out this task, the statement `a[4]=10` can be used. Here, other four elements are not disturbed.

5. Any element of an array `a[]` can be assigned/equated to another ordinary variable or array variable of its type.

Example:

```
b= a[2];
```

```
a[2]=a[3];
```

- (a) In the statement `b=a[2]` or vice versa, the value of `a[2]` is assigned to 'b', where 'b' is an integer.
 - (b) In the statement `a[2]=a[3]` or vice versa, the value of `a[2]` is assigned to `a[3]`, where both the elements are of the same array.
 - (c) The array elements are stored in continuous memory locations.
6. Array elements are stored in contiguous memory locations.

A program on array initialization and determination of their memory locations is given below:

- 7.1 Write a program to display array elements with their addresses.

```
int main()
{
    int num[5]={1,2,3,2,5};
    clrscr();
    printf("\n num[0] = %d Address : %u", num[0], &num[0]);
    printf("\n num[1] = %d Address : %u", num[1], &num[1]);
    printf("\n num[2] = %d Address : %u", num[2], &num[2]);
    printf("\n num[3] = %d Address : %u", num[3], &num[3]);
    printf("\n num[4] = %d Address : %u", num[4], &num[4]);
    return 0;
}
```

OUTPUT:

```
num[0] = 1 Address : 65516
num[1] = 2 Address : 65518
num[2] = 3 Address : 65520
num[3] = 2 Address : 65522
num[4] = 5 Address : 65524
```

Explanation:

In the output of the program, elements and their addresses are displayed. Recall that integer requires two bytes in memory. Hence, the memory locations displayed at the output have a difference of two. From the above program, it is clear that array elements are stored in contiguous memory locations. Figure 7.3 shows the memory location and values stored.

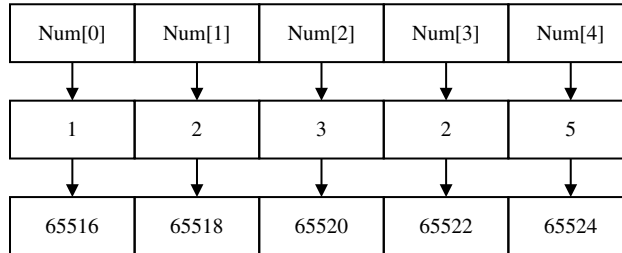


Figure 7.3 Storage of one-dimensional array

7. Once the array is declared, its lowest boundary cannot be changed but upper boundary can be expanded. The array name itself is a constant pointer and we cannot modify it. Therefore, the lowest boundary of an array cannot be expanded. In other words, even if the boundary exceeds than specified, nothing happens. The compiler throws no errors.

The reader can execute the following program for verifying the above concept.

- 7.2 Write a program to exceed the upper boundary of an array and see the element after expansion of an array.

```
void main()
{
    int num[5]={1,2,3,2,5};
    num[5]=6;
    clrscr();
    printf ("num[5]=%d", num[5]);
    getch();
}
```

OUTPUT:

Num[5]=6

Explanation:

In the above program, array `num[5]` is declared with array size 5 and it is initialized with 5 elements. In the next statement, 6th element is also initialized and displayed. Hence, we can say that upper boundary of an array can be expanded.

8. We know that an array name itself is a pointer. Though it is a pointer, it does not need `**` operator. The brackets `[ ]` automatically denote that the variable is a pointer.
9. All the elements of an array share the same name, and they are distinguished from one another with the help of the element number.

10. The amount of memory required for an array depends upon the data type and the number of elements. The total size in bytes for a single dimensional array is computed as shown below:

$$\text{Total bytes} = \text{sizeof}(\text{data type}) \times \text{size of array}$$

11. The operation such as insertion, deletion of an element can be done with the list but cannot be done with an array. Once an array is created, we cannot remove or insert memory location. An element can be deleted, replaced but the memory location remains as it is.
12. When an array is declared and not initialized, it contains garbage values. If we declared an array as **static**, all elements are initialized to zero. However, the values of static type data persist and remain in the memory as long as program executes. To overcome this problem, initialize first element of an array with zero or any number. Remaining all elements are automatically initialized to zero, provided the initialization is done in the declaration statement of an array. The following program illustrates this.

➤ 7.3 Write a program to initialize the static array and display its elements.

```
void main()
{
    int num[5]={0},j;
    clrscr();
    for(j=0;j<5;j++)
        printf("\nnum[%d]=%d",j,num[j]);
    getch();
}
```

OUTPUT:

```
num[0]=0
num[1]=0
num[2]=0
num[3]=0
num[4]=0
```

Explanation:

In the above program, an array `num[5]` is declared and the first element is initialized with zero. The compiler automatically initializes all elements with zero. Using the **for** loop the contents of an array are displayed and we can see that all elements have zero values.

7.6 ONE-DIMENSIONAL ARRAY

Array elements are stored contiguously in sequence one after the other. The elements of an array are just arranged in one-dimension. They can be shown in a row or column. Single subscript will be used in one-dimensional array to represent its elements.

An example of initialization of an array: - `int a[5];` in this initialization of an array is done. The type of variable is integer; its variable name is *a* and 5 is the size of the array.

The elements of the integer array `a[5]` are stored in contiguous memory locations. It is assumed that the starting memory location is 2000. Each integer element requires 2 bytes. Hence, subsequent element appears after the gap of two locations. Table 7.2 shows the locations of elements of integer array.

Table 7.2 *Integer data type and their memory locations*

Element	A[0]	A[1]	A[2]	A[3]	A[4]
Address	2000	2002	2004	2006	2008

Similarly, the elements of arrays of any data type are stored in contiguous memory location. The only difference is that the number of locations is different for different data types.

An example is illustrated below on the basis of this point.

- 7.4 Write a program to print bytes reserved for various types of data and space required for storing them in memory using arrays.

```
void main()
{
    int i[10];
    char c[10];
    long l[10];
    clrscr();
    printf("The type 'int' requires %d Bytes",sizeof(int));
    printf("\nThe type 'char' requires %d Bytes",sizeof(char));
    printf("\nThe type 'long' requires %d Bytes",sizeof(long));
    printf("\n %d memory locations are reserved for ten 'int'
elements",sizeof(i));
    printf("\n %d memory locations are reserved for ten 'char'
elements",sizeof(c));
    printf("\n %d memory locations are reserved for ten 'long'
elements",sizeof(l));
}
```

OUTPUT:

```
The type 'int' requires 2 Bytes
The type 'char' requires 1 Bytes
The type 'long' requires 4 Bytes
20 memory locations are reserved for ten 'int' elements
10 memory locations are reserved for ten 'char' elements
40 memory locations are reserved for ten 'long' elements
```

Explanation:

The `sizeof()` function provides the size of data type in bytes. In the above example, `int`, `char` and `long` type of data variables are supplied to this function which gives the results 2, 1 and 4 bytes, respectively. The required number of memory locations for `int`, `char` and `long` will be 2, 1 and 4. Memory locations required for the arrays = argument of an array $\times$ `sizeof(data type)`. In the above example, an array `int i[10]` requires 20 memory locations, since each element requires two memory locations. Memory requirement for various data types will be as given in Table 7.3.

Table 7.3 *Data type and their required bytes*

<i>Data Type</i>	<i>Memory Requirement</i>
char	1 bytes
int	2 bytes
float	4 bytes
long	4 bytes
double	8 bytes

Character arrays are called strings. There is a slight difference between an integer array and character array. In character array, NULL (`'\0'`) character is automatically added at the end, whereas in integer or other types of arrays, no null/character is placed at the end.

The NULL character acts as the end of the character array. By using this NULL character compiler detects the end of the character array. When compiler reads the NULL character `'\0'`, there is end of character array.

Note: Detailed information about strings (character array) is given in another chapter ‘*Strings and Standard functions.*’ The explanation about strings is given in brief in this chapter.

Given below is an example of a string.

➤ 7.5 Write a program to display character array with their address.

```
void main()
{
    char name[10]={'A','R','R','A','Y'};
    int i=0;
    clrscr();
    printf("\n Character Memory Location \n");
    while(name[i]!='\0')
    {
        printf("\n [%c]\t\t [%u]",name[i],&name[i]);
        i++;
    }
}
```

OUTPUT:

```
Character Memory Location
[A]          4054
[R]          4055
[R]          4056
[A]          4057
[Y]          4058
```

Explanation:

The elements of an array are stored in contiguous memory locations. In the above example, elements of one-dimensional array `'A','R','R','A','Y'` are stored from location 4054 to 4058.

One-dimensional character array elements will be stored in memory as per Table 7.4

Table 7.4 Character array elements and their locations

Array Element No	0	1	2	3	4
Elements	A	R	R	A	Y
Memory Addresses	4054	4055	4056	4057	4058

Notes: In case a NULL '`\0`' is initialized in the above example after 'Y', the result displayed will be 'ARRAY'.

A few programs are provided on one-dimensional array and they are as follows:

- 7.6 Write a program to add even and odd numbers from 1 to 10. Store them and display their results in two separate arrays.

```
void main()
{
    int sumo=0,sume=0,i=0,odd[5],even[5],a=-1,b=-1;
    clrscr();
    for (i=1;i<=10;i++)
    {
        if(i%2==0)
            even[++a]=i;
        else
            odd[++b]=i;
    }
    printf("\n\tEven \t\t\tOdd");
    for(i=0;i<5;i++)
    {
        printf("\n\t %d\t\t %d",even[i],odd[i]);
        sume=sume+even[i];
        sumo=sumo+odd[i];
    }
    printf("\n\t=====\\n");
    printf("Addition: %d %14d",sume,sumo);
}
```

OUTPUT:

```

          Even   Odd
           2     1
           4     3
           6     5
           8     7
          10     9
=====
Addition: 30    25
```

Explanation:

The `for` loop executes 10 times. In the `for` loop, the value of loop variable '`i`' is tested for 'even' and 'odd' conditions. If the value of '`i`' is even then it is assigned to array `even[]`

otherwise to `odd[]`. Thus, for 10 times this task is performed. Finally, the second `for` loop displays both the `even[]` and `odd[]` arrays. In the same array, sum of even and odd elements is calculated and displayed.

- 7.7 Write a program to input five numbers through the keyboard. Compute and display the addition of even numbers and product of odd numbers.

```
void main()
{
    int a=0,m=1,i,num[5];
    clrscr();
    for(i=0;i<5;i++)
    {
        printf("\nEnter Number[%d]:",i+1);
        scanf("%d",&num[i]);
    }
    printf("\n=====");
    for(i=0;i<5;i++)
    {
        if(num[i]%2==0)
        {
            printf("\n Even Number : %d",num[i]);
            a=a+num[i];
        }
        else
        {
            printf("\n Odd Number : %d",num[i]);
            m=m*num[i];
        }
    }
    printf("\n=====");
    printf("\n Addition of Even Numbers : %d",a);
    printf("\n Product of Odd Numbers :%d",m);
    printf("\n=====");
}
```

OUTPUT:

```
Enter Number[1]: 1
Enter Number[2]: 2
Enter Number[3]: 3
Enter Number[4]: 4
Enter Number[5]: 5
=====
Odd Number : 1
Even Number : 2
Odd Number : 3
Even Number : 4
Odd Number : 5
=====
Addition of Even Numbers : 6
Product of Odd Numbers : 15
=====
```

Explanation:

In the above example, five integers are entered through the keyboard. To detect even and odd numbers `mod(%)` operation is carried out and remainder of each number is obtained. If the remainder is 0, then the number is even and added to variable 'a'. If the remainder is non-zero then this number is multiplied to 'm'. Variables 'a' and 'm' are initialized with 0 and 1, respectively. Both the variables are printed through `printf()` function which gives addition of even numbers and product of odd numbers, respectively.

- 7.8 Write and display a program to detect the occurrence of a character in a given string.

```
void main()
{
    static char s[15];
    int i,c=0;
    char f;
    clrscr();
    puts("Enter a String :");
    gets(s);
    puts("Enter a Character to Find :");
    f=getchar();
    for(i=0;i<=15;i++)
    {
        if(s[i]==f)
            c++;
    }
    printf("The Character (%c) in a String (%s) occurs %d times.",f,s,c);
}
```

OUTPUT:

```
Enter a String : programmer
Enter a Character to Find : r
The Character (r) in a String (programmer) occurs (3) times.
```

Explanation:

In this program, the string and a single character are entered through the keyboard. Inside the `for` loop, the `if` statement checks each element of the string for the occurrence of the single entered character. If the character ('r') is found then 'c' counter is incremented otherwise without incrementing the counter loop continuous till 'i' reaches to 15. At last the value of 'c' gives the total occurrence of the given character.

- 7.9 Write a program to display the elements of two arrays in two separate columns and add their corresponding elements. Display the result of addition in the third column.

```
void main()
{
    int i,num[]={24,34,12,44,56,17};
    int num1[]={12,24,35,78,85,22};
    clrscr();
    printf("Element of Array 1st - 2nd Array Addition\n");
```

```

    for(i=0;i<=5;i++)
    {
        printf("\n\t\t %d + %d = \t%d",num[i],num1[i],num[i]+num1[i]);
    }
}

```

OUTPUT:

Element of Array 1st - 2nd Array Addition

```

    24 + 12 = 36
    34 + 24 = 58
    12 + 35 = 47
    44 + 78 = 122
    56 + 85 = 141
    17 + 22 = 39

```

Explanation:

In the above program, two integer arrays are initialized and the corresponding elements of arrays are added through a simple arithmetic operation.

- 7.10 Write a program to enter a character and integer data type. Use the two-dimensional array. Perform and display the addition of three numbers.

Tips: The Unsigned character data type is capable of performing mathematical operations on numbers from 1 to 255.

```

void main()
{
    static unsigned char l,r,i,real[3][5],ima[3][5];
    long c;
    clrscr();
    for(l=0;l<3;l++)
    {
        printf("Enter Number[%d] :",1+l);
        scanf("%ld",&c);
        r=c/255;
        i=c%255;
        real[l][5]=r;
        ima[l][5]=i;
    }
    c=0;
    for(l=0;l<3;l++)
        c=c+real[l][5]*255+ima[l][5];
    printf("\nSum of 3 Numbers :%3ld",c);
    getch();
}

```

OUTPUT:

```

Enter Number [1] : 5
Enter Number [2] : 4
Enter Number [3] : 3
Sum of 3 Numbers : 12

```

Explanation:

The unsigned character data type ranges from 0 to 255. In the above example, three numbers are entered. They are divided and the mod operation is carried out with 255. If the numbers are less than 255 neither division nor mod operations are carried out. Their sum is evaluated and displayed on the screen. In case the entered numbers are greater than 255, real and imaginary parts are computed and stored in the separate arrays. To obtain the whole number, the real part is multiplied with 255 and added to imaginary part.

- 7.11 Write a program to display names of days of a week using single-dimensional array having length of 7. (A week having seven days).

```
void main()
{
    int day[7],i;
    clrscr();
    printf("\nEnter numbers between 1 to 7 :\n");
    for(i=0;i<=6;i++)
        scanf("%d",&day[i]);
    for(i=0;i<=6;i++)
        switch(day[i])
        {
            case 1:
                printf("\n%dst day of week is Sunday",day[i]);
                break;
            case 2:
                printf("\n%dnd day of week is Monday",day[i]);
                break;
            case 3:
                printf("\n%drd day of week is Tuesday",day[i]);
                break;
            case 4:
                printf("\n%dth day of week is Wednesday",day[i]);
                break;
            case 5:
                printf("\n%dth day of week is Thursday",day[i]);
                break;
            case 6:
                printf("\n%dth day of week is Friday",day[i]);
                break;
            case 7:
                printf("\n%dth day of week is Saturday",day[i]);
                break;
            default :
                printf("\n %dth is Invalid day",day[i]);
        }
}
```

OUTPUT:

Enter numbers between 1 to 7 : 1 3 2 4 5 7 8

```

1st day of week is Sunday
3rd day of week is Tuesday
2nd day of week is Monday
4th day of week is Wednesday
5th day of week is Thursday
7th day of week is Saturday
8th is invalid day

```

Explanation:

In the above example, depending upon the value entered by the user, the `switch()` statement decides which day to print.

- 7.12 Write a program to display the contents of two arrays. The 1st array should contain the string and 2nd numerical numbers.

```

void main()
{
    char city[6]={'N','A','N','D','E','D'};
    int i,pin[6]={4,3,1,6,0,3};
    clrscr();
    for(i=0;i<6;i++)
        printf("%c",city[i]);
    printf("-");
    for(i=0;i<6;i++)
        printf("%d",pin[i]);
}

```

OUTPUT:

NANDED - 431603

Explanation:

In the above example, two arrays of different data types are printed through `printf()` function. The two `for` loops are used for printing two arrays containing the first string and second numerics.

- 7.13 Write a program to display the number of days of different months of year.

```

# include <process.h>
void main()
{
    int month[12]={31,28,31,30,31,30,31,31,30,31,30,31};
    int i;
    clrscr();

```

```

for(i=0;i<=11;i++)
{
    printf("\n Month [%d] of a year contains %d days.",i+1,month[i]);
    printf("\n");
}
getche();
}

```

OUTPUT:

```

Month [1] of a year contains 31 days.
Month [2] of a year contains 28 days.
Month [3] of a year contains 31 days.
Month [4] of a year contains 30 days.
Month [5] of a year contains 31 days.
Month [6] of a year contains 30 days.
Month [7] of a year contains 31 days.
Month [8] of a year contains 31 days.
Month [9] of a year contains 30 days.
Month [10] of a year contains 31 days.
Month [11] of a year contains 30 days.
Month [12] of a year contains 31 days.

```

Explanation:

In the above example, one-dimensional array `month[12]` is initialized with the number of days of different months of a year from 1 to 12 as per their order. In `printf()` function, number of days of the months are printed. The value of '*i*' is incremented by one for obtaining the increasing order of the month of a year.

- 7.14 Write a program to display the number of days of a given month of a year.

```

# include <process.h>
void main()
{
    int month[12]={1,3,5,7,8,10,12,4,6,9,11,2};
    int i,mn;
    clrscr();
    printf("Enter Number of Month :");
    scanf("%d",&mn);
    for(i=0;i<=11;i++)
    {
        if(mn==month[i])
            goto compare;
    }
    printf("\n Invalid Month");
    exit(1);
    compare;
}

```

```

    if(i+1==12)
        printf("Month (%d) Contains 28 days.",month[i]);
    if(i+1<8)
        printf("Month (%d) Contains 31 days.",month[i]);
    if(i+1>7 && i+1!=12)
        printf("Month (%d) Contains 30 days.",month[i]);
    getch();
}

```

OUTPUT:

```

Enter Number of Month : 2
Month (2) Contains 28 days.

```

Explanation:

This program is slightly different as compared to the last one. The numbers of days of different months of a year are sorted. For example, first seven (1,3,5,7,8,10,12) elements of an array have month numbers having 31 days, next four 30 days and last one 28 days. The user enters the month number and `if` statement checks where this month number appears in the array. Whenever there is a match control goes to the compare statements. The `if` statements print the number of days depending upon the conditions stated as above.

- 7.15 Write a program to find the average sales of an item out of 12 months sale.

```

void main()
{
    float sum=0,avg=0;
    int sale;
    int item[12];
    clrscr();
    printf("\tEnter Month No.-Sale of an Item/month\n");
    for(sale=0;sale<=11;sale++)
    {
        printf("\t\t %d =",sale+1);
        scanf("%d",&item[sale]);
    }
    for(sale=0;sale<=11;sale++)
        sum=sum+item[sale];
    avg=sum/12;
    printf("\n\t Average Sale of an item /month=%f",avg);
}

```

OUTPUT:

```

Enter Month No.-Sale of an Item/month
1 = 125
2 = 225
3 = 325
4 = 425
5 = 525
6 = 625

```



```

7 = 725
8 = 825
9 = 925
10 = 500
11 = 600
12 = 700
Average Sale of an item /month= 543.750000

```

Explanation:

In the above program, sales of 12 months are entered and stored in an array `item[12]`. By using the `for` loop the sum of sales of 12 months is calculated. Average of sales is then computed and the result is displayed.

- 7.16 Write a program to calculate and display the total cost of four models of Pentium PCs. Use the single-dimension arrays for PC codes, their price and quantity available.

```

void main()
{
    int i, pccode[4]={1,2,3,4};
    long t=0, price[4]={25000,30000,35000,40000};
    int stock[4]={25,20,15,20};
    clrscr();
    printf("\t Stock & Total Cost Details \n");
    printf("=====\n");
    printf("Model\t Qty.\tRate (Rs.) Total Value");
    printf("\n=====");
    for(i=0;i<=3;i++)
    {
        printf("\nPentium%d %d %8ld %15ld",pccode[i],stock[i],price[i],
            price[i]*stock[i]);
        t=t+price[i]*stock[i];
    }
    printf("\n=====\n");
    printf("Total Value of All PCs in Rs. %ld",t);
    printf("\n=====");
}

```

OUTPUT:

```

Stock & Total Cost Details
=====
Model Qty. Rate (Rs.) Total Value
=====
Pentium1 25 25000 625000
Pentium2 20 30000 600000
Pentium3 15 35000 525000
Pentium4 20 40000 800000
=====
Total Value of All PCs in Rs.2550000
=====

```

Explanation:

Here, in the above program, three integer arrays `pccode[]`, `price[]` and `stock[]` are initialized. After this the `for` loop is used for finding the total value of the available stock of each model. The result is printed by using simple multiplication of `price[]` and `stock[]`. This product is then added to variable '`t`', which is the total value. The `for` loop executes four times. At the end, variable '`t`' gives us the total cost of all PCs. Total cost is printed with `printf()` statement.

➤ 7.17 Write a program to display the given message by using `putc()` and `stdout()` functions.

```
void main(void)
{
    char msg[ ] = "C is Easy";
    int i = 0;
    clrscr();
    while(msg[i])
        putc(msg[i++], stdout);
}
```

OUTPUT:

C is Easy

Explanation:

A character array and integer variable are initialized. Array always begins with element number 0. In case if '`i`' is not initialized with 0 result provides garbage value. Standard `stdout()` function prints the string on console. Here, the string is '`C is Easy`'.

7.7 ONE-DIMENSIONAL ARRAY AND OPERATIONS

We learned how to declare, initialize and access the array elements. One-dimensional array elements may be shown in one row. So far, the examples, we discussed is of one-dimensional array.

Example:

```
int num[5]; // one dimensional array
```

One can perform numerous operations on elements of an array and the same are explained together with the following programs. We will learn how to traverse, insert, delete and display elements in the array during program execution.

Traversing: The operation of displaying or listing all elements of an array is called traversing. The following program explains traversing with one-dimensional array.

➤ 7.18 Write a program to read and display the elements of an array.

```
void main()
{
    int num[5],j;
    clrscr();
    printf("\n Enter five elements :");
    for(j=0;j<5;j++)
        scanf("%d",&num[j]);
    printf("\n Elements Address");
    for(j=0;j<5;j++)
        printf("\n%d %u",num[j],&num[j]);
    getch();
}
```

OUTPUT:

```
Enter five elements: 4 6 4 2 1
Elements Address
4 65516
6 65518
4 65520
2 65522
1 65524
```

Explanation:

In the above program, an array `num[]` is declared. The first `for` loop with the help of `scanf()` statement reads the elements and places in the array. The element position is indicated by the loop variable `j`. Same procedure is applied for displaying elements. The `printf()` statement displays the elements and addresses on the screen.

From Figure 7.4, one can see that one-dimensional arrays are stored one after another in sequence in the memory.

In an array, we can insert, delete or add any element but we cannot insert or delete the memory location. We can change only values.

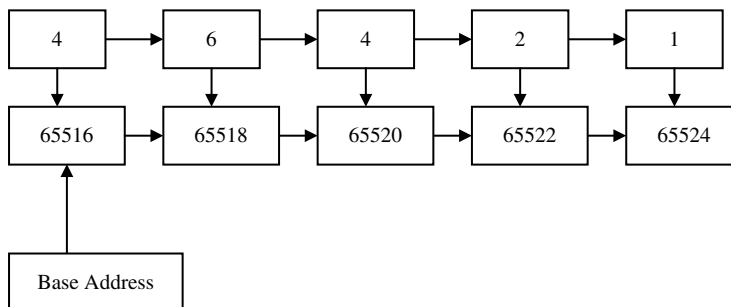


Figure 7.4 Array elements in memory

7.8 OPERATIONS WITH ARRAYS

Figure 7.5 shows frequently performed operations with arrays.

- (i) Deletion
 - (ii) Insertion
 - (iii) Searching
 - (iv) Merging
 - (v) Sorting
- (i) *Deletion*: This operation involves deleting specified elements from an array. Now consider the following programs.

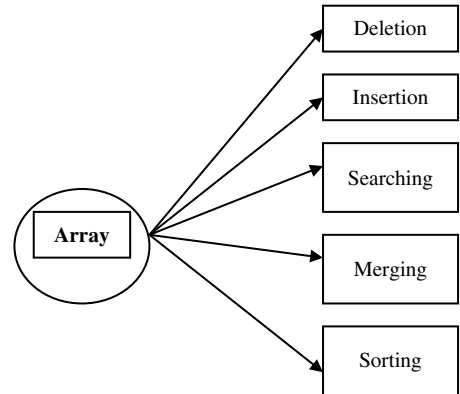


Figure 7.5 Operations with arrays

➤ 7.19 Write a program to delete specified element from an array and rearrange the elements.

```

void main()
{
    int num[20]={0},j,k,n,p,t;
    clrscr();
    printf("\n Enter number of elements :");
    scanf("%d",&n);
    printf("\n Enter elements :");
    for(j=0;j<n;j++)
        scanf("%d",&num[j]);
    printf("\n Elements are :");
    for(j=0;j<n;j++)
        printf("\n %d %u",num[j],&num[j]);
    printf("\n Enter element number to delete :");
    scanf("%d",&p);
    p--;
    for(j=0;j<n;j++)
    {
        if(j>=p)
            num[j]=num[j+1];
    }
    for(j=0;j<n;j++)
        if(num[j]!=0)
            printf("\n %d %u",num[j], &num[j]);
    getch();
}
  
```

OUTPUT:

```

Enter number of elements: 4
Enter elements: 5 4 1 2
  
```

```

Elements are:
5 65482
4 65484
1 65486
2 65488
Enter element number to delete: 3
5 65482
4 65484
2 65486

```

Explanation:

In the above program, an array `num[20]` is declared. The program asks for the number of elements to be entered. User has to enter the following input.

- (i) Number of elements to be entered and integers.
- (ii) Element number to be erased from an array.

The first `for` loop and `scanf()` statement reads numbers from keyboard and places in the array. In the second `for` loop onwards, the position of an element number is to be erased, the next array element is replaced with the previous one. Thus, the specified element is removed from an array. The third `for` loop and `printf()` statement display the elements of an array. You can see in the output that the third memory location is the same only if its contents are changed (see Figure 7.6 for view).

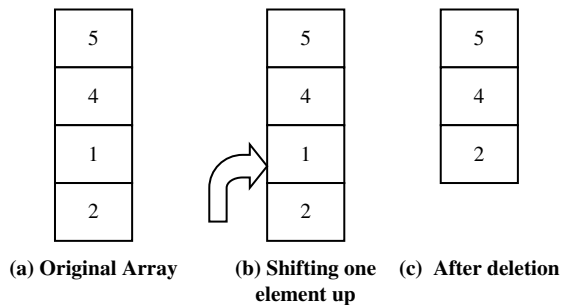


Figure 7.6 Deletion in steps

- (ii) *Insertion:* This operation is used to insert an element at a specified position in an array. Consider the following program.

➤ 7.20 Write a program to insert an element at a specified position in an array.

```

void main()
{
    int num[20]={0},j,k,n,p,t,s;
    clrscr();

```

```

printf("\n Enter number of elements :");
scanf("%d",&n);
printf("\n Enter elements :");
for(j=0;j<n;j++)
scanf("%d",&num[j]);
printf("\n Elements and their locations are :");
for(j=0;j<n;j++)
printf("\n%d %u",num[j],&num[j]);
printf("\n Enter element and position to insert at :");
scanf("%d %d",&s,&p);
p--;
for(j=n;j!=p;j--)
num[j]=num[j-1];
num[j]=s;
for(j=0;j<=n;j++)
printf("\n %d %u",num[j],&num[j]);
getche();
}

```

OUTPUT:

```

Enter number of elements : 4
Enter elements: 1
2
3
4
Elements and their locations are:
1 65450
2 65452
3 65454
4 65456
Enter element and position to insert at: 9 2
1 65450
9 65452
2 65454
3 65456
4 65458

```

Explanation:

This program is somewhat like previous program. Here, an element is inserted. The array elements are shifted to the next location and at a specified position and a space is created as shown in Figure 7.7 (b); the new element is inserted as shown in Figure 7.7 (c). Here, you can also see that though we inserted a new element, the memory location of the second element is the same (65452). Once again, it is proved that in array operation only contents of memory can change but the actual addresses remain as it is. The address of the first element, i.e. `num[0]` (65450), is called the base address. This address can also be stored in another pointer and array elements can be accessed. The next program is illustrated in this regard.

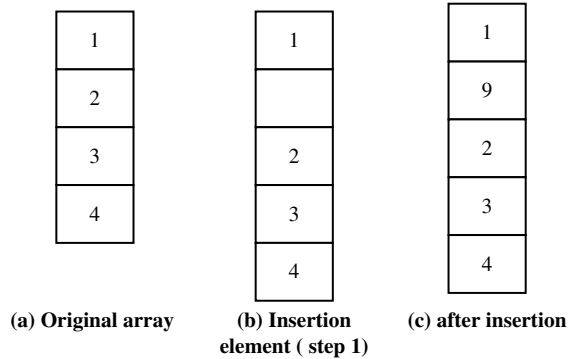


Figure 7.7 Insertion steps

➤ 7.21 Write a program to display one-dimensional array using integer pointer.

```
void main()
{
    int *p, num[5]={4,5,6,7,8}, j;
    clrscr();
    p=num;
    for(j=0; j<5; j++)
        printf("\n%d %u", *(p+j), (p+j));
    getch();
}
```

OUTPUT:

```
4 65486
5 65488
6 65490
7 65492
8 65494
```

Explanation:

In the above program, an integer array `num[]` is declared and initialized. In the same statement, pointer `p` and integer variable `j` are declared. The base address is assigned to pointer `p`. While assigning base address, it is enough to write the name of an array, and it is optional to write subscripts number, i.e. `num[0][0]`. The `for` loop executes five times and the value of `j` varies from 0 to 4. First time 0 is added to base address and there is no change in the address. Hence, 1st element is displayed. In the second iteration, one is added to the base address and it takes the next successive address and 2nd element is displayed. Same procedure is continued and array elements are displayed. Figure 7.8 simulates what exactly takes place.

In the above figure, the first line of boxes contains values, second contains memory addresses and the third contains loop variable values as used in the last program. When the value of a loop variable is added to the base address, we get the successive memory address and values stored in them can be displayed.

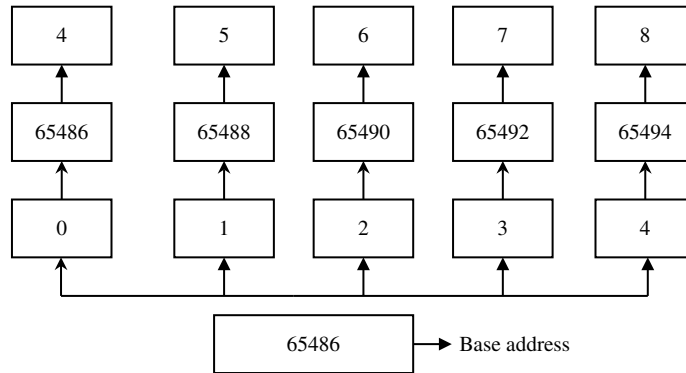


Figure 7.8 Accesses through base address

In this example, the base address is 65486 and the address gets incremented by 2 due to integers. For example, at following locations the data observed is as follows:

(65486)=4

(65488)=5

(65490)=6

(65492)=7

(65494)=8

(iii) *Searching*: An array element can be searched. The process of seeking specific elements in an array is called searching.

(iv) *Merging*: Merging of two arrays is an important operation with an array. The elements of two arrays are merged into a single one. The easier way of merging two arrays is first copy all elements of one array into third one and then copy all elements of the second array into the third one. We can also merge in alternate order as shown in the following program. Figure 7.9 indicates the merging of two arrays. One should take into account the following points:

- (i) Elements of one array can be appended to end of the second array.
- (ii) Elements of two arrays can be merged in alternate order.
- (iii) The size of resulting array must be more than the size of the two arrays.

➤ 7.22 Write a program to merge two arrays into third one. Display the contents of all the three arrays.

```
void main()
{
    int j,h=0,k=0;
    int x[4]={1,2,3,4};
    int y[4]={5,6,7,8};
    int z[8];
    clrscr();
    printf("\n Array 1:-");
    for(j=0;j<4;j++)
        printf("%d",x[j]);
```



```

printf("\n Array 2:-");
for(j=0;j<4;j++)
printf("%d",y[j]);
j=0;
while(j<8)
{
    if(j%2==0) z[j]=x[k++];
    else z[j]=y[h++];
    j++;
}
printf("\n Array 3:");
for(j=0;j<8;j++)
printf("%d",z[j]);
getche();
}

```

OUTPUT:

```

Array 1:- 1 2 3 4
Array 2:- 5 6 7 8
Array 3: 1 5 2 6 3 7 4 8

```

Explanation:

In the above program, three integer arrays are declared and initialized. The first two `for` loops are used to view the elements of arrays `X` and `Z` by transverse process. The `while` loop is used to execute until the condition is true. The `if()` statement checks the condition and accordingly `if` and `else` blocks are executed. These statements also fetch element from both arrays placed into array 3 (see Figure 7.9).

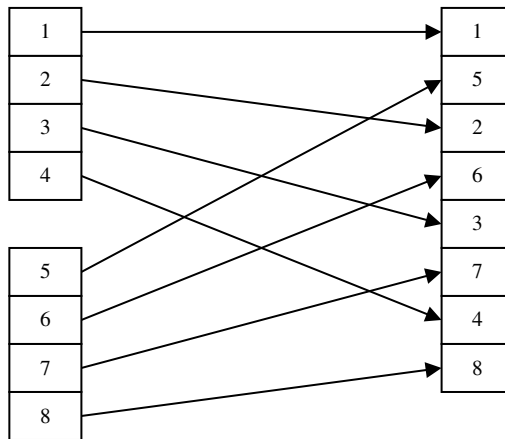


Figure 7.9 Merging two arrays

(v) *Sorting*: Arranging elements in a specific order either in ascending or in descending order is known as sorting. Sorting is a very important operation and compulsorily used in database application programs. Let us study the following program, which sorts an array of integers, and store them in another array.

➤ 7.23 Write a program to enter integer elements and sort them in ascending order.

```
void main()
{
    int num[5], j, k, s=0;
    clrscr();
    printf("\n Enter five Elements :");
    for(j=0; j<5; j++)
    {
        scanf("%d", &num[j]);
        s=s+num[j];
    }
    for(k=0; k<s; k++)
    {
        for(j=0; j<5; j++)
        {
            if(num[j]==k)
                printf("%d", num[j]);
        }
    }
}
```

OUTPUT:

```
Enter five Elements: 5 8 9 7 2
2 5 7 8 9
```

Explanation:

In the above program, an integer array is declared and five numbers are entered. The sum of all the numbers is taken. Using nested loop, every number of an array is compared from one to s (s =sum of all numbers). The `if` statement checks every array element with the value of s and displays number in the ascending order. The sorting can be done in various ways. The above is the simplest way, but consumes more time for sorting.

7.9 PREDEFINED STREAMS

Following are the stream functions:

- (i) `stdin()`
- (ii) `stdout()`
- (iii) `stderr()`

When C program is executed, a few 'files' are automatically opened by the system for use by the program. Constant FILE pointers recognize these files. Streams `stdin`, `stdout` and `stderr` are defined in the standard I/O include files. These are macros. Their details are illustrated in the chapter *Preprocessor Directives*.

- (i) `stdin`:

The file pointer `stdin` identifies the standard input text and it is associated with the terminal. All standard I/O functions perform input and do not take a FILE pointer as an argument and get their input from `stdin`.

- (ii) **stdout**: Similarly, it is used for outputting the text. It is a standard output stream.
 - (iii) **stderr**: It is an output stream in the text mode. It is a standard error stream.
- 7.24 Write a program to read the text through keyboard and display it by using **stdin** and **stdout** streams.

```
void main(void)
{
    char ch[11];
    int i;
    clrscr();
    printf("Input a Text:");
    for(i=0;i<10;i++)
        ch[i]=getc(stdin);
    printf("The Text inputted was");
    for(i=0;i<10;i++)
        putc(ch[i],stdout);
}
```

OUTPUT:

```
Input a Text: Hello World
The text inputted was Hello World
```

Explanation:

In the above program, two **for** loops are used. The first **for** loop reads input characters from the keyboard and stores in an array **ch[11]** by using **stdin** standard function. The second **for** loop displays the string using **stdout** standard function.

- 7.25 Write a program to sort the given strings alphabetically.

```
# include <ctype.h>
void main()
{
    int i,j;
    char text[30];
    clrscr();
    printf("Enter Text Below :");
    gets(text);
    clrscr();
    printf("Sorted Text Below :\n");
    for(i=65;i<=90;i++)
    {
        for(j=0;j<30;j++)
        {
            if( text[j]==toupper(i) || text[j]==tolower(i))
                printf("%c",text[j]);
        }
    }
}
```

OUTPUT:

```
Enter Text Below :
Hello
Sorted Text Below :
Ehllo
```

Tips: The `tolower` and `toupper` are the 'C' functions for conversion from lower to upper or vice versa or convert numerical to its ASCII equivalent. For initializing these functions header file `ctype.h` is to be included

Explanation:

ASCII equivalents of alphabets are used to sort the given string. The standard functions `toupper()` and `tolower()` in the `if` statement are used to ignore the case; i.e. capitals or small letters are treated the same. If character value given by `for` loop '`i`' and string `text[]` value denoted by `for` loop '`j`' are the same that value gets printed, because the value of character supplied by the outer `for` loop is taken alphabetically. Hence, characters when matched get printed.

- 7.26 Write a program to arrange the numbers in increasing and decreasing order (ascending and descending order) using loops.

```
void main()
{
    int i,j,sum=0,a[10];
    clrscr();
    printf("Enter ten numbers :");
    for(i=0;i<10;i++)
    {
        scanf("%d",&a[i]);
        sum=sum+a[i];
    }
    printf("Numbers In Ascending Order :");
    for(i=0;i<=sum;i++)
    {
        for(j=0;j<10;j++)
        {
            if(i==a[j])
                printf("%3d",a[j]);
        }
    }
    printf("\nNumbers In Descending Order :");
    for(i=sum;i>=0;i--)
    {
        for(j=0;j<10;j++)
        {
            if(i==a[j])
                printf("%3d",a[j]);
        }
    }
}
```

OUTPUT:

Enter ten numbers : 5 2 1 4 7 9 10 12 9 3

Numbers In Ascending Order : 1 2 3 4 5 7 9 9 10 12

Numbers In Descending Order: 12 10 9 9 7 5 4 3 2 1

Explanation:

In the above program, 10 numbers are entered through the keyboard in an array `a[10]`. In the same loop, their sum is performed. The outer loop executes from 0 to variable 'sum'. Here, 'sum' variable contains addition of all the 10 entered numbers. The inner loop checks all the values of an array `a[10]` with the current value of outer loop. The `if` statement checks the value of outer loop with the entire array, when it finds match number gets printed. The outer loop continues till it reaches the value of variable 'sum'. The outer loop is in ascending order. So the elements of an array `a[10]` are printed in the ascending order. For descending order, the outer `for` loop is made descending.

OR

➤ 7.27 Write a program to sort the numbers in ascending order by comparison method.

```
void main()
{
    int i,j,n,num[10],temp;
    clrscr();
    printf("Enter how many numbers to sort :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter numbers # %2d:",i+1);
        scanf("%d",&num[i]);
    }
    printf("The Numbers Entered through keyboard \n");
    for(i=0;i<n;i++)
    printf("%4d",num[i]);
    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(num[i]>num[j])
            {
                temp=num[i];
                num[i]=num[j];
                num[j]=temp;
            }
        }
    }
    printf("\n The Numbers in ascending order \n");
    for(i=0;i<n;i++)
    printf("%4d",num[i]);
    getch();
}
```

OUTPUT:

```

Enter how many numbers to sort : 5
Enter numbers # 1: 8
Enter numbers # 2: 3
Enter numbers # 3: 2
Enter numbers # 4: 1
Enter numbers # 5: 9

The Numbers Entered through keyboard
8 3 2 1 9
The Numbers in ascending order
1 2 3 8 9

```

Explanation:

Here, sorting of numbers is made through the exchange method. The element number given by the outer `for` loop of an array `num [10]` is compared with all the other elements of the same array. If the first element is larger than the successive element, the larger value is assigned to variable '`temp`' (`temp=num[i];`) and in place of larger value, smaller value is replaced (`num [i]=num[j];`). In place of smaller value, the value of '`temp`' variable (which is larger) is replaced (`num[j]=temp;`). Thus, the array elements in ascending order are obtained using the above program.

- 7.28 Write a program to evaluate the following series. The series contains the sum of square of numbers from 1 to 'n'. Store result of each term in an array. Calculate the value of 's' using an array
/* s= 1²+2²+3²+4² ...n² */.

```

# include <math.h>
void main()
{
    static int sqr[15];
    int i,n,s=0;
    clrscr();
    printf("Enter value of n:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
        sqr[i]=pow(i+1,2);
    for(i=0;i<n;i++)
    {
        printf("%d\n",sqr[i]);
        s=s+sqr[i];
    }
    printf("Sum of square : %d",s);
}

```

OUTPUT:

```

Enter value of n: 4
1
4
9
16
Sum of square : 30

```

Explanation:

The above program evaluates squares up to 'n' numbers. The value of 'n' is entered through the keyboard. Square of each number is stored in an array `sqr[15]` and their sum is calculated. Final result is displayed.

7.10 TWO-DIMENSIONAL ARRAY AND OPERATIONS

Two-dimensional arrays can be thought of rectangular display of elements with rows and columns. Consider the following example `int x[3][3]`; The two-dimensional array can be declared as in Figure 7.10.

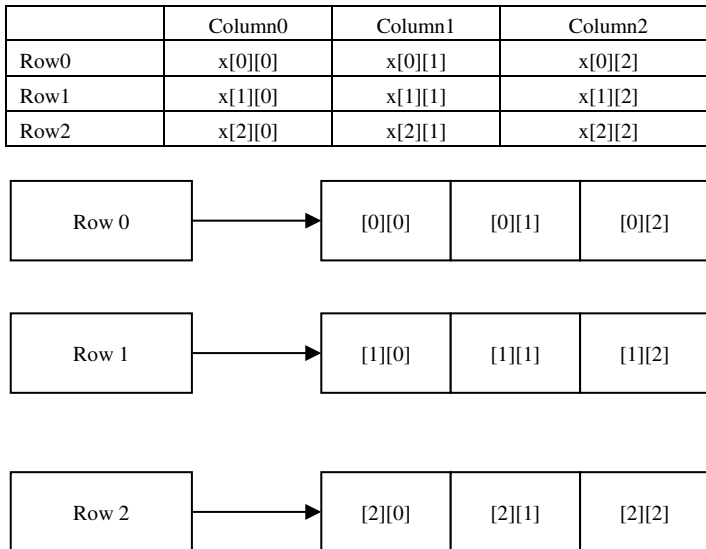


Figure 7.10 Two-dimensional array

The arrangement of array elements shown in Figure 7.10 is only for the sake of understanding. Actually, the elements are stored in the contiguous memory locations. The two-dimensional array is a collection of two one-dimensional arrays. The meaning of the first argument is in `x[3][3]` means number of rows, i.e. the number of one-dimensional array and the second argument indicate the number of elements. The `x[0][0]` means the first element of the first row and column. In one row, the row number remains the same and the column number changes. The number of rows and columns is called the range of an array. A two-dimensional array clearly shows the difference between logical assumption and physical representation of the data. The computer memory is linear and any type of an array may be one, two or multi-dimensional, it is stored in the continuous memory location (Figure 7.11).

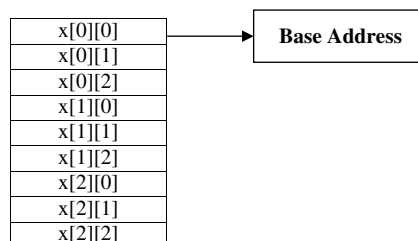


Figure 7.11 Storage of two-dimensional array

- 7.29 Write a program to demonstrate the use of two-dimensional array.

```
void main()
{
    int i,j;
    int a[3][3]={1,2,3,4,5,6,7,8,9};
    clrscr();
    printf("\n Array elements and address");
    printf("\n\t Col-0 Col-1 Col-2");
    printf("\n\t =====");
    printf("\nRow0");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
            printf("%d [%u]",a[i][j],&a[i][j]);
        printf("\nRow%d",i+1);
    }
    printf("\r");
}
```

OUTPUT:

```
Array elements and address
      Col-0      Col-1      Col-2
      =====
Row0 1 [65508] 2 [65510] 3 [65512]
Row1 4 [65514] 5 [65516] 6 [65518]
Row2 7 [65520] 8 [65522] 9 [65524]
```

Explanation:

From the above program's output, you can see that the memory address displayed is in sequence and it is true that the elements of two-dimensional array are stored in successive memory locations. The one-dimensional array can be accessed using a single loop. However, for two-dimensional array two loops are required for row and column. The inner loop helps to access the rowwise elements and outer loop changes the row number. Like, one-dimensional array base address of an array can be stored in pointer. Consider the following program:

- 7.30 Write a program to assign base address of two-dimensional array to pointer and display the elements.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int *p,num[2][2]={4,5,6,7},j;
    clrscr();
    p=&num[0][0];
    for(j=0;j<4;j++)
```



```

    printf(" %d %u\n", *(p+j), unsigned(p+j));
    getch();
}

```

OUTPUT:

```

4 65518
5 65520
6 65522
7 65524

```

Explanation:

The above program is the same as the last one. But the point to note here is that to store base address of two-dimensional array, it is not only enough to mention array, in addition subscript number and address operation also should be preceded. Then compiler accepts the statement; otherwise the compiler flags an error message. For one-dimensional array an array name is sufficient but onwards we have to mention element number with address operator.

7.10.1 | Insert Operation with Two-Dimensional Array

We have studied the example of insertion of element with one-dimensional array. We are also aware of the fact how pointers can be used to access array elements. The following program gives you an idea of insert operation with two-dimensional array.

➤ 7.31 Write a program to illustrate insert operation with two-dimensional array.

```

void main()
{
    int num[5][5]={0,0},j,*p,at,e,n;
    clrscr();
    printf("\n Enter how many elements (<=25) :");
    scanf("%d",&n);
    p=&num[0][0];
    for(j=0;j<n;j++)
        scanf("%d",p++);
    p=&num[0][0]; printf ("\n");
    for(j=0;j<n;j++,p++)
        printf("%2d",*p);
    printf("\n Enter a number & position to insert :");
    scanf("%d %d",&e,&at);
    for(j=n;j>=at;j--)
    {
        *p=*(p-1);
        p--;
    }
}

```

```

    *p=e;
    p=&num[0][0];
    for(j=0;j<=n;j++,p++)
        printf("%2d", *p);
}

```

OUTPUT:

```

Enter how many elements (<=25): 5
1 2 3 4 5
1 2 3 4 5
Enter a number & position to insert: 8 3
1 2 8 3 4 5

```

Explanation:

In all types of array, one-, two- and three-dimensional elements are stored in successive memory locations. A pointer is used to get successive memory location of memory. In this way, elements of an array can be accessed, changed or replaced. For all this, we require only the base address of the array that is to be assigned to pointer.

Consider the statement `p=&num [0][0];` used to store the base address (address of 0th element of the array) of an array. Later in the program, we need not access the array by its name and corresponding row, column numbers. The total capacity of array of storing element is 25 as per the declaration. The user is asked to enter the number of elements. The entered value is stored in variable *n* through the `scanf ( )` statement. Thus, using loops, values are repetitively entered and displayed.

The user is again asked to enter a number and position in the array where the element is to be inserted. These values are stored in variables (`scanf ( "%d %d", &e, &at );`) variables *e* and *at*.

```

for(j=n;j>=at;j--)
{
    *p=*(p-1);
    p--;
}

```

Using the `for` loop the elements are shifted to the next position up to the value of variable '*at*'.

When loop ends, the entered element is assigned to `*(p=e;)`. The '*e*' element is already shifted to the next position. Thus, finally the list of the latest elements is displayed. Figures 7.12 (a) and (b) show the representation of the insertion of elements.

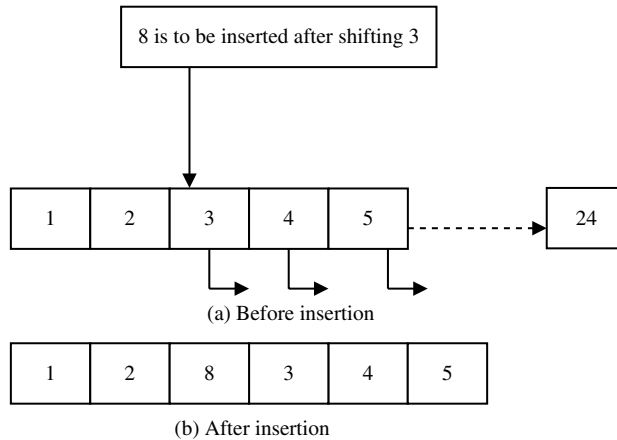


Figure 7.12 Insertion of element

7.10.2 | Delete Operation with Two-Dimensional Array

Like insert operation delete operation can also be performed on an array. Consider the following program:

➤ 7.32 Write a program to demonstrate delete operation of element with two-dimensional array.

```
void main()
{
    int num[5][5]={0,0},j,*p,at,e,n;
    clrscr();
    printf("\n Enter how many elements (<=25) :");
    scanf("%d",&n);
    p=&num[0][0];
    for(j=0;j<n;j++)
        scanf("%d",p++);
    p=&num[0][0]; printf ("\n");
    for(j=0;j<n;j++,p++)
        printf("%d",*p);
    printf("\n Enter Element number to delete :");
    scanf("%d",&at);
    at--;
    p=&num[0][0];
    p+=at;
    for(j=at;j<n;j++)
    {
        *p=*(p+1);
        p++;
    }
    *p=0;
    p=&num[0][0];
}
```

```

while(*p!=0)
{
    printf("%d", *p);
    p++;
}

```

OUTPUT:

```

Enter how many elements (<=25) : 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
Enter Element number to delete: 5
1 2 3 4 6 7

```

Explanation:

The method obtaining base address and pointer arithmetic involved in the program is the same as the last program. The element to be deleted is replaced with the next element. Thus, the entire elements are shifted to previous location. Figures 7.13(a) and (b) describe the deletion of element.

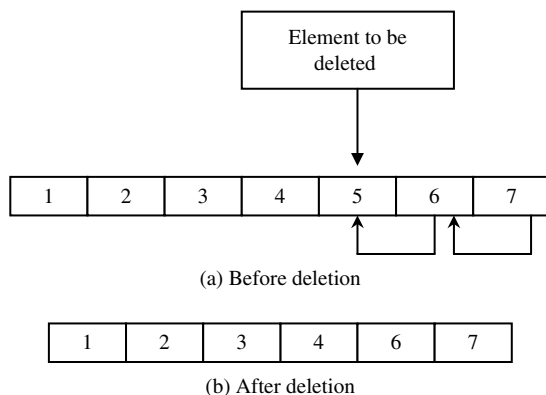


Figure 7.13 Deletion of element

Two-dimensional array can be thought as a rectangular display of elements with rows and columns. For example, elements of `int x[3][3]` are shown in Table 7.5.

Table 7.5 Array elements in matrix form

	Col0	Col1	Col2
Row1	X[0][0]	x[0][1]	x[0][2]
Row2	x[1][0]	x[1][1]	x[1][2]
Row3	x[2][0]	x[2][1]	x[2][2]

The arrangement of array elements in Table 7.5 is only for the sake of understanding. Conceptually, the elements are shown in matrix form. Physically array elements are stored in one contiguous form in memory.

The two-dimensional array is a collection of a number of one-dimensional arrays, which are placed one after another. For example, in Table 7.5 each row of a two-dimensional array can be thought of as a single-dimensional array.

➤ 7.33 Write a program to display two-dimensional array elements together with their addresses.

```
void main()
{
    int i,j;
    int a[3][3]={1,2,3},{4,5,6},{7,8,9}};
    clrscr();
    printf("Array Elements and addresses.\n\n");
    printf("Col-0 Col-1 Col-2\n");
    printf("===== \n");
    printf("row0");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
            printf("%d [%5d]", a[i][j], &a[i][j]);
        printf("\nRow%d", i+1);
    }
    printf("\r");
}
```

OUTPUT:

```

      Array Elements and addresses.
      Col-0      Col-1      Col-2
Row0      1 [4052]  2  [4054]  3 [4056]
Row1      4 [4058]  5  [4060]  6 [4062]
Row2      7 [4064]  8  [4066]  9 [4068]
```

Explanation:

In the above program, two-dimensional array is declared and initialized. Using two nested **for** loops elements of array together with their addresses are displayed. It is shown at the output that elements of two-dimensional array are displayed in rectangle form. But, in memory they are not stored in this particular format. They are stored in contiguous memory location as shown in Table 7.6.

Table 7.6 Memory map of two-dimensional array elements

Row,col	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]	A[2][0]	A[2][1]	A[2][2]
Value	1	2	3	4	5	6	7	8	9
Address	4052	4054	4056	4058	4060	4062	4064	4066	4068

- 7.34 Write a program to display the balance and code number in two separate columns. Indicate the number of code, which are having the balance of less than 1000.

```
void main()
{
    int bal[5][2],i,j=0;
    clrscr();
    printf("\nEnter Code No & Balance:\n");
    for(i=0;i<5;i++)
        scanf("%d %d",&bal[i][1],&bal[i][2]);
    printf("Your Entered Data :");
    for(i=0;i<5;i++)
    {
        printf("\n%d %d",bal[i][1],bal[i][2]);
        if(bal[i][2]<1000)
            j++;
    }
    printf("\nBalance Less than 1000 are %d",j-1);
}
```

OUTPUT:

Enter Code No & Balance:

```
1 900
2 800
3 1200
4 550
5 600
```

Your Entered Data :

```
1 900
2 800
3 1200
4 550
5 600
```

Balance Less than 1000 are 4

Explanation:

The above program finds the number of balance deposits less than 1000. Through the keyboard the **codeno** and **balance** deposits are entered. The first **for** loop is initialized from 0 to less than five for inputting the **codenos** and their balance *deposits*. The second **for** loop is for displaying the entered elements. The **if** statement checks whether the balance deposit is less than 1000 or not. With this, the **codenos** having deposits less than 1000 are sorted out. The counter gets incremented when deposits are less than 1000.

- 7.35 Write a program to initialize single- and two-dimensional arrays. Accept three elements in single-dimension array. Using the elements of this array, compute addition, square and cube. Display the results in two-dimensional arrays.

```
void main()
{
    int i,j=0, a[3][3],b[3];
```

```

    clrscr();
    printf("\n Enter Three Numbers :\n");
    for(i=0;i<=2;i++)
        scanf("%d", &b[i]);
    for(i=0;i<=2;i++)
    {
        a[i][j] = b[i]*2;
        a[i][j+1]=pow(b[i],2);
        a[i][j+2]=pow(b[i],3);
    }
    clrscr();
    printf("Number\tAddition\t\t Square\t\t Cube\n");
    for(i=0;i<=2;i++)
    {
        printf("\n%d", b[i]);
        for(j=0;j<=2;j++)
            printf("\t%4d\t", a[i][j]);
        printf("\n");
    }
    getch();
}

```

OUTPUT:

Enter Three Numbers : 5 6 7

Number	Addition	Square	Cube
5	10	25	125
6	12	36	216
7	14	49	343

Explanation:

The one-dimensional array is used for storing the elements entered through the keyboard. The second **for** loop performs operations such as multiplication (double), square and cube. The square and cube of the entered numbers are performed by standard library function **pow()** defined in **math.h**. The results obtained are assigned to the corresponding elements of two-dimensional arrays. The elements of two arrays are displayed.

- 7.36 Read the matrix of the order up to 10×10 elements and display the same in the matrix form.

```

void main()
{
    int i,j,row,col,a[10][10];
    clrscr();
    printf("\n Enter Order of matrix up to (10 X 10) A :");
    scanf("%d %d",&row,&col);
    printf("\nEnter Elements of matrix A :\n");
    for(i=0;i<row;i++)
        for(j=0;j<col;j++)

```

```

scanf("%d",&a[i][j]);
printf("\n The Matrix is: \n");
for(i=0;i<row;i++)
{
    for(j=0;j<col;j++)
        printf("%6d",a[i][j]);
    printf("\n");
}
}

```

OUTPUT:

```

Enter Order of matrix up to (10 X 10) A : 3 3
Enter Elements of matrix A :
3 5 8
4 8 5
8 5 4
The Matrix is :
3 5 8
4 8 5
8 5 4

```

Explanation:

In the above program, the order of the matrix is entered. The order of the matrix should be up to 10×10 . For the sake of understanding, we have taken 3×3 matrix. Using the first two nested **for** loops with respect to rows and columns the elements of matrix are entered. The last two nested **for** loops are used for displaying the matrix elements with respect to row and column.

Transpose of the matrix: The transpose of matrix interchanges rows and columns, i.e. the row elements become column elements and vice versa.

- 7.37 Read the elements of the matrix of the order up to 10×10 and transpose its elements.

```

void main()
{
    int i,j,row1,row2,col1,col2,a[10][10],b[10][10];
    clrscr();
    printf("\n Enter Order of matrix up to (10 X 10) A :");
    scanf("%d %d",&row1,&col1);
    printf("\nEnter Elements of matrix A : \n");
    for(i=0;i<row1;i++)
    {
        for(j=0;j<col1;j++)
            scanf("%d",&a[i][j]);
    }
    row2=col1;
    col2=row1;

```



```

for(i=0;i<row1;i++)
{
    for(j=0;j<col1;j++)
        b[j][i]=a[i][j];
}
printf("\n The Matrix Transpose is \n");
for(i=0;i<row2;i++)
{
    for(j=0;j<col2;j++)
        printf("%4d",b[i][j]);
    printf("\n");
}
}

```

OUTPUT:

Enter Order of matrix up to (10 X 10) A : 3 3

Enter Elements of matrix A :

3 5 8

4 8 5

8 5 4

The Matrix is :

3 4 8

5 8 5

8 5 4

Explanation:

This program is the same as the last one. The difference between them is the first program displays the elements in the order in which they are entered. But in this program row and column elements are interchanged. This is obtained by interchanging the order of matrix through the statements `row2=col1; & col2=row1;`

- 7.38 Write a program to perform addition and subtraction of two matrices whose orders are up to 10×10 .

```

void main()
{
    int i,j,r1,c1,a[10][10],b[10][10];
    clrscr();
    printf("Enter Order of Matrix A & B up to 10 X 10:");
    scanf("%d %d", &r1,&c1);
    printf("Enter Elements of Matrix of A :\n");
    for(i=0;i<r1;i++)
    {
        for(j=0;j<c1;j++)
            scanf("%d",&a[i][j]);
    }
}

```

```

printf("Enter Elements of Matrix of B :\n");
for(i=0;i<r1;i++)
{
    for(j=0;j<c1;j++)
        scanf("%d",&b[i][j]);
}
printf("\nMatrix Addition \n");
for(i=0;i<r1;i++)
{
    for(j=0;j<c1;j++)
        printf("%5d",a[i][j]+b[i][j]);
    printf("\n");
}
printf("\nMatrix Subtraction \n");
for(i=0;i<r1;i++)
{
    for(j=0;j<c1;j++)
        printf("%5d",a[i][j]-b[i][j]);
    printf("\n");
}
getch();
}

```

OUTPUT:

Enter Order of Matrix A & B up to 10 X 10: 3 3

Enter Elements of Matrix of A :

4 5 8

2 9 8

2 9 4

Enter Elements of Matrix of B :

1 3 5

0 5 4

6 7 2

Matrix Addition

5 8 13

2 13 12

8 16 6

Matrix Subtraction

3 2 3

2 4 4

-4 2 2

Explanation:

The elements of two matrices are read in the same manner as described in the previous examples. Addition and subtraction are computed and the results are displayed. Addition of corresponding elements of A and B matrices is performed by the statement `a[i][j]+b[i][j]`. Similarly, for subtraction `a[i][j]-b[i][j]` statement is used.

- 7.39 Write a program to perform multiplication of two matrices whose orders are up to 10×10 .

```

void main()
{
    int i,j,k,r1,c1,a[10][10],b[10][10],k,c[10][10];
    clrscr();
    printf("Enter Order of Matrix A & B up to 10 X 10:");
    scanf("%d %d", &r1,&c1);
    printf ("Enter Elements of Matrix of A:\n");
    for(i=0;i<r1;i++)
    {
        for(j=0;j<c1;j++)
            scanf ("%d",&a[i][j]);
    }
    printf("Enter Elements of Matrix of B :\n");
    for(i=0;i<r1;i++)
    {
        for(j=0;j<c1;j++)
        {
            scanf ("%d",&b[i][j]);c[i][j]=0;
        }
    }
    printf("\n Matrix Multiplication \n");
    for(i=0;i<r1;i++)
    {
        for(j=0;j<c1;j++)
        {
            for(k=0;k<r1;k++)
            {
                c[k][i]=c[k][i]+a[k][j]*b[j][i];
            }
        }
    }
    for(i=0;i<r1;i++)
    {
        for(j=0;j<r1;j++)
            printf("%5d",c[i][j]);
        printf("\n");
    }
    getch();
}

```

OUTPUT:

```

Enter Order of Matrix A & B up to 10 X 10: 3 3
Enter Elements of Matrix of A :
4 5 8
2 9 8
2 9 4
Enter Elements of Matrix of B :
1 3 5

```

```

0 5 4
6 7 2
Matrix Multiplication
52 93 56
50 107 62
26 79 54

```

Explanation:

This program is the same as the last one. With the help of the first two `for` loops the elements of the first matrix are entered. In a similar way, by using the next two `for` loops, the elements of the second matrix are entered. The multiplication is performed in the last two `for` loops. In this program, multiplication of corresponding elements of two matrices are performed.

- 7.40 Write a program to read the quantity and price of various Pentium models using an array. Compute the total cost of all models.

```

void main()
{
    int i;
    long pc[4][2];
    long t=0;
    clrscr();
    for(i=0;i<4;i++)
    {
        printf("\t Enter Qty. & Price for Pentium%d :",i+1);
        scanf("%ld %ld",&pc[i][0],&pc[i][1]);
    }
    clrscr();
    printf("=====\n");
    printf("Model\t Qty.\tRate (Rs.) Total Value");
    printf("\n=====");
    for(i=0;i<=3;i++)
    {
        printf("\nPentium%d    %2ld    %6ld    %15ld",i+1,pc[i][0],pc[i][1],
            pc[i][0]*pc[i][1]);
        t=t+pc[i][0]*pc[i][1];
    }
    printf("\n=====\n");
    printf("Total Value of All PCs in Rs. %ld",t);
    printf("\n=====");
}

```

OUTPUT:

```

Enter Qty. & Price for Pentium1 :25 25000
Enter Qty. & Price for Pentium2 :20 40000
Enter Qty. & Price for Pentium3 :15 35000
Enter Qty. & Price for Pentium4 :20 40000

```

```

=====
Model    Qty.    Rate (Rs.)    Total Value
=====
Pentium1  25      25000        625000
Pentium2  20      30000        600000
Pentium3  15      35000        525000
Pentium4  20      40000        800000
=====
Total Value of All PCs in Rs.2550000
=====

```

Explanation:

Two-dimensional array is initialized for entering the quantity and price. The first **for** loop is used for entering the quantity and price of four models through keyboard. The second **for** loop is for computing total value of each model. This is obtained by multiplying price of each model with quantity. The product is added to variable 't'. At the beginning, the value of 't' is 0. The equation $t = t + pc[i][0] * pc[i][1]$ is executed four times for finding the total cost of all the models. The output displays model, quantity, price and total value of each model and the gross value.

- 7.41 Write a program to read the capacity of HD, its price and quantity available in the form of an array. Compute the cost of HD.

```

void main()
{
    int i,j;
    long hd[3][4],t=0;
    clrscr();
    for(j=0;j<4;j++)
    {
        printf("\t Enter Capacity,Price & Qty.:");
        for(i=0;i<3;i++)
        {
            scanf("%ld",&hd[i][j]);
        }
    }
    clrscr();
    printf("=====\n");
    printf("HD Capacity GB Price Rs.\tQuantity Total Value Rs.");
    printf("\n=====");
    for(j=0;j<4;j++)
    {
        for(i=0;i<3;i++)
        {
            printf("%2ld",hd[i][j]);
            printf("\t\t");
        }
    }
}

```

```

        if (i==2)
        {
            printf("%5ld",hd[i-1][j]*hd[i][j]);
            t=t+hd[i-1][j]*hd[i][j];
        }
    }
    printf("\n");
}
printf("=====\n");
printf("Total Value Rs. %37ld",t);
printf("\n=====\\n");
}

```

OUTPUT:

```

Enter Capacity, Price & Qty.: 10 8000 25
Enter Capacity, Price & Qty.: 20 12000 20
Enter Capacity, Price & Qty.: 40 15000 15
Enter Capacity, Price & Qty.: 90 20000 10
=====
HD Capacity GB      Price Rs.   Quantity Total Value
=====
10   8000    25          200000
20  12000    20          240000
40  15000    15          225000
90  20000    10          200000
=====
      Total Value Rs. 865000
=====

```

Explanation:

In the above program, two-dimensional array `hd[3][4]` is initialized. The first two `for` loops are used for reading capacity, price and quantity of hard disk. The next two `for` loops are used for evaluating the product of quantity and price of hard disk. Here, the first row contains the capacity, the *second* row contains price and *third* row contains the quantity of hard disk. Here, *first* row does not require for computing purpose. To avoid this, the `if` statement is used. Total cost is obtained by using `printf("%5ld",hd[i-1][j]*hd[i][j])` and `t=t+hd[i-1][j]*hd[i][j]` statements.

- 7.42 Write a program to display the names of the cities with their base addresses.

```

void main()
{
    int i;
    char city[5][8]={
        "Mumbai",
        "Chennai",
        "Kolkata",
        "Pune",
        "Delhi" };
}

```

```

clrscr();
for(i=0;i<5;i++)
{
    printf("Base Address = %u",&city[i]);
    printf("String = %s\n",city[i]);
}
}

```

OUTPUT:

```

Base Address = 4028 String = Mumbai
Base Address = 4036 String = Chennai
Base Address = 4044 String = Kolkata
Base Address = 4052 String = Pune
Base Address = 4060 String = Delhi

```

Explanation:

In the above program, a two-dimensional character array `array[][]` is initialized with city names. It prints the city name and base address using the '&' operator. In all the base addresses, there is difference of eight locations because every string occupies 8 bytes. Though the string name occupies less than eight characters, the total space eight, which is allocated, is taken in account.

- 7.43 Write a program to display the binary bits corresponding to Hexadecimal numbers from 0 to F. Attach an odd parity to Hex numbers and display them.

```

void main()
{
    int bin[16][4], x, y, a=0, c=0;
    for(x=0; x<16; x++)
    {
        y=x;
        for(a=0; a<4; a++)
        {
            bin[x][a]=y%2;
            y=y/2;
        }
    }
    clrscr();
    printf("\n Input Bits Parity Bits");
    printf("\n =====");
    for(x=0; x<16; x++)
    {
        c=0;
        printf("\n");
        for(y=3; y>=0; y--)
        {
            printf("%3d", bin[x][y]);

```

```

        if(bin[x][y]==1)
            c++;
    }
    if (c%2==0)
        printf("%7d",1);
    else
        printf("%7d",0);
    }
}

```

OUTPUT:

Binary Bits	Parity Bits
=====	=====
0 0 0 0	1
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	0
0 1 0 1	1
0 1 1 0	1
0 1 1 1	0
1 0 0 0	0
1 0 0 1	1
1 0 1 0	1
1 0 1 1	0
1 1 0 0	1
1 1 0 1	0
1 1 1 0	0
1 1 1 1	1

Explanation:

In the above program, two `for` loops are used. The inner `for` loop generates four binary bits for each hexadecimal number and assigns it to array `bin[][]`. The 0 to F hexadecimal numbers are taken from the outer `for` loop. If the bit is 1 counter 'c' is incremented. The value of 'c' is checked for even or odd condition. If 'c' is odd the parity bit '0' is displayed otherwise '1'. The output is displayed.

➤ 7.44 Write a program to convert binary to gray codes.

```

void main()
{
    int bin[16][4], x, y, a=0, c=0;
    clrscr();
    for(x=0; x<16; x++)
    {
        y=x;
        for(a=0; a<4; a++)

```



```

        {
            bin[x][a]=y%2;
            y=y/2;
        }
    }

    clrscr();
    printf("\n Binary Bits Gray Bits");
    printf("\n =====");
    for(x=0;x<16;x++)
    {
        printf("\n");
        for(y=3;y>=0;y--)
            printf("%3d",bin[x][y]);
        printf("%5d  %2d  %2d  %2d",bin[x][3],  bin[x][3]^bin[x][2],bin[x]
        [2]^bin[x][1],bin[x][1]^bin[x][0]);
    }
}

```

OUTPUT:

Binary Bits =====	Gray Bits =====
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 0	0 0 1 1
0 0 1 1	0 0 1 0
0 1 0 0	0 1 1 0
0 1 0 1	0 1 1 1
0 1 1 0	0 1 0 1
0 1 1 1	0 1 0 0
1 0 0 0	1 1 0 0
1 0 0 1	1 1 0 1
1 0 1 0	1 1 1 1
1 0 1 1	1 1 1 0
1 1 0 0	1 0 1 0
1 1 0 1	1 0 1 1
1 1 1 0	1 0 0 1
1 1 1 1	1 0 0 0

Explanation:

In this program, the binary bits corresponding to the Hex are obtained. The first two `for` loops accomplish this. Exclusive OR operation with previous and successive bits obtains the gray code. The MSB bits should not be changed.

7.11 THREE- OR MULTI-DIMENSIONAL ARRAYS

The C program allows array of two- or multi-dimensions. The compiler determines the restriction on it. The syntax of multi-dimensional array is follows:

```
data_type  array_ name[size1][size2][size3]...[sizei];
```

where S_i is the size of the i th dimensions.

Three-dimensional array can be initialized as follows:

```
int mat[3][3][3] =
{
    1, 2, 3,
    4, 5, 6,
    7, 8, 9,
    1, 4, 7,
    2, 5, 8,
    3, 6, 9,
    1, 4, 4,
    2, 4, 7,
    8, 8, 5};
```

A three-dimensional array can be thought of an array of arrays. The outer array contains three elements. The inner array size is two-dimensional with size[3][3].

➤ 7.45 Write a program to explain the working of three-dimensional array.

```
void main()
{
    int array_3d[3][3][3];
    int a,b,c;
    clrscr();
    for(a=0;a<3;a++)
    for(b=0;b<3;b++)
    for(c=0;c<3;c++)
    array_3d[a][b][c]=a+b+c;
    for(a=0;a<3;a++)
    {
        printf("\n");
        for(b=0;b<3;b++)
        {
            for(c=0;c<3;c++)
            printf("%3d", array_3d[a][b][c]);
            printf("\n");
        }
    }
}
```

OUTPUT:

```
0 1 2
1 2 3
2 3 4
1 2 3
2 3 4
```

```

3 4 5
2 3 4
3 4 5
4 5 6

```

Explanation:

The three-dimensional array `array_3d` is initialized. The first three `for` loops are used for adding the values of `a`, `b` and `c`. Here, initially `a` and `b` are zero and '`c`' varies from 0 to 2. Hence, the addition of `a`, `b` and `c` will be 0 1 2. This will be printed in the first row. In the second output row in which `a = 0` and `b = 1`, `c` varies from 0 to 2. Thus, the output of the second row will be 1 2 3. In this way, the values of `a`, `b` and `c` are changed and the total 27 iterations are carried out.

➤ 7.46 Write a program to explain the working of four-dimensional array.

```

void main()
{
    int array_4d[2][2][2][2];
    int a,b,c,d;
    clrscr();
    for(a=0;a<2;a++)
    for(b=0;b<2;b++)
        for(c=0;c<2;c++)
            for(d=0;d<2;d++)
                array_4d[a][b][c][d]=a+b+c+d;
    for(a=0;a<2;a++)
    {
        printf("\n");
        for (b=0;b<2;b++)
        {
            for(c=0;c<2;c++)
            {
                for(d=0;d<2;d++)
                    printf("%3d",array_4d[a][b][c][d]);
                printf("\n");
            }
        }
    }
}

```

OUTPUT:

```

0 1
1 2
1 2
2 3
1 2
2 3
2 3
3 4

```

Explanation:

Here, in the above-cited program, instead of three-dimension, four-dimension array is used. The operations performed are the same as explained in the previous example.

- 7.47 Write a program to read the quantity and rate of certain items using multi-dimensional array. Calculate the total cost by multiplying the quantity and rate and offer 2% discount on it and display the net amount.

```
void main()
{
    long m[2][2][2][2];
    int a,b,c,d;
    clrscr();
    printf("Enter Quantity & Rate\n");
    for(a=0;a<2;a++)
        for(b=0;b<2;b++)
            for(c=0;c<2;c++)
                for(d=0;d<1;d++)
                {
                    if(a==0)
                    {
                        printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
                        scanf("%ld %ld",&m[a][b][c][d],&m[a][b][c][d+1]);
                    }
                    else
                    {
                        m[a][b][c][d]=m[a-1][b][c][d]*m[a-1][b][c][d+1];
                        m[a][b][c][d+1]=m[a-1][b][c][d]*m[a-1][b][c][d+1]*2/100;
                    }
                }
    printf("\t=====\n");
    printf("\tQuantity Rate Amount Discount (@2%) Net Amount\n");
    printf("\t=====\n");
    for(a=0;a<1;a++)
        for(b=0;b<2;b++)
            for(c=0;c<2;c++)
                for(d=0;d<1;d++)
                {
                    printf("\n%10ld %10ld",m[a][b][c][d],m[a][b][c][d+1]);
                    printf("%8ld.00 %6ld.00 %12ld.00", m[a+1][b][c][d],m[a+1][b][c][d+1],
                        m[a+1][b][c][d]-m[a+1][b][c][d+1]);
                }
    printf("\n\t=====\n");
    getch();
}
```

OUTPUT:

```
Enter Quantity & Rate
A=0 b=0 c=0 d=0
25 50
```

```

A=0 b=0 c=1 d=0
30 60
A=0 b=1 c=0 d=0
35 70
A=0 b=1 c=1 d=0
40 75

```

Quantity	Rate	Amount	Discount (@2%)	Net Amount
25	50	125000	25.00	1225.00
30	60	1800.00	36.00	1764.00
35	70	2450.00	49.00	2401.00
40	75	3000.00	60.00	2940.00

Explanation:

In the above example, the four-dimension array `m[2][2][2][2]` is declared. The first four `for` loops are used to read the quantity and rate. The values of variables '*a*' and '*d*' for four times remain zero whereas the values of '*b*' and '*c*' change to 1,(0,0), 2 (0,1), 3 (1,0) and 4 (1,1). This happens while execution of the `for` loops. The `if` statement checks as to whether the value of *a* = 0 or not. As long as its value is zero reading operation is performed. When it is greater than zero rate and quantity are multiplied for obtaining the amount. Also discount is calculated. Amount and discount are stored in the array `m[2][2][2][2]`. Net amount is printed after subtracting discount from the gross amount.

➤ 7.48 Write a program to demonstrate the use of three-dimensional array.

```

void main()
{
    int a,b,c;
    int mat[3][3][3] ={ 1,2,3,
                        4,5,6,
                        7,8,9,
                        1,4,7,
                        2,5,8,
                        3,6,9,
                        1,4,4,
                        2,4,7,
                        8,8,5 };

    clrscr();
    for(a=0;a<3;a++)
    { printf("\n");
      for (b=0;b<3;b++)
      {
          for(c=0;c<3;c++)
              printf("%3d [%u]",mat[a][b][c],&mat[a][b][c]);
          printf("\n");
      }
    }
}

```

OUTPUT:

```

1 [65470] 2 [65472] 3 [65474]
4 [65476] 5 [65478] 6 [65480]

```

```

7 [65482] 8 [65484] 9 [65486]
1 [65488] 4 [65490] 7 [65492]
2 [65494] 5 [65496] 8 [65498]
3 [65500] 6 [65502] 9 [65504]
1 [65506] 4 [65508] 4 [65510]
2 [65512] 4 [65514] 7 [65516]
8 [65518] 8 [65520] 5 [65522]

```

Explanation:

In the above example, a three-dimensional array is initialized. The three loops are used to access the elements. The logic of accessing elements is the same as two-dimensional array. From the output obtained, we can say that elements of multi-dimensional array are stored in continuous memory locations.

7.12 THE `sscanf()` AND `sprintf()` FUNCTIONS

(a) `sscanf()`:

The `sscanf()` function allows to read characters from a character array and writes them to another array.

This function is similar to `scanf()` but instead from standard input it reads data from an array.

➤ 7.49 Write a program to read string from a character array.

```

void main()
{
    char in[10],out[10];
    clrscr();
    gets(in);
    sscanf(in,"%s",out);
    printf("%s\n",out);
}

```

OUTPUT:

```

HELLO
HELLO

```

Explanation:

In the above program, two character arrays `in[10]` and `out[10]` are declared. The `gets()` function reads the string through the terminal and it is stored in the array `in[]`. Till this time, the `out[]` array is empty. The `sscanf()` function reads characters from array `in[]` and assigns it to array `out[]`. Thus, both the arrays contain the same string. At the end, the `printf()` function displays the contents of array `out[]`.

➤ 7.50 Write a program to read integers in character array, convert and assigns them to integer variable using `sscanf()` function.

```

void main()
{
    int *x;

```

```

char in[10];
clrscr();
printf("\n Enter Integers :");
gets(in);
sscanf(in, "%d", x);
printf("\n Value of int x : %d", *x);
getche();
}

```

OUTPUT:

```

Enter Integers : 123
Value of int x : 123

```

Explanation:

In the above program, integer is read and stored in the character array `in[]`. The variable 'x' is declared as integer pointer. The `sscanf()` function assigns base address of array `in[]` to pointer 'x'. The content of pointer 'x' is displayed using pointer notation (*).

(b) sprintf():

The `sprintf()` function is similar to the `printf()` function except a small difference between them. The `printf()` function sends the output to the screen whereas the `sprintf()` function writes the values of any data type to an array of characters. The following program is illustrated pertaining to `sprintf()`.

➤ 7.51 Write a program to explain the use of `sprintf()`.

```

void main()
{
    int a=10;
    char c='C';
    float p=3.14;
    char spf[20];
    clrscr();
    sprintf(spf, "%d %c %.2f", a, c, p);
    printf("\n%s", spf);
    getche();
}

```

OUTPUT:

```

10 C 3.14

```

Explanation:

In the above program, `sprintf()` stores the values of variables in the character array `spf[]`. When `sprintf()` function executes, the contents of variables are not be displayed on the screen. The contents of array `spf` is displayed using `printf()` function.

- 7.52 Write a program to complete the string by filling the spaces with appropriate characters.

```
void main()
{
    char name[]="Mas er ng A SI C";
    int x=0;
    clrscr();
    printf("Mastering ANSI C\n\n");
    while(name[x]!=0)
    {
        if(name[x]==32)
            name[x]=getche();
        else
            putchar(name[x]);
        x++;
    }
    getche();
}
```

OUTPUT:

```
Mastering ANSI C
Mastering ANSI C
```

Explanation:

In the above program, a string is initialized with array `name []`. The string is not complete and contains blanks. The `if` statement within the `while()` loop checks every character of the string. If blank space is found, `if` block is executed and user needs to enter a character. Otherwise, the next successive characters are displayed.

7.13 DRAWBACKS OF LINEAR ARRAYS

1. The number of elements, which are to be stored in an array, is to be known first.
2. When an array is declared, memory is allocated to it. If array is not filled completely, the vacant place occupies memory.
3. Once the array is declared, its dimension can be changed.

SUMMARY

You have now learnt how to initialize an array by different ways. The Characteristics of an array have been discussed in depth. How to specify the elements of one-dimensional, two-dimensional and three- or multi-dimensional arrays is explained in detail together with ample examples. The functions such as the `sscanf()` and `sprintf()` functions are demonstrated through examples. In this chapter, the reader finds practical examples. The programmer should make an attempt to execute all the programs so as to get expertise in arrays. Unless the unsolved problems are taken up for solving, the depth of the chapter would not be followed.

EXERCISES

I Fill in the blanks:

1. In an array `x[10]`, the `x` represents the _____.
 - (a) base address
 - (b) base value
 - (c) void pointers
 - (b) 0th element
 - (c) 1st element
2. Each array element is stored in separate _____.
 - (a) memory locations
 - (b) value
 - (c) data type
3. `int x[5]`; this array can hold values in between _____ to _____.
 - (a) -32768 to 32767
 - (b) 0 to 255
 - (c) -32768 to 32768
4. `int x[5]={2, 3, 4, 5, 6}`; the base address is 65564 then the location of element 2 is _____.
 - (a) 65564
 - (b) 65566
 - (c) 65568
5. `int x[3]={1, 2, 3}`; the address of `x[2]` is 65498, i.e. the base address of the array is _____.
 - (a) 65494
 - (b) 65492
 - (c) 65496
6. Output?


```
void main()
{
    int x[]={1, 2.2, 3}, i;
    clrscr();
    printf("%f", x[1]);
}
```

 - (a) the program will not run
 - (b) the program will run with warning messages
 - (c) None of the above
7. The 1st element according to C compiler physically is a _____.
 - (a) 2nd element
8. In `long k[4]` the total memory occupied by the array is _____.
 - (a) 16
 - (b) 8
 - (c) 4
9. When in between two array of same type values are interchanged _____.
 - (a) address of elements are also changed
 - (b) only base address of the array is changed
 - (c) None of the above happens
10. `int x[3]` if `x[0]=12` and `x[2]=26` then `x[3]` is _____.
 - (a) 0
 - (b) 15
 - (c) garbage value
11. Array element counting starts from zero hence the statement `x[0]` defined can hold _____.
 - (a) nothing
 - (b) one integer value
 - (c) one char value
12. `char c[]="Hello"` the H is stored at 65470 then the o is stored at _____.
 - (a) 65474
 - (b) 65475
 - (c) 65480
13. Array element of two dimensions are stored at _____.
 - (a) subsequent memory locations
 - (b) alternate memory locations
 - (c) any locations
14. In _____, values of the array elements are passed to function
 - (a) call by value
 - (b) call by reference
 - (c) None of them

15. In _____, addresses of the array elements are passed to function
- call by value
 - call by reference
 - None of them
16. Fast access of array elements can be done using _____
- pointer
 - call by value
 - call by reference

II True or false:

- When we declare array elements without initialization, then its elements are set to zero.
- If we declare static array without initialization, then its element are set to garbage value.
- When the size of an array is exceeded, then compiler will not show any error.
- It is not essential to mention the size of array while initializing it at the time of declaration.
- Speed of accessing array elements with pointers is slower than accessing by subscripts.
- Two-dimensional array is also called matrix.
- $S[i][j]$ is two-dimensional array where $-1 < i < 4$ and $-1 < j < 2$ then we can say $S[1][1] = *(*(S+1)+1)$.
- Array contains elements of the same data type which are stored contiguously in memory.
- Default storage class is static
- Storage class decides when to create and destroy variable.
- An integer array always terminates with '\0' (null).
- When we assign one array to other, array elements of one are directly copied in to the other array.
- The output of the following program is 'We won First Test against SriLanka'

```
void main()
{
    int num[]={24,34,12,44,57,9};
    int i = 0;
```

```
    if(num[i]=i[num])
        printf("\n We won First
        Test against SriLanka");
    else
        printf("\n We won Second Test
        against SriLanka");
    getch();
}
```

14. Output of the following program is 4

```
void main()
{
    int arr[]={10,20,30,45,67,
    56,74};
    int *i,*j;
    i=&arr[1];
    j=&arr[5];
    printf("%d",j-i);
    getch();
}
```

15. Output of the following program is 67

```
void main()
{
    int
    arr[]={10,20,30,45,67,
    56,74};
    int *j;
    clrscr();
    j=&arr[5];
    printf("%d",*(j-1));
    getch();
}
```

16. Character array never ends with '\0' (NULL).

III Select the appropriate option from the multiple choices given below:

- An array is a collection of
 - elements of different data types
 - same data types
 - Both A and B
- Array elements are stored in
 - scattered memory locations
 - sequential memory locations
 - Both (a) and (b)

3. A character array always ends with

- (a) null ('\0') character
- (b) question Mark (?)
- (c) full Stop (.)

4. If you declare array without static the elements will be set to

- (a) null value
- (b) zero
- (c) garbage value

5. Arrays cannot be initialized if they are

- (a) automatic
- (b) external
- (c) static
- (d) None of the above

6. All the elements in the array must be

- (a) initialized
- (b) defined
- (c) Both (a) and (b)
- (d) None of the above

7. What will be the output of the following program

```
void main()
{
    int a1[5]={1};
    int b=0,k=0;
    clrscr();
    for(b=0;b<=4;b++)
    {
        printf("%3d",++a1[0]);
    }
}
```

- (a) 2 3 4 5 6
- (b) 1 2 3 4 5
- (c) 1 1 1 1 1
- (d) 1 2 2 2 2

8. What will be the value of *x?

```
void main()
{
    int *x;
    char in[10]="10";
    clrscr();
    sscanf(in,"%d",x);
    printf("\n Value of int x : %d",*x);
}
```

- (a) 10
- (b) 012
- (c) "1"

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

```
void main()
{
    int j[]={5,1,2,5,4,8};
    int m[]={1,5,8,4,5,9};
    int l[]={1,2,9,1,5,9},k;
    for(k=0;k<6;k++)
        l[k]=j[k]+m[k]-l[k];
    clrscr();
    for (k=0;k<6;k++)
        printf("%d", l[k]);
}
```

- (a) 5 4 1 8 4 8
- (b) 5 5 1 8 4 9
- (c) 1 2 5 5 5 9

10. What is the output of the following program?

```
void main()
{
    int j[]={15,11,17,15,14,18};
    int m[]={1,5,8,4,5,9};
    int l[]={1,2,9,1,5,9},k;
    for(k=0;k<6;k++)
        l[k]=j[k]-m[k]/l[k];
    clrscr();
    for(k=0;k<6;k++)
        printf ("%d", l[k]);
}
```

- (a) 14 9 17 11 13 17
- (b) 12 8 16 10 12 16
- (c) 14 19 17 10 12 11

11. What is the output of the following program?

```
void main()
{
    char a[]={'A','B','C','D','E','F'};
    char b[]={'0','1','2','3','4','4'};
    char c[6],k;
    for(k=0;k<6;k++)
        c[k]=a[k]+b[k];
    clrscr();
    for(k=0;k<6;k++)
        printf("%c", c[k]);
}
```

- (a) q s u w y z
(b) Q S U W Y Z
(c) K J L M N O
12. What will happen if you enter (at run time) more values into an array than its capacity?
- (a) program will be terminated
(b) run time error message will be displayed
(c) No bug
13. What will happen if you assign values in few locations of an array?
- (a) rest of the elements will be set to 0
(b) compiler error message will be displayed
(c) possible system will crash
14. When an array is passed to function, in real what gets passed?
- (a) base address of the array
(b) element numbers
(c) values of the array
15. Which is the correct statement to declare an array?
- (a) `int x[];`
(b) `int x[5];`
(c) `int x{5}'`
16. The array name itself is pointer to
- (a) 0th element
(b) 1st element
(c) last element
17. The array name itself is a
- (a) constant pointer object
(b) address
(c) None of the above
18. `int x[5], *p;` and `p=x;` then following which one operation is wrong ?
- (a) `x++;`
(b) `p++;`
(c) `p=x[1];`
19. `int x[3];` the base address of x is 65460 the elements are stored at locations
- (a) 65460, 65462, 65464
(b) 65460, 65461, 65462
(c) 65460, 65464, 65468
20. `int j[4]` the `sizeof(j)` and `sizeof(int)` will display the value
- (a) 8,2
(b) 2,8
(c) 2,2

IV Attempt the following programming exercises:

- Write a program to read 10 integers in an array. Find the largest and smallest number.
- Write a program to read a number containing five digits. Perform square of each digit. For example, number is 45252. Output should be square of each digit, i.e. 4 25 4 25 16.
- Write a program to read a number of any lengths. Perform the addition and subtraction on the largest and smallest digits of it.
- Write a program to read three digits positive integer number 'n' and generate possible permutation of numbers using their digits. For example, n=123 then the permutations are 123, 132, 213, 231, 312, 321⁸.
- Write a program to read the text. Find out number of lines in it.
- Write a program to read any three characters. Find out all the possible combinations.
- Write a program to enter string in lower and uppercase. Convert lower to uppercase and vice versa and display the string.
- Read the marks of five subjects obtained by five students in an examination. Display the top two student codes and their marks.
- Write a program to enter five numbers using array and rearrange the array in the reverse order. For Example numbers entered are 5 8 3 2 4 and after arranging array elements must be 4 2 3 8 5.
- Accept a text up to 50 words and perform the following actions.
 - Count total vowels, consonants, spaces, sentences and words with spaces.
 - Program should erase more than one space between two successive words.

Hint: total number of sentences can be computed based on total number of full stops.

11. Evaluate the following series. Calculate every term and store its result in an array. Using array calculate the final result.

- (a) $x = 1^1 + 2^2 + 3^3 + 4^4 \dots n^n$
 (b) $x = 1! + 2! + 3! + \dots n!$
 (c) $x = 1! - 2! + 3! - \dots n!$

12. Refer the given below tables. Write a program to calculate bill amount of stationary articles. Use array for initializing items and their prices.

(a)

Item	Price (₹)
10th Book Set	850
12th Book Set	1150
Note books	Rate (Rs.)
100 Pages/dozen	75
200 pages/dozen	125
PEN SET	50

- (b) 15% discount if sets are more than 10 otherwise 10%.
 (c) 10% discount if notebooks are more than 10 dozen.

13. Consider the following table and write a program to find the total cost of the computer systems using arrays. 10% discount is to be offered in case the total systems sold are more than 5 in quantity. Display the code, parts of the system and price. User should enter the configuration and quantity.

Code No.	Particulars	Price (₹)
1	MOTHER BOARD WITH PROCESSOR P3 600 MHz	3000
2	MOTHER BOARD WITH PROCESSOR P4 1000 MHz	8000
3	HD (200 GB)	4000
4	HD (400 GB)	8000
5	RAM (1 GB)	1000
6	RAM (2 GB)	2000
7	CACHE (256 MB)	1000
8	CACHE (1 GB)	1500
9	FDD (1.44 MB)	1500
10	CD ROM DRIVE	3000

(Continued)

Code No.	Particulars	Price (₹)
11	CABINET	400
12	KEYBOARD	750
13	MOUSE	200
14	MULTIMEDIA KIT	3000
15	MODEM	1500

14. Enter the following information through the key-board using multi-dimension arrays. Calculate the salary of individuals and total salary of all employees.

Code No.	Designation	Basic	DA	Other perks
1	Manager	40000	50%	15%
2	Executive	20000	25%	20%
3	Senior Asst.	15000	25%	20%
4	Junior Asst.	10000	20%	15%
5	Steno	8000	20%	10%
6	Clerk	5000	20%	10%
7	Peon	4000	20%	10%

15. Refer the table given below. Find the total bill amount using arrays. The user should have option to enter the quantity of bulbs.

Code No.	Wattage of the bulb	Price/bulb (₹)
1	15 W	5
2	40 W	8
3	60 W	12
4	100 W	20
5	200 W	35

16. Write a program to display the different parameters of 10 men in a table. The different parameters to be entered are height, weight, age and chest in cm. Find the number of men who satisfy the following condition.

- (a) Age should be greater than 18 and less than 25.
 (b) Height should be between 5.2 to 5.6 inches.
 (c) Weight should be in between 45 to 60 kg.
 (d) Chest should be greater than 45 cm.

(Continued)

17. Write a program to display the items that have been exported between 1995 and 2000. The table should be displayed as follows. Input the amount of various items described as under.

- (a) Find the total and average export value of all the items.
 (b) Find the year in which minimum and maximum export was made.

Items/Year 1995–96 1996–97 1997–98 1998–99 1999–2000

Coffee
 Tea
 Sugar
 Milk powder
 Zinger

18. Write a program to replace the zero with successive number in following arrays.

- (a) `int x[8] = {1, 0, 3, 0, 5, 0, 7, 0}`
 (b) `int y[8] = {-1, 0, -3, 0, -5, 0, -7, 0}`

V Answer the following questions:

1. Explain multi-dimensional array.
2. Array name contains base address of the array. Can we change the base address of the array?
3. What is the relation between array name and element number? How elements are referred using base address.
4. Can we store values and addresses in the same array? Explain.
5. Mention differences between character array and integer array.
6. What are arrays?
7. How elements of an array are stored?
8. Explain two-dimensional array.
9. What are the drawback of liner array?
10. Explain insertion and deletion of an array.
11. Explain various operations with an array.

VI What will be the output/s of the following program/s?

1.

```
void main()
{
    int i, marks[5]={55, 56, 67, 78, 64};
    clrscr();
    for(i=2; i<=4; i++)
        printf("\n %d", marks[i]);
    getch();
}
```

2.

```
void main()
{
    char name []={'a', 'm', 'i', 't'};
    int i;
    clrscr();
    for(i=0; i<=4; i++)
        printf("%c", name[i]);
    getch();
}
```

3. Sum of numbers from 1 to 10

```
void main()
{
    int i, sum=0, num[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
    clrscr();
    for(i=0; i<=9; i++)
        sum=sum+num[i];
    printf("%d", sum);
    getch();
}
```

4. A program on multiplication table of 6

```
void main()
{
    int i, ans, num[5]={1, 2, 3, 4, 5};
    clrscr();
    for(i=0; i<=4; i++)
    {
        ans=num[i]*6;
        printf("\n      %d*%d=%d", 6, num[i], ans);
    }
}
```

5. Factorial of 5

```
void main()
{
    int i, m=1, num[5]={1, 2, 3, 4, 5};
    clrscr();
    for(i=0; i<=4; i++)
```

```

    m=m*num[i];
    printf("Total is %d",m);
    getch();
}

```

6. Searching an element in the array.

```

void main ()
{
    int n=9,i;
    int x[5]={2,3,4,6,7};
    clrscr();
    for(i=0;i<=4;i++)
        printf(" %d",x[i]);
    for(i=0;i<=4;i++)
    {
        if(x[i]==n)
        {
            printf("\n Element %d
found in the array",n);
            getch();
            exit();
        }
    }
}

```

```

    printf("\n Element %d not found
in the array",n);
    getch();
}

```

7. Sum of even and odd numbers from

```

void main()
{
    int sumo=0,sume=0,i=0,odd[5],e
ven[5];
    clrscr();
    for(i=5;i<=10;i++)
    {
        if i%2==0)
            sume=i+sume;
        else
            sumo=i+sumo;
    }
    printf("Addition of even and odd
numbers from 1 to 10 : %d %d",
sume,sumo);
    getch();
}

```

VII Find the bugs in the following program/s:

1.

```

void main()
{
    int a[5]={5,2,3,1,4};
    int i,result=1;
    clrscr();
    for(i=0;i<=5;i++)
        result=result*a[i];
    printf("\nresult of multi-
plication =%d",result);
    getch();
}

```

2.

```

void main()
{
    int a[10]={5,6,7,8,9,3,4,5,4,6};
    int i,sum=0;
    float avg;
    clrscr();
    for(i=0;i<=9;i++)
    {
        sum=sum+a[i];
    }
    avg=(float)sum/10;
    printf("\nSum of ten numbers
=%d",&sum);
    printf("\nAverage of ten
numbers=%.2f",&avg);
}

```

```

    getch();
}

```

3.

```

void main()
{
    int i,num[]={24,34,12,44,56,17};
    int num1[]={12,24,35,78, 85,22};
    clrscr();
    printf("Array 1st - 2nd Array
Addition\n");
    for(i=0;i<=5;i++)
    {
        num[i]=num1[i]+ ++num[i];
        printf("\n %d",num[i]);
    }
}

```

4.

```

void main()
{
    int i,num[]={24,34,12,44,56,17};
    clrscr();
    for(i=1;i<=5;i++)
    {
        printf ("%d",num[i]);
    }
}

```

5.

```
void main()
{
    int i, j;
    int k[]={1,2,4,5,6,7,8,9};
```

```
    clrscr();
    clrscr();
    for(j=0; j<=8; j++)
        printf ("%u", *(k+j));
}
```

ANSWERS

I Fill in the blanks:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	a	4.	a	5.	a
6.	a	7.	b	8.	a	9.	c	10.	a
11.	a	12.	a	13.	a	14.	a	15.	b
16.	a								

II True or false:

Q.	Ans.
1.	False. Default declaration is auto, where elements are initialized to garbage value.
2.	False. They are set to zero.
3.	True. It is not compiler's job to check whether array size is exceeded or not. So no error will be shown by the compiler.
4.	True. E.g.: int n[]={2,4,12,34}; is accepted.
5.	False. Speed of accessing array elements by pointer is always faster.
6.	True.
7.	True. Let us take an example: <pre>void main() { int arr[3][2] = {{1,2},{3,4},{5,6}}; int a = (*(arr + 1) + 1); int b = arr[2][1]; clrscr(); printf("a = %d \t b = %d", a, b); }</pre> OUTPUT: a = 4 b = 6
8.	True.
9.	False. Default storage class is auto.
10.	True.
11.	False. Character array always ends with '\0'.

12.	False. C doesn't allow it. We've to copy content of one array element by element to other array.
13.	True.
14.	True Because, Suppose address start from 0 address of arr[1] = 2 address of arr[5] = 10 ans = (10-2)/2 In windows int has 2 byte size. So divide by 2.
15.	True Access of any array element is possible with pointer
16.	False

III Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	b	2.	b	3.	a	4.	c	5.	d
6.	d	7.	a	8.	a	9.	a	10.	a
11.	a	12.	c	13.	a	14.	a	15.	b
16.	a	17.	b	18.	c	19.	a	20.	a

VI What will be the output/s of the following program/s?

Q.	Ans.	Q.	Ans.
1.	67 78 64	5.	Total is 120
2.	a m i t	6.	2 3 4 6 7
3.	55	7.	Element 9 not found in the array Addition of even and odd numbers from 1 to 10 : 24 21
4.	6*1=6 6*2=12 6*3=18 6*4=24 6*5=30		

VII Find the bugs in the following program/s:

Q.	Ans.
1.	No bug. The for loop should be from 0 to 4 instead up to 5.
2.	& should be removed from the printf statements. Ans Sum 57 and average 5.7
3.	No bug.
4.	No bug.
5.	Attempt to read more elements than declared in array. Last value of array displayed will be garbage.

CHAPTER

8

Strings and Standard Functions

Chapter Outline

- 8.1** Introduction
- 8.2** Declaration and Initialization of String
- 8.3** Display of Strings with Different Formats
- 8.4** String Standard Functions
- 8.5** String Conversion Functions
- 8.6** Memory Functions
- 8.7** Applications of Strings

8.1 INTRODUCTION

Characters are the basic requirement of any program. A program contains statements and statements are built with words that has specific meaning to be understood by the compiler. In C language, a sequence of characters, digits and symbols enclosed within double quotation marks is called a string. The string is always declared as character array and its elements are stored in contiguous memory locations.

In other words, in C, a string is defined as an array of characters. To manipulate text such as words and sentences, normally strings are used. Every string is terminated with '`\0`' (NULL) character. The NULL character is a byte with all bits at logic zero. Hence, its decimal value is zero. A pointer can access the string. The value of the string is its base address, i.e. addresses of the first character. When a string is created, a few compilers place the string in the memory where it cannot be modified.

An example of a string is as follows:

```
char name[]={ 'I', 'N', 'D', 'I', 'A', '\0' };
```

Each character of the string occupies 1 byte of memory. The last character is always '`\0`'. It is not compulsory to write '`\0`' in string. The compiler automatically puts '`\0`' at the end of the character array or string. The characters of a string are stored in contiguous (neighbouring) memory locations. In the above example, the compiler automatically determines the size of the array. Table 8.1 shows the storing of string elements in contiguous memory locations.

Table 8.1 *Memory map of a string*

<i>I</i>	<i>N</i>	<i>D</i>	<i>I</i>	<i>A</i>	<i>\0</i>
5001	5002	5003	5004	5005	5006

While storing characters in a string array one should confirm if the array length is large enough to store the given string. Moreover, double quotes can be used to store a word or constants with type `char`. The length of the characters in "MAHARASHTRA" is 11 but in C, the character array should have a length of 12. One more character is needed at the end of string, which is called a null character.

The declaration is as follows:

```
char name[12]="MAHARASHTRA"
```

or

```
char name[12]={ 'M', 'A', 'H', 'A', 'R', 'A', 'S', 'H', 'T', 'R', 'A', '\0' };
```

C permits the storage of string to any length. However, if a string exceeds the limit of a character array, it will overwrite the data beyond the array.

8.2 DECLARATION AND INITIALIZATION OF STRING

The syntax for initialization of a string is as follows:

```
char name[ ]="INDIA";
```

The C compiler inserts the NULL (`\0`) character automatically at the end of the string. So initialization of the NULL character is not essential. Even if null is added at the end of string, compiler does not throw any error.

By initializing character arrays as per the following ways, the programmer can see the output:

(a) `char name[6]={ 'S', 'A', 'N', 'J', 'A', 'Y' };`

(b) `char name[7]={ 'S', 'A', 'N', 'J', 'A', 'Y' };`

In case (a) the output will not be 'SANJAY' as it contains some garbage value at the end of SANJAY. Array index/size in this example is initialized with [6], which is exactly equal to the number of

characters within the braces. The NULL character must be included at the end of string and hence, the array index/size must be [7] instead of [6] as given in statement (b).

- 8.1 Write a program to display the output when the account of NULL character is not considered.

```
void main()
{
    char name1[6]={'S','A','N','J','A','Y'};
    clrscr();
    printf("Name1 = %s",name1);
    getch();
}
```

OUTPUT:

Name1 = SANJAYabdn12

Explanation:

The output of the above program would be SANJAY followed by some garbage values. To get the correct result, the argument must be [7] instead of [6]. The output can be seen as given below after changing the size [7] in place of [6].

The output will be: SANJAY

The array size must be equal to the number of characters of the word + NULL character. In case the NULL character is not taken into account (statement (a)) the string followed by the first string (statement (b)) will be displayed. The output can be observed by executing the following program.

- 8.2 Write a program to display successive string in case first string is not terminated with NULL character.

```
void main()
{
    char name1[6]={'S','A','N','J','A','Y'};
    char name2[7]={'S','A','N','J','A','Y'};
    clrscr();
    printf("Name1 = %s",name1);
    printf("\nName2 = %s",name2);
}
```

OUTPUT:

Name1 = SANJAYSANJAY
Name2 = SANJAY

Explanation:

The NULL character has not been considered in the first statement. The compiler reads the second string immediately followed by the first string because the end of first string is not identified. Because of this, the second string is printed followed by the first string. Hence, the argument must include the account of the NULL character.

Simple string programs are as follows.

- 8.3 Write a program to print 'WELCOME' by using different syntax of initialization of array.

```
void main()
{
    char arr1[9]={'W','E','L',' ',' ','C','O','M','E','\0'};
    char arr2[9]="WELCOME";
    char arr3[9]= {'W'},{'E'},{'L'},{' '},{ 'C'},{ 'O'},{ 'M'},{ 'E'};
    clrscr();
    printf("\nArray1 = %s",arr1);
    printf("\nArray2 = %s",arr2);
    printf("\nArray3 = %s",arr3);
}
```

OUTPUT:

```
Array1 = WEL COME
Array2 = WELCOME
Array3 = WEL COME
```

Explanation:

The string elements can be initialized individually with enclosing single quote and curly braces. The curly braces are optional. While initializing individual elements, they must be separated by a comma. This is done in the first and third statements. Also it can be initialized with double-quotation marks and this is done in the second statement.

8.3 DISPLAY OF STRINGS WITH DIFFERENT FORMATS

The `printf()` function is used for displaying various data types. The `printf()` function with `%s` format is to be used for displaying a string on the screen. Various `printf()` formats are shown in Table 8.2, when `char text[15]="PRABHAKAR";`

Table 8.2 String formats with different precision

Sr. No.	Statement	Output
1.	<code>printf ("%s\n",text);</code>	PRABHAKAR
2.	<code>printf (".5s\n",text);</code>	PRABH
3.	<code>printf (".8s\n",text);</code>	PRABHAKA
4.	<code>printf (".15s\n",text);</code>	PRABHAKAR
5.	<code>Printf ("%10.4s\n",text);</code>	PRAB
6.	<code>Printf ("%11s",text);</code>	PRABHAKAR

- (i) The 1st statement displays the output 'PRABHAKAR'. The entire string is displayed with the first statement.
- (ii) We can also specify the precision with character string, which is to be displayed. The precision is (the number of characters to be displayed) provided after the decimal point. For instance in the 2nd statement in Table 8.2 the first five characters are displayed. Here, the integer value 5 on the right side of the decimal point specifies the five characters to be displayed.

- (iii) In the 3rd statement, the first eight characters are displayed.
- (iv) Statement number four displays the entire string.
- (v) The 5th statement with minus (–) sign (e.g. %–10.4s) displays the string with left justified.
- (vi) When the field length is less than the length of the string the entire string is printed. When it is greater than the length of the string, blank spaces are initially printed followed by the string. This effect can be seen in the 6th statement.
- (vii) When the number of characters to be displayed is specified as zero after decimal point nothing will be displayed.

A few examples are illustrated below giving the effects of various formats of strings.

- 8.4 Write a program to display the string 'PRABHAKAR' using various `printf()` format specifications.

```
void main()
{
    char text[15]="PRABHAKAR";
    clrscr();
    printf("%s\n",text);
    printf("%.5s\n",text);
    printf("%.8s\n",text);
    printf("%.15s\n",text);
    printf("%-10.4s\n",text);
    printf("\n%11s",text);
}
```

OUTPUT:

```
PRABHAKAR
PRABH
PRBHAKA
PRABHAKAR
PRAB
    PRABHAKAR
```

- 8.5 Use the `while` loop and print out the elements of the character array.

```
void main()
{
    char text[]="HAVE A NICE DAY";
    int i=0;
    clrscr();
    while(i<15)
    {
        printf("%c",text[i]);
        i++;
    }
}
```

OUTPUT:

```
HAVE A NICE DAY
```

Explanation:

In this program, `while` loop is used for printing the characters up to the length of 15; thereafter it terminates when variable '`i`' reaches 15.

- 8.6 Given below is an example in which the string elements are displayed. Use the `while` loop and print out the elements of the character array. Take the help of NULL (`'\0'`) character.

```
void main()
{
    char text[]="HAVE A NICE DAY";
    int i=0;
    clrscr();
    while(text[i]!='\0')
    {
        printf("%c",text[i]);
        i++;
    }
    getch();
}
```

OUTPUT:

HAVE A NICE DAY

Explanation:

It is not needed to give the array size as given in the previous programs. One can display the character string without knowing the length of the string. As we know that every character array always ends with NULL (`'\0'`) character, by using the NULL character in `while` loop we can write a program to display the string.

8.4 STRING STANDARD FUNCTIONS

C compiler supports a large number of string handling library functions. Table 8.3 provides the list of many frequently used functions and their description. The header file `string.h` is to be initialized whenever standard library function is used. In other words, the string handling functions are given in `string.h`.

Table 8.3 Standard C string library functions

Functions	Description
<code>strlen()</code>	Determines the length of a string.
<code>strcpy()</code>	Copies a string from the source to destination.
<code>strncpy()</code>	Copies characters of a string to another string up to the specified length.
<code>strcmp()</code>	Compares characters of two strings (function discriminates between small and capital letters)
<code>stricmp()</code>	Compares two strings (function does not discriminate between the small and capital letters).

(Continued)

Table 8.3 *Standard C string library functions (Continued)*

<i>Functions</i>	<i>Description</i>
<code>strncmp()</code>	Compares characters of two strings up to the specified length.
<code>strnicmp()</code>	Compares characters of two strings up to the specified length. Ignores case.
<code>strlwr()</code>	Converts uppercase characters of a string to lowercase.
<code>strupr()</code>	Converts lowercase characters of a string to uppercase.
<code>strdup()</code>	Duplicates a string.
<code>strchr()</code>	Determines the first occurrence of a given character in a string.
<code>strrchr()</code>	Determines the last occurrence of a given character in a string.
<code>strstr()</code>	Determines the first occurrence of a given string in another string.
<code>strcat()</code>	Appends source string to the destination string.
<code>strncat()</code>	Appends the source string to the destination string up to a specified length.
<code>strrev()</code>	Reversing all characters of a string.
<code>strset()</code>	Sets all characters of a string with a given argument or symbol.
<code>strnset()</code>	Sets specified numbers of characters of a string with a given argument or symbol.
<code>strspn()</code>	Finds up to what length two strings are identical.
<code>strpbrk()</code>	Searches the first occurrence of the character in a given string and then displays the string starting from that character.

We shall elaborate the above-cited functions by providing a few examples of each of them. Without using the standard functions, we can also write the programs. A few such programs are briefly described.

`strlen()` function

This function counts the number of characters in a given string. The syntax of the function is `strlen(char *S)`; A program in this regard is illustrated below.

➤ 8.7 Write a program to count the number of characters in a given string.

```
# include <string.h>
void main()
{
    char text[20];
    int len;
    clrscr();
    printf("Type Text Below.\n");
    gets(text);
    len=strlen(text);
    printf("Length of String =%d", len);
}
```


OUTPUT:

Type Text Below.

Hello

Length of String = 5

Explanation:

In the above program, `strlen()` function is called. Through the text array base address will be sent and this function returns the length of the string. It counts the string length without the NULL character.

- 8.8 Write a program to read a name through the keyboard. Determine the length of the string and find its equivalent ASCII codes.

```
# include <string.h>
void main()
{
    static char name[20];
    int i,l;
    clrscr();
    printf("Enter your name :");
    scanf("%s",name);
    l=strlen(name);
    printf("Your Name is %s &", name);
    printf("it contains %d characters.",l);
    printf("\nName & it's Ascii Equivalent.\n");
    printf("==== = === =====\n");
    for(i=0;i<l;i++)
        printf("\n %c\t\t%d",name[i],name[i]);
    getch();
}
```

OUTPUT:

Enter your name : SACHIN

Your Name is SACHIN & it contains 6 characters.

Name & it's Ascii Equivalent.

==== = === =====

S	83
A	65
C	67
H	72
I	73
N	78

Explanation:

In the above program, a string is entered. The string length is determined using the `strlen()` function. The `for` loop is used for displaying characters and their equivalent ASCII codes.

- 8.9 Write a program to remove the occurrences of 'The' word from the entered text.

```
# include <string.h>
void main()
{
    static char line[80], line2[80];
    int i, j=0, l;
    clrscr();
    printf("Enter Text Below.\n");
    gets(line);
    l=strlen(line);
    for(i=0; i<=l; i++)
    {
        if (i>=l-4 || l==3 && line[l-4]==' ' && line[l-3]=='t' &&
            line[l-2]=='h' && line[l-1]=='e')
            continue;
        if(line[i]=='t' && line[i+1]=='h' && line[i+2]=='e' &&
            line[i+3]==' ')
        {
            i+=2;
            continue;
        }
        else
        {
            line2[j]=line[i];
            j++;
        }
    }
    printf("\n Text with 'the'      : %s", line);
    printf("\n Text without 'the' : %s", line2);
    getch();
}
```

OUTPUT:

Enter Text Below.

Printf write data to the screen.

Text with 'the' : Printf write data to the screen.

Text without 'the' : Printf write data to screen

Explanation:

The first `if` condition is identifying the appearance of 'the' at the end of text as well as when the string contains only 'the'. If it is found the program terminates.

The second `if` condition in the above program, also identifies the appearance of 'the' word in the text. If it do not found the word 'the', elements of the first array are copied to the second array as it is. If the word 'the' is found program will continue to loop and never reach the statement `line2[j]=line[i];`.

- 8.10 Write a program to delete all the occurrences of vowels in a given text. Assume that the text length will be of one line.

```
void main()
{
    char line[80], line2[80];
    int i, j=0;
    clrscr();
    printf("Enter Text Below.\n");
    gets(line);
    for(i=0; i<80; i++)
    {
        if(line[i]=='a' || line[i]=='e' || line[i]=='i' || line[i]=='o'
           || line[i]=='u' || line[i]=='A' || line[i]=='E' || line[i]=='I' ||
           line[i]=='O' || line[i]=='U')
            continue;
        else
        {
            line2[j]=line[i];
            j++;
        }
    }
    printf("\n Text with Vowels : %s", line);
    printf("\n Text without Vowels : %s", line2);
}
```

OUTPUT:

```
Enter Text Below.
Have a nice day.
Text with Vowels : Have a nice day.
Text without Vowels : Hv nc dy.
```

Explanation:

The logic used here is the same as described in the previous program. The `if` statement checks the occurrence of vowels (a, e, i, o, u, A, E, I, O & U). If vowels are found loop is continued, otherwise elements are copied. Thus, vowels are skipped in the final output.

- 8.11 Write a program to find the no. of characters in a given string including and excluding spaces.

```
void main()
{
    char text[20];
    int i=0, len=0, ex=0;
    clrscr();
    printf("Enter Text Here :");
    gets(text);
    while(text[i]!='\0')
    {
        if(text[i]!=' ')
            ex++;
    }
}
```

```

        else
            len++;
        i++;
    }
    printf("\nLength of String Including Spaces. : %d", len+ex);
    printf("\nLength of String Excluding Spaces. : %d", len);
}

```

OUTPUT:

```

Enter Text Here :
Time is Money.
Length of String Including Spaces. : 13
Length of String Excluding Spaces. : 11

```

Explanation:

In the above program, some text is entered. The `if` statement within the `while` loop checks every element of string, and if space is observed the counter variable '`ex`' is incremented otherwise variable '`len`' is incremented. Thus, at the end the variable '`ex`' contains the total number of spaces in the string and the variable '`len`' contains the length of the string excluding spaces. The length including spaces is calculated by adding both the variables.

- 8.12 Write a program to display the length of the entered string and display it as per the output shown.

```

#include <string.h>
void main()
{
    int c,d;
    static char string[12];
    int ln;
    printf("Enter a String :");
    gets(string);
    ln=strlen(string);
    clrscr();
    printf("\n Length of given string :%d",ln);
    printf("\n Sequence of characters displayed on screen");
    printf("\n ===== == ===== ===== == =====");
    printf("\n\n\n");
    for(c=0;c<=ln-1;c++)
    {
        d=c+1;
        printf("\t%. *s\n",d,string);
    }
    for(c=ln-1;c>=0;c--)
    {
        d=c+1;

```

```
        printf("\t%. *s\n", d, string);
    }
}
```

OUTPUT:

```
Enter a String : HAPPY
Length of given string : 5
Sequence of characters displayed on screen
===== == ===== ===== == =====
H
HA
HAP
HAPP
HAPPY
HAPPY
HAPP
HAP
HA
H
```

Explanation:

In the above program, the string is entered and its length is calculated. The first `for` loop displays characters from left to right. It next adds one character in each successive line. The first line displays the first character, the second two characters and so on.

The second `for` loop displays characters from right to left. It removes the right-most character in each successive line.

Here, in the `printf()` the asterisk (*) used after decimal plays an important role in displaying the characters. It requires an integer argument. It displays only the given number of characters from the string.

strcpy() function

This function copies the contents of one string into another.

The syntax of `strcpy()` is `strcpy(char *s2, char *s1);`

where

`s1` is the source string; `s2` is the destination string and `s1` is copied to `s2`.

In many programs, we copy the contents of one string to other. Given below is an example which is based on the `strcpy()` function.

➤ 8.13 Write a program to copy the contents of one string to another by using `strcpy()`.

```
# include <string.h>
void main()
{
    char ori[20], dup[20];
    clrscr();
```

```

printf("Enter Your Name :");
gets(ori);
strcpy(dup,ori);
printf("Original String : %s",ori);
printf("\nDuplicate String : %s",dup);
}

```

OUTPUT:

```

Enter Your Name : SACHIN
Original String : SACHIN
Duplicate String : SACHIN

```

Explanation:

In the above example, we have declared two arrays namely `ori[20]` and `dup[20]`. The function `strcpy()` copies characters of `ori[]` to `dup[]`. The characters are copied one by one from source string (`ori [20]`) to destination string (`dup[20]`).

Program without `strcpy()` function

- 8.14 Write a program to copy the contents of one string to another, without the `strcpy()` function.

```

# include <string.h>
void main()
{
    char ori[20],dup[20];
    int i;
    clrscr();
    printf("Enter Your Name :");
    gets(ori);
    for(i=0;i<20;i++)
        dup[i]=ori[i];
    printf("Original String : %s",ori);
    printf("\nDuplicate String : %s",dup);
}

```

OUTPUT:

```

Enter Your Name : SACHIN
Original String : SACHIN
Duplicate String : SACHIN

```

Explanation:

In the above program, we also have declared two arrays namely `ori[20]` and `dup[20]`. Instead of using the `strcpy()` function, by using the `for` loop elements of source array are copied into the destination array one by one.

`strncpy()` function

`strncpy()` function performs the same task as `strcpy()`. The only difference between them is that the former function copies specified length of characters from the source to destination string, whereas the latter function copies the whole source string to destination string. The syntax of the function is

```
strncpy(char *destination ,char *source, int n);
```

where n is the argument.

A simple example is illustrated below.

- 8.15 Write a program to copy source string to destination string up to a specified length. Length is to be entered through the keyboard.

```
# include <string.h>
void main()
{
    char str1[15], str2[15];
    int n;
    clrscr();
    printf("Enter Source String :");
    gets(str1);
    printf("Enter Destination String :");
    gets(str2);
    printf("Enter Number of Characters to Replace in Destination
String :");
    scanf("%d",&n);
    strncpy(str2,str1,n);
    printf("Source String :%s",str1);
    printf("\nDestination String :%s",str2);
}
```

OUTPUT:

```
Enter Source String      : wonderful
Enter Destination String : beautiful
Enter Number of Characters to Replace in Destination String : 6
Source String           : wonderful
Destination String      : wonderful
```

Explanation:

In the above program, two strings are read from terminal. The number of characters to replace in the destination string from source string is also entered. After obtaining these three arguments the `strncpy()` function replaces the destination string with the number of characters (argument). The source string characters are 'wonderful' and the destination 'beautiful' before the use of `strncpy()`. After execution, the first six characters of destination string ('beauti') are replaced with first six characters of source string ('wonder'). The output of the program is as shown above.

stricmp() function

The syntax of the function is `strcmp(char *s1, char *s2);`

This function compares two strings. The characters of the strings may be in lowercase or uppercase; the function does not discriminate between them. That is, this function compares two strings without case. If the strings are the same it returns to zero otherwise non-zero value.

- 8.16 Write a program to compare the two strings using the `strcmp()` function. If strings are identical display 'The Two Strings are Identical' otherwise 'The Strings are Different'.

Note: `strcmp()` function compares two strings character by character and returns 0 if the strings are identical otherwise non-zero value. This function does not discriminate between small and capital letters.

```
void main()
{
    char sr[10],tar[10];
    int diff;
    clrscr();
    printf("Enter String(1) : ");
    gets(sr);
    printf("Enter String(2) : ");
    gets(tar);
    diff=strcmp(sr,tar);
    if(diff==0)
        puts("The Two Strings are Identical.");
    else
        puts("The Two Strings are Different");
    getch();
}
```

OUTPUT:

```
Enter String(1) : HELLO
Enter String(2) : hello
The Two Strings are Identical.
```

Explanation:

In the above program, two strings are entered. Both the strings are compared using the `strcmp()` function. If both the strings are identical it returns 0 otherwise non-zero value. The returned value of the `strcmp()` function is assigned to variable 'diff'. The `if` condition checks the value of variable 'diff' and respective message is displayed.

- 8.17 Write a program to perform the following:
1. Display the question "What is the Unit of Traffic?"
 2. Accept the answer.
 3. If the answer is wrong (Other than Earlang) display "Try again!" & continues to answer.
 4. Otherwise, if it is correct "Earlang" display the message "Answer is correct".
 5. If the user gives correct answer in first two attempts the program will terminate.
 6. If the user fails to provide correct answer in three attempts the program it self gives the answer.


```
# include <process.h>
void main()
{
    char ans[8];
    int i;
    clrscr();
    for(i=1;i<=3;i++)
    {
        printf("\nWhat is the Unit of Traffic?");
        scanf("%s",ans);
        fflush(stdin);
        if(stricmp(ans,"Earlang")==0)
        {
            printf("\nAnswer is Correct.");
            exit(1);
        }
        else
            if(i<3)
                printf("\n Try Again !\n");
    }
    clrscr();
    printf("\nunit of Traffic is Earlang.");
}
```

OUTPUT:

```
What is the Unit of Traffic ? Earlan
Try Again !
What is the Unit of Traffic ? Earlam
Try Again !
What is the Unit of Traffic ? Earlang
Answer is Correct.
```

Explanation:

In the above program `stricmp()` function is used for comparing character array `ans[]` and 'Earlang'. If function returns 0 (zero) the message displayed will be 'Answer is Correct'. In case, the answer is wrong the message displayed will be 'Try again!'. Three attempts are provided using the `for` loop for answering the question. The `fflush()` function is used for clearing the buffer.

strcmp() function

`strcmp()`: One can also use `strcmp()` function instead of `stricmp()`. The only difference between them is the former function discriminates between small and capital letters whereas the latter does not. The output of the above program after `strcmp()` in place of `stricmp()` will be as follows:

```
Enter String(1) : HELLO
Enter String(2) : hello
```

The Two Strings are Different.

The above function compares two strings for finding whether they are the same or different. Characters of these strings are compared one by one. In case of a mismatch the function returns to non-zero value, otherwise it returns zero, i.e. when the two strings are the same `strcmp()` returns the value zero. If they are different it returns the numeric difference between the ASCII values of non-matching characters.

`strncmp()` function

Comparison of two strings can be made up to a certain specified length. The function used for this is `strncmp()`. This function is the same as `strcmp()` but it compares the character of the string to the specified length. The syntax of this function is as follows:

```
strncmp(char *source, char *target, int argument);
```

where the argument is the number of characters up to which the comparison is to be made.

➤ 8.18 Write a program to compare two strings up to specified length.

```
# include <string.h>
void main()
{
    char sr[10],tar[10];
    int n,diff;
    clrscr();
    printf("Enter String(1) : ");
    gets(sr);
    printf("Enter String(2) : ");
    gets(tar);
    printf("\nEnter Length up to which comparison is to be made ");
    scanf("%d",&n);
    diff=strncmp(sr,tar,n);
    if(diff==0)
        printf("The Two Strings are Identical up to %d characters.",n);
    else
        puts("The Two Strings are Different.");
    getch();
}
```

OUTPUT:

```
Enter String(1) : HELLO
Enter String(2) : HE MAN
Enter Length up to which comparison is to be made : 2
The Two Strings are Identical up to 2 characters.
```

Explanation:

One can also use `strnicmp()` function instead of `strncmp()`. The only difference between them is that, the former function discriminates between small and capital letters whereas the latter does not. The output of the above program after `strnicmp()` in place of `strncmp()` will be as follows:

```
Enter String(1) : HELLO
Enter String(2) : HE MAN
```

The two strings are identical up to two characters.

Similarly, a program without using `strcmp()` can be developed which is as given below.

- 8.19 Write a program to enter the two strings and compare them without using any standard function. Determine whether the strings are identical or not. Also display the number of position where the characters are different.

```
# include <string.h>
void main()
{
    static char sr[10],tar[10];
    int diff=0,i;
    clrscr();
    printf("Enter String(1) : ");
    gets(sr);
    printf("Enter String(2) : ");
    gets(tar);
    for(i=0;i<10;i++)
    {
        if(sr[i]==tar[i])
            continue;
        else
        {
            printf("%c %c\n", sr[i], tar[i]);
            diff++;
        }
    }
    if(strlen(sr)==strlen(tar) && diff==0)
        puts("\nThe Two Strings are Identical");
    else
        printf("\nThe Two Strings are Different at %d places.",diff);
    getch();
}
```

OUTPUT:

```
Enter String(1) : BEST LUCK
Enter String(2) : GOOD LUCK
G B
O E
O S
D T
```

The Two Strings are Different at 4 places.

Explanation:

In the above program, two strings up to 10 characters can be entered. The `if` condition within the `for` loop checks corresponding characters of both the strings. If they are identical the

loop is continued. Otherwise, counter variable 'diff' is incremented and different characters of two strings are displayed. The last if condition checks the variable 'diff' and displays respective messages.

strlwr() function

This function can be used to convert any string to lowercase. When you are passing any upper case string to this function it converts into lowercase. The standard syntax of `strlwr()` is as follows:

```
strlwr(char *string);
```

➤ 8.20 Write a program to convert the uppercase string to lowercase using `strlwr()`.

```
# include <string.h>
void main()
{
    char upper[15];
    clrscr();
    printf("\nEnter a string in Upper case :");
    gets(upper);
    printf("After strlwr() : %s", strlwr(upper));
}
```

OUTPUT:

```
Enter a string in Upper case : ABCDEFG
After strlwr()                : abcdefg
```

Explanation:

In this program string is entered in capital letters. The string is passed to the function `strlwr()`. This function converts the string to lower case.

strupr() function

This function is the same as `strlwr()` but the difference is that `strupr()` converts lowercase strings to uppercase strings. The syntax of this function is `strupr(char *string)`;

➤ 8.21 Write a program to convert the lowercase string to upper case using `strupr()`.

```
# include <string.h>
void main()
{
    char upper[15];
    clrscr();
    printf("\nEnter a string in Lower Case :");
    gets(upper);
```

```
printf("Afterstrup() : %s",strup(upper));
}
```

OUTPUT:

```
Enter a string in Lower Case : abcdefg
Afterstrup()                : ABCDEFG
```

Explanation:

In this program a string is entered. The string is passed to the function `strup()`. This function converts the string to uppercase.

strup() function

This function is used for duplicating a given string at the allocated memory which is pointed by the pointer variable. The syntax of this function is `text2=strup(text1)`.

Where `text1` is a string and `text2` is a pointer.

- 8.22 Write a program to enter the string and get its duplicate. Use the `strup()` function.

```
#include <string.h>
void main()
{
    char text1[20],*text2;
    clrscr();
    printf("Enter Text :");
    gets(text1);
    text2=strup(text1);
    /* pointer *text2 is initialized to the address of text1 through
      strup() function. */
    printf("Original String = %s\nDuplicate String = %s",text1,text2);
}
```

OUTPUT:

```
Enter Text : Today is a Good Day.
Original String = Today is a Good Day.
Duplicate String = Today is a Good Day.
```

Explanation:

In the above program character array `text1[]` and pointer variable `*text2` are declared. A string is entered in character array `text1[]`. The `strup()` function copies the contents of `text1[]` array to pointer variable `*text2`. The `printf()` function displays the contents of both the variables which are the same.

strchr() function

This function returns the pointer position to the first occurrence of the character in the given string. The format of this function is `chp=strchr(string, ch);`

Where `string` is a character array, `ch` is a character variable and `chp` is a pointer which collects address returned by the `strchr()` function. The syntax of this function is `strchr(char*string, char character);`

- 8.23 Write a program to find first occurrence of a given character in a given string. Use the `strchr()` function.

```
# include <string.h>
void main()
{
    char string[30],ch,*chp;
    clrscr();
    printf("Enter Text Below :");
    gets(string);
    printf("\nCharacter to find :");
    ch=getchar();
    /* returns a pointer to the first occurrence of the given
    character in*/
    /* pointer chp, if given character is not found strchr()
    returns null.*/
    chp=strchr(string,ch);
    if(chp)
        printf("Character %c found in string.",ch);
    else
        printf("Character %c not found in string.",ch);
}
```

OUTPUT:

```
Enter Text Below: Hello Beginners.
Character to find : r
Character r found in string.
```

OR

- 8.24 Write a program to find the first occurrence of a given character in a given string. Find the memory location where the character occurs. Use the `strchr()` function.

```
# include <string.h>
void main()
{
    char line1[30],line2,*chp;
    int i;
    clrscr();
    puts("Enter Text :");
    gets(line1);
    puts("Enter Character to find from the text :");
    line2=getche();
```

```

for(i=0;i<strlen(line1);i++)
printf("\n%c %u", line1[i],&line1[i]);
chp=strchr(line1, line2);
if(chp)
{
    printf ("\nAddress of first %c returned by strchr() is %u
", line2, chp);
}
else
printf("%c character is not present in Given String", line2);
}

```

OUTPUT:

```

Enter Text : HELLO
Enter Character to find from the text : L
H 4032
E 4033
L 4034
L 4035
O 4036
Address of first L returned by strchr() is 4034.

```

Explanation:

This function finds the memory location where, the first occurrence of a given character is found in a given string. When it finds the character the program terminates and the `strchr()` function provides the address of that character. The `strchr()` function returns the memory address of the first occurred character, i.e. it returns the memory address, which is collected by the pointer variable.

Note: In place of `strchr()` one can use `strrchr()`. The difference between them is that `strchr()` searches the occurrence of a character from the beginning of the string whereas `strrchr()` searches the occurrence of a character from the end (reverse).

strstr() function

This function finds the second string in the first string. It returns the pointer location from where the second string starts in the first string. In case the first occurrence in the string is not observed, the function returns a NULL character.

The syntax of this function is `strstr (char *string1, char *string2);`

➤ 8.25 Write a program using `strstr()` function for occurrence of second string in the first string.

```

# include <string.h>
void main()

```

```

{
    char line1[30], line2[30], *chp;
    clrscr();
    puts("Enter Line1 :");
    gets(line1);
    puts("Enter Line2 :");
    gets(line2);
    chp=strstr(line1, line2);
    if(chp)
        printf("\'%s\' String is present in Given String.", line2);
    else
        printf("\'%s\' String is not present in Given String.", line2);
}

```

OUTPUT:

```

Enter Line1 : INDIA IS MY COUNTRY.
Enter Line2 : INDIA
'INDIA' String is present in Given String.

```

strcat() function

This function appends the target string to the source string. Concatenation of two strings would be done using this function. The syntax of this function is `strcat(char *text1, char *text2);`

- 8.26 Write a program to append the second string at the end of the first string using the `strcat()` function.

```

# include <string.h>
void main()
{
    char text1[30], text2[10];
    puts("Enter Text1 :");
    gets(text1);
    puts("Enter Text2 :");
    gets(text2);
    strcat(text1, " ");
    strcat(text1, text2);
    clrscr();
    printf("%s\n", text1);
}

```

OUTPUT:

```

Enter Text1 : I am
Enter Text2 : an Indian
I am an Indian

```


Explanation:

In the above example, two strings are entered in the character array `text1[]` and `text2[]`. The `strcat()` function concatenates both the strings, i.e. the second string is appended in the first string. The `printf()` function displays the contents of the `text1[]` which is the concatenation of the two strings.

- 8.27 Write a program to concatenate two strings without the use of the standard function.

```
# include <string.h>
void main()
{
    char name[50], fname[15], sname[15], lname[15];
    int i, j, k;
    clrscr();
    printf("First Name :");
    gets(fname);
    printf("Second Name :");
    gets(sname);
    printf("Last Name :");
    gets(lname);
    for(i=0; fname[i]!='\0'; i++)
        name[i]=fname[i];
    name[i]=' ';
    for(j=0; sname[j]!='\0'; j++)
        name[i+j+1]=sname[j];
    name[i+j+1]=' ';
    for(k=0; lname[k]!='\0'; k++)
        name[i+j+k+2]=lname[k];
    name[i+j+k+2]='\0';
    printf("\n\n");
    printf("Complete Name After Concatenation.\n");
    printf("%s", name);
    getch();
}
```

OUTPUT:

```
First Name : MOHAN
Second Name : KARAMCHAND
Last Name : GANDHI
Complete Name After Concatenation.
MOHAN KARMCHAND GANDHI
```

Explanation:

In the above program, three strings are entered. The first `for` loop copies string `fname[]` to `name[]` array using simple assignment. The statement following the `for` loop adds space after the string. The next two `for` loops follow the same procedure to concatenate arrays `sname[]` and `lname[]` in `name[]`. Thus, we get in `name[]` concatenation of three strings.

strncat() function

This function is the same as that of `strcat()`. The difference between them is that, the former does the concatenation of the strings with another up to the specified length. The syntax of this function is `strncat (text1, text2, n)`; where n is the number of characters to append.

- 8.28 Write a program to append the second string with specified (n) number of characters at the end of the first string using the `strncat()` function.

```
# include <string.h>
void main()
{
    char text1[30], text2[10],n;
    puts("Enter Text1 :");
    gets(text1);
    puts("Enter Text2 :");
    gets(text2);
    printf("Enter Number of Characters to Add :");
    gets(n);
    strcat(text1," ");
    strncat(text1,text2,n);
    clrscr();
    printf("%s\n", text1);
}
```

OUTPUT:

```
Enter Text1 : MAY I
Enter Text2 : COME IN ?
Enter Number of Characters to Add : 4
MAY I COME
```

Explanation:

In this program, two strings are entered. The number of characters of the second string to append in the first string is also entered. The `strncat()` function uses three arguments viz. `Text1[]`, `text2[]` and ' n ', where ' n ' characters of `text2[]` are appended in the `text1[]` string. In this program, two strings 'MAY I', 'COME IN' and $n=4$ are entered. The `strncat()` function returns 'MAI I COME'.

strrev() function

This function simply reverses the given string. The syntax of this function is `strrev(char *s)`;

- 8.29/8.30 Write a program to display the reverse of the given string.

```
# include <string.h>
void main()
{
```

```

char text[15];
puts("Enter String");
gets(text);
puts("Reverse String");
puts(strrev(text));
}

```

OUTPUT:

```

Enter String
ABCDEFGHI
Reverse String
IHGFEDCBA

```

OR

```

# include <string.h>
void main()
{
    char text[15];
    int i=0;
    clrscr();
    printf("Enter String :-");
    gets(text);
    while (text[i]!='\0')
    {
        printf("\n %c is stored at location %u",text[i],&text[i]);
        i++;
    }
    strrev(text);
    printf("\nReverse String :-");
    printf("%s",text);
    i=0;
    while (text[i]!='\0')
    {
        printf ("\n %c is stored at location %u",text[i],&text[i]);
        i++;
    }
}

```

OUTPUT:

```

Enter String :- ABC
A is stored at location 4054
B is stored at location 4055
C is stored at location 4056
Reverse String :- CBA
C is stored at location 4054
B is stored at location 4055
A is stored at location 4056

```

Explanation:

In the above programs, string is entered and passed into the `strrev()` function. On the execution of the function, the given string appears in the reverse order. The `strrev()` function physically changes the sequence of characters in the reverse order. The output of the second program shows the sequence of characters and their locations.

strset() function

This function replaces every character of a string with the symbol given by the programmer, i.e. the elements of the strings are replaced with the arguments given by the programmer. The syntax of the function is `strset(char *string, char symbol)`

- 8.31 Write a program to replace (set) the given string with the given symbol. Use the `strset()` function.

```
# include <string.h>
void main()
{
    char string[15];
    char symbol;
    clrscr();
    puts("Enter String :");
    gets(string);
    puts("Enter Symbol for Replacement:");
    scanf("%c",&symbol);
    printf("Before strset() : %s\n", string);
    strset(string, symbol);
    printf("After strset() : %s\n", string);
}
```

OUTPUT:

```
Enter String : LEARN C
Enter Symbol for Replacement: Y
Before strset() : LEARN C
After strset() : YYYYYYY
```

Explanation:

The `strset()` function requires two arguments. First one is the string and another character by which the string is to be replaced. Both these arguments are to be entered when the program executes. The `strset()` function replaces every character of the first string with the given character/symbol, i.e. every character of the string replaces by the entered character.

strnset() function

This function is the same as that of `strset()`. Here, the specified length is provided. The syntax of this function is `strnset(char *string, char symbol, int n);` where *n* is the number of characters to replace.

- 8.32 Write a program to replace (set) the given string with the given symbol for the given number of arguments. Use the `strnset()` function.

```
# include <string.h>
void main()
{
    char string[15];
    char symbol;
    int n;
    clrscr();
    puts("Enter String :");
    gets(string);
    puts("Enter Symbol for Replacement:");
    scanf("%c",&symbol);
    puts("How many String Character to be replaced.");
    scanf("%d",&n);
    printf("Before strnset() : %s\n", string);
    strnset(string, symbol,n);
    printf("After strnset() : %s\n", string);
}
```

OUTPUT:

```
Enter String : ABCDEFGHIJ
Enter Symbol for Replacement: +
How many String Characters to be replaced. 4
Before strnset() : ABCDEFGHIJ
After strnset() : ++++EFGHIJ
```

Explanation:

This program is the same as that of the previous one. The only difference is that instead of replacing all characters of the string, only a specified number of characters are to be replaced. Here, the number entered is 4. Hence, only the first four characters are replaced by the given symbol. The replacing process starts from the first character of the string.

`strspn()` function

This function returns the position of the string from where the source array is not matching the target one. The syntax of this function is `strspn (char *string1, char *string2)`

- 8.33 Write a program to enter two strings. Indicate after how many characters the source string is not matching the target string.

```
# include <string.h>
void main()
```

```

{
    char stra[10],strb[10];
    int length;
    clrscr();
    printf("First String :");
    gets(stra);
    printf("Second String :");
    gets(strb);
    length=strspn(stra,strb);
    printf("After %d Characters there is no match.\n",length);
}

```

OUTPUT:

```

First String : GOOD MORNING
Second String : GOOD BYE
After 5 Characters there is no match.

```

Explanation:

In this program, two strings are entered. Both the strings are passed to the function `strspn()`. The function searches the second string in the first string. It searches from the first character of the string. If there is a match from the beginning of the string, the function returns the number of characters that are the same.

This function returns 0, when the second string mismatches with the first from the beginning. For example, assume the first string is 'BOMBAY' and the second 'TROMBAY'. On the application of this function in the above case, the function returns 0 and message displayed will be 'After 0 characters there is no match'.

strpbrk() function

This function searches the first occurrence of the character in a given string, and then it displays the string starting from that character. This function returns the pointer position to the first occurrence of the required character in `text2[2]`. The syntax of this function is `strpbrk(char *text1, char text2);`

➤ 8.34 Write a program to print the given string from the first occurrence of the given character.

```

# include <string.h>
void main()
{
    char *ptr;
    char text1[20],text2[2];
    clrscr();
    printf("Enter String :");
    gets(text1);
    printf("Enter Character :");
    gets(text2);
    ptr=strpbrk (text1,text2[0]);
}

```

```
puts("String from given Character");
printf(ptr);
}
```

OUTPUT:

```
Enter String : INDIA IS GREAT
Enter Character : G
String from given Character : GREAT
```

Explanation:

In the above program, two strings and character pointer are declared. The strings are entered. Both the strings are used as arguments with the function `strpbrk()`. This function finds first occurrence of required character in second string and returns that address which is assigned to the character pointer `*ptr`. The pointer `ptr` displays the rest string.

Here, the first string is 'INDIA IS GREAT' and the second string is 'G'. The 'G' occurs at the beginning of the third word. Hence, on the execution of the program the string from 'G' onwards is displayed. The output is only 'GREAT'.

8.5 STRING CONVERSION FUNCTIONS

Sr. No.	Function Name	Description
1	<code>double atof(const char *s);</code>	Converts the given string to double.
2	<code>int atoi(const char *s);</code>	Converts the given string to int
3	<code>long atoi(const char *s);</code>	Converts the given string to long.
4	<code>double strtod(char *s, char **endptr);</code>	Separates char and float data from the given string.
5	<code>long strtol(char *s, char **endptr, int radix);</code>	Separates char and long int data from the given string.

➤ 8.35 Write a program to demonstrate the use of the `atof()` function.

```
# include <stdlib.h>
void main()
{
    double d;
    d=atof("99.1254");
    clrscr();
    printf("%g",d);
}
```

OUTPUT:

```
99.1254
```

Explanation:

In the above program, the string '99.1254' is passed as an argument to the function `atof()` which converts string to double. The converted number is stored in the variable `d` and is displayed.

- 8.36 Write a program to demonstrate the use of `atoi()` function.

```
# include <stdlib.h>
void main()
{
    int i;
    i=atoi("99.11");
    clrscr();
    printf("%d",i);
}
```

OUTPUT:

99

Explanation:

The above program is the same as the last one. Here, the entered string "99.11" is converted to an integer, i.e. 99.

- 8.37 Write a program to demonstrate the use of `strtod()`.

```
# include <stdlib.h>
void main()
{
    const char *string = "12.2% is rate of interest";
    char *stp;
    double d;
    clrscr();
    d=strtod(string,&stp);
    printf("%g",d);
    printf("\n%s",stp);
}
```

OUTPUT:

12.2
% is rate of interest

Explanation:

In the above program, the string contains 12.2 a float number. The function `strtod()` separates float, and strings are stored in separate variables. The same is displayed.

8.6 MEMORY FUNCTIONS

Sr. No	Function	Description
1	<code>memcpy()</code>	Copies <i>n</i> number of characters from one string to another.
2	<code>memmove()</code>	Moves a specified range of <code>char</code> from one place to another.
3	<code>memchr()</code>	Searches for the first occurrence of the given character.
4	<code>memcmp()</code>	Compares the contents of the memory.

- 8.38 Write a program to demonstrate the use of `memcpy()` function.

```
void main()
{
    char *str = "Mukesh and Kamlesh";
    char stp[20];
    clrscr();
    memcpy(stp, str, 20);
    printf("%s", stp);
}
```

OUTPUT:

Mukesh and Kamlesh

Explanation:

In this program, the contents of variable `str` are copied to `stp` using the `memcpy()` function. The function requires three arguments, i.e. destination, source and size.

- 8.39 Write a program to demonstrate the use of `memmove()`.

```
void main()
{
    char str[] = "Good Very Good";
    clrscr();
    printf("\n before : %s", str);
    memmove(str, &str[5], 10);
    printf("\nAfter : %s", str);
}
```

OUTPUT:

before : Good Very Good
After : Very Good Good

Explanation:

In this program, the function `memmove()` moves a specified range of **char** to the given location. In the output, you can observe how the strings are shifted to the beginning of the string.

- 8.40 Write a program to demonstrate the use of `memcmp()`.

```
void main()
{
    char sf[] = "a";
    char ss[] = "A";
    clrscr();
```

```
printf("%d",memcmp(sf,ss,2));
}
```

OUTPUT:

32

Explanation:

In the above, two strings are compared up to the specified range. Their ASCII difference is returned.

- 8.41 Write a program to demonstrate the use of `memchr()`.

```
void main()
{
    char *sf= "C IS EASY";
    clrscr();
    printf("%s",memchr(sf,'E',10));
}
```

OUTPUT:

EASY

Explanation:

In the above program, the function `memchr()` searches for the first occurrence of 'E'. After getting the desired result, the remaining string is displayed.

8.7 APPLICATIONS OF STRINGS

- 8.42 Write a program to count a character that appears in a given text for a number of times. Use the `while` loop.

```
void main()
{
    char text[20];
    char find;
    int i=0,count=0;
    clrscr();
    printf("Type Text Below.\n");
    gets(text);
    printf("Type a character to count :");
    find=getche();
    while(text[i]!='\0')
    {
        if(text[i]==find)
            count++;
    }
}
```

```

        i++;
    }
    printf("\nCharacter (%c) found in Given String = %d
        Times.", find, count);
}

```

OUTPUT:

Type Text Below.

Programming

Type a character to count : m

Character (m) found in Given String = 2 Times.

Explanation:

In the above program, a string is entered. A single character, which is to be searched in the string, is also entered. The `if` condition in the `while` loop checks every character of the string with the single character. If there is a match counter variable '`count`' gets incremented. After the complete execution of the `while` loop the counter displays the number of times the character found in the string.

- 8.43 Write a program to count '`m`' characters that appear in a given string without using any function. Use the `for` loop.

```

void main()
{
    char text[25];
    int i, count=0;
    clrscr();
    printf("\nEnter the string:");
    for(i=0; i<25; ++i)
    {
        scanf("%c", &text[i]);
        if(text[i]=='\n')
            break;
        else
            if(text[i]=='m')
                ++count;
    }
    printf("Character 'm' Found in Text=%d times.\n", count);
    getch();
}

```

OUTPUT:

Enter the string:

Programming is a skill.

Character 'm' Found in Text=2 times.

Explanation:

The logic of the program is the same as that of the previous one. Here, in this program the character that is to be searched is '`m`' which is a default character. The first `if` statement

within the `for` loop terminates the loop when the user presses the 'enter' key. The second `if` statement checks every character of the entered string with 'm'. If there is a match counter variable, 'count' is incremented. Thus, at last count variable gives the number of times 'm' present in the string.

- 8.44 Write a program to count the following characters that appear in a string without using any functions.

```
1. 'm'
2. 'r'
3. 'o'
void main()
{
    char text[25]="Programming is good habit";
    int i,m=0,o=0,r=0;
    clrscr();
    for(i=0;i<25;++i)
    {
        if(text[i]=='m')
            ++m;
        if(text[i]=='r')
            ++r;
        if(text[i]=='o')
            ++o;
    }
    printf("\nCharacter 'm' Found in Text=%d times.\n",m);
    printf("\nCharacter 'r' Found in Text=%d times.\n",r);
    printf("\nCharacter 'o' Found in Text=%d times.\n",o);
    getch();
}
```

OUTPUT:

```
Character 'm' Found in Text=2 times.
Character 'r' Found in Text=2 times.
Character 'o' Found in Text=3 times.
```

Explanation:

The logic of the program is the same as that of the previous one. Here, in this program the characters that are to be searched are 'm', 'r' and 'o'. The `if` statements within the `for` loops increment with respective counter variables when there is a match of these characters. Thus, after the execution of the `for` loop the three counter variables 'm', 'r' and 'o' are printed.

- 8.45 Write a program to copy the contents of one string to another string without using the function.

```
void main()
{
    char ori[20],dup[20];
    int i;
```

```

clrscr();
printf("Enter Your Name :");
gets(ori);
for(i=0;ori[i]!='\0';i++)
dup[i]=ori[i];
dup[i]='\0';
printf("Original String : %s",ori);
printf("\nDuplicate String : %s",dup);
}

```

OUTPUT:

```

Enter Your Name : SACHIN
Original String : SACHIN
Duplicate String : SACHIN

```

Explanation:

In the above program, two character arrays are declared. The source string is entered. Using the `for` loop and assignment operator each character of the source array (`ori[]`) is assigned to the target array `dup[]`. After the execution of the `for` loop, NULL character is appended in the target string to mark the end of the string. Using the `printf()` function both the strings are displayed.

- 8.46 Write a program to know whether the entered character string is palindrome or not. (Palindrome word reads the same from left to right and right to left.)

(Ex. DAD, ABBA, MUM)

```

# include <string.h>
# include <process.h>
void main()
{
    char str[10];
    int i=0,j,test;
    clrscr();
    printf("Enter the word :");
    scanf("%s",str);
    j=strlen(str)-1;
    while(i<=j)
    {
        if (str[i]==str[j])
            test=1;
        else
        {
            test=0;
            break;
        }
        i++;
    }
}

```

```

        j--;
    }
    if(test==1)
        printf("Word is palindrome.\n");
    else
        printf("\n Word is not Palindrome.\n");
}

```

OUTPUT:

```

Enter the word : ABBA
Word is palindrome.

```

Explanation:

In the above program, a string is entered that is to be tested for 'palindrome'. The string length is calculated and assigned to variable 'j'. The value of 'j' is less by one with the original string length because the array elements are counted from zero (0).

The **if** statement within the **while** loop checks the first and last characters of the string for equality. Counter variables 'i' and 'j' denote the first and last characters, respectively. To get the successive characters from both the ends, variable 'i' is incremented and 'j' is decremented. Till there is a match variable 'test' is 1 and the loop continues, otherwise test is set to zero (0) and the **break** statement terminates the loop.

- 8.47 Write a program to compare the strings using the **strcmp()** function and display their ASCII difference. Initialize the strings and copy some names of the cities to the variables.

```

#include <string.h>
void main()
{
    char a1[15],a2[15],a3[15],a4[15],a5[15],a6[15];
    int c1,c2,c3;
    strcpy(a1,"NAGPUR");
    strcpy(a2,"NAGPUR");
    strcpy(a3,"PANDHARPUR");
    strcpy(a4,"KOLHAPUR");
    strcpy(a5,"AURANGABAD");
    strcpy(a6,"HYDERABAD");
    clrscr();
    c1=strcmp(a1,a2);
    c2=strcmp(a3,a4);
    c3=strcmp(a5,a6);
    printf("\nAscii Difference between two strings\n");
    printf("Difference between (%s %s)=%d\n",a1,a2,c1);
    printf("Difference between (%s %s)=%d\n",a3,a4,c2);
    printf("Difference between (%s %s)=%d\n",a5,a6,c3);
    getch();
}

```

OUTPUT:

```
Difference between (NAGPUR NAGPUR)= 0
Difference between (PANDHARPUR KOLHAPUR)= 5
Difference between (AURANGABAD HYDERABAD)=-7
```

Explanation:

In the above program, five character arrays are declared. Using the `strcpy()` function the names of cities are copied to arrays. Using the `strcmp()` function, strings are compared. The `strcmp()` returns the ASCII difference of two strings. The ASCII values of the first characters of two strings are taken into account for comparison. Rest of the elements are not considered for ASCII difference. The ASCII value of the first character of the first string is subtracted from the ASCII value of the first character of second string. Table 8.4 illustrates the calculation.

Table 8.4 ASCII difference

ASCII Value	ASCII Value	Difference
78 (N)	78 (N)	0
80 (P)	75 (K)	5
65 (A)	72 (H)	-7

- 8.48/8.49 Write a program to enter names of cities and display all the entered names alphabetically.

```
void main()
{
    char city[5][20];
    int i,j;
    clrscr();
    printf("Enter Names of Cities.\n\n");
    for(i=0;i<5;i++)
        scanf("%s",city[i]);
    printf("Sorted List of Cities.\n\n");
    for(i=65;i<=122;i++)
    {
        for(j=0;j<5;j++)
        {
            if(city[j][0]==i)
                printf("\n%s",city[j]);
        }
    }
}
```

OUTPUT:

```
Enter Names of Cities.
MUMBAI
```

```

NANDED
BANGLORE
KANPUR
INDORE
Sorted List of cities.
BANGLORE
INDORE
KANPUR
MUMBAI
NANDED

```

Explanation:

In the above program, the first `for` loop is used for entering the names of cities. The city name can be entered in either upper or lowercase. Hence, the second `for` loop is initialized from 65 to 122 where 65 is the ASCII value of 'A' and 122 ASCII value of 'z'. The `if` statement within the third `for` loop makes the comparison. If there is a match, the city name will be displayed, otherwise the loop continues. Thus, the elements of `city[][]` array are displayed in a sorted order.

OR

```

# include <string.h>
void main()
{
    char city[5][20],temp[20];
    int i,j;
    clrscr();
    printf("Enter Names of Cities\n");
    for(i=0;i<5;i++)
        scanf("%s",city[i]);
    printf("\nSorted List of Cities");
    for(i=1;i<5;i++)
    {
        for(j=1;j<5;j++)
        {
            if(strcmp(city[j-1],city[j])>0)
            {
                strcpy(temp,city[j-1]);
                strcpy(city[j-1],city[j]);
                strcpy(city[j],temp);
            }
        }
    }
    for(i=0;i<5;i++)
        printf("\n%s",city[i]);
}

```


OUTPUT:

Enter Names of Cities.

MUMBAI

NANDED

BANGLORE

KANPUR

INDORE

Sorted List of Cities.

BANGLORE

INDORE

KANPUR

MUMBAI

NANDED

Explanation:

In the above program, standard string functions `strcmp()` and `strcpy()` are used. The `strcmp()` function compares two successive city names. If their ASCII difference is greater than zero, city names are exchanged. This is accomplished by the body of the `if` statement. Thus, on the execution of the program cities are displayed in the alphabetical order.

- 8.50/8.51 Write a program to find the number of words in a given statement. Exclude spaces between them.

```
void main()
{
    char text[30];
    int count=0,i=0;
    clrscr();
    printf("Enter The Line of Text \n");
    printf("Give One Space After Each word\n");
    gets(text);
    while(text[i++]!= '\0')
    if(text[i]==32 || text[i]== '\0')
    count++;
    printf("The Number of words in line = %d\n",count);
}
```

OUTPUT:

Enter The Line of Text

Give One Space After Each word

Read Books

The Number of words in line = 2

Explanation:

In the above program, a string is entered. It is known that the single space separates two consecutive words. The logic for finding the number of words in a statement is to detect the number

of spaces and NULL characters. For example, in a statement 'C IS A PROGRAMMING LANGUAGE' there are four spaces and a NULL character when the string is terminated. Thus, the total characters are five (4+1). The `if` statement counts the number of spaces and NULL characters in the string.

OR

```
void main()
{
    char text[30];
    int count=0,i=0;
    clrscr();
    printf("Enter Text Below :");
    gets(text);
    while(text[i]!='\0')
    {
        if(((text[i]>=97) && (text[i]<=122)) || ((text[i]>=65) &&
            (text[i]<=90)))
        {
            i++;
            if(text[i]==32)
            {
                count++;
                i++;
            }
        }
    }
    if(text[i]=='\0')
    count++;
    printf("The Number of words in line = %d\n",count);
}
```

OUTPUT:

```
Enter Text Below : Reading is a good Habit
The Number of words in line = 5
```

Explanation:

In the above program, a string is entered. The first `if` statement within the `while` loop checks every element of string whether it is a character or space. If it is a character variable 'i' is incremented and checked with second `if` statement. If it is space then countervariables 'i' and 'count' are incremented. Thus, by counting spaces and NULL characters the total number of words is calculated.

- 8.52 Read the names of mobile customers through keyboard and sort them alphabetically on the last name. Display the sorted list on the monitor.

```
# include <string.h>
void main()
{
    char fname[20][10], sname[20][10], surname[20][10];
    char name[20][20], mobile[20][10], temp[20];
```

```

int i,j;
clrscr();
printf("Enter Names and Mobile Numbers.\n");
for(i=0;i<5;i++)
{
    scanf("%s %s %s %s",fname[i],sname[i],surname[i],mobile[i]);
    strcpy(name[i],surname[i]);
    strcat(name[i],",");
    temp[0]=fname[i][0];
    temp[1]='\0';
    strcat(name[i],temp);
    strcat(name[i],".");
    temp[0]=sname[i][10];
    temp[1]='\0';
    strcat(name[i],temp);
}
for(i=1;i<=5-1;i++)
for (j=1;j<=5-i;j++)
if(strcmp(name[j-1],name[j])>0)
{
    strcpy(temp,name[j-1]);
    strcpy(name[j-1],name[j]);
    strcpy(name[j],temp);
    strcpy(temp,mobile[j-1]);
    strcpy(mobile[j-1],mobile[j]);
}
strcpy(mobile[j],temp);
printf("List of Customers in alphabetical Order.");
for(i=0;i<5;i++)
printf("\n%-20s\t %-10s\n",name[i],mobile[i]);
}

```

OUTPUT:

Enter Names and Mobile Numbers.

```

K S MORE      458454
J M CHATE     658963
M M GORE      660585
L K JAIN      547855
J J JOSHI     354258

```

List of Customers in alphabetical Order.

```

CHATE J.M.    658963
GORE M M     660585
JAIN L K     547855
JOSHI J J    354258
MORE K S     458454

```

Explanation:

The logic used here is the same as logic used in the program where city names are sorted alphabetically.

SUMMARY

This chapter is focused on strings. In this chapter, you have learnt how to declare and initialize strings. It is also very important to identify the end of the string. This is followed by NULL (\0) character. The various formats for display of the strings are demonstrated with numerous examples.

String handling has strong impact in our life string problems such as conversion of lower to uppercase, reversing, concatenation, comparing, searching and replacing of string elements. It is also discussed how to perform these activities with and without standard library functions.

Memory functions have also been illustrated together with programming examples.

After having performed programs discussed in this chapter, the programmer should not face any problem in solving string-handling applications.

EXERCISES

I Fill in the blanks:

1. The _____ is a group of characters, digits and symbols.
 - (a) number
 - (b) array
 - (c) string
2. The string is terminated with _____ character.
 - (a) '?'
 - (b) '\0'
 - (c) '#'
3. Consider the declaration `char name[10];`. Out of following, _____ cannot be held in name?
 - (a) "A12BC34D"
 - (b) "hello"
 - (c) "1a3b5c7d9e"
4. _____ is not the library string function.
 - (a) `strlen()`
 - (b) `strrev()`
 - (c) `strstrstr()`
5. _____ copies one string to another.
 - (a) `strstr()`
 - (b) `strcpy()`
 - (c) `strcat()`
6. The length of string name in the following program segment is _____.


```
char[]name
={ 'h', 'e', 'l', 'l', 'o', '\0' };
printf("%d", strlen(name));
```

 - (a) 5
 - (b) 6
 - (c) 7
7. The `printf` statement in the following program segment prints _____.


```
char name[] = "AbCdEf";
printf("%s", strlen(strlwr(strupr(name))));
```

 - (a) AbCdEf
 - (b) ABCDEF
 - (c) Abcdef
 - (d) none of the above three
8. _____ header file is to be included for using string functions.
 - (a) `str.h`
 - (b) `string.h`
 - (c) `stdio.h`
9. The `printf()` statement in the following program segment prints _____.


```
char name[] = "India";
char *a;
a = name;
printf("%s", a);
```

 - (a) India
 - (b) I
 - (c) no output
10. Keyword `CONST` can be used with _____.
 - (a) string
 - (b) pointer
 - (c) both string and pointer

11. The `free()` frees blocks allocated with _____.
 (a) `malloc()`
 (b) `gets()`
 (c) `stralloc()`
12. _____ is used to allocate main memory.
 (a) `gets()`
 (b) `malloc()`
 (c) `stralloc()`
13. _____ is used for duplication of a string.
 (a) `gets()`
 (b) `strdup()`
 (c) `strcmp()`
14. _____ appends source to destination string.
 (a) `strcat()`
 (b) `strdup()`
 (c) `strcmp()`

II True or false:

- Array of characters is also known as a string.
- The name of array is the pointer to the first element of the array.
- `strlen()` function is used to find the length of the string.
- In `const char* p = "Hello"` pointer is fixed and string is constant.
- The array of pointers, declared as `char *a[]`, is a 2D character array.
- In any array, its *i*th element can be accessed using the - `arr[i]`, `i[arr]`, `*(i+arr)` and `*(arr+i)`.
- Function `malloc()` can be used to allocate space in memory at compile time.
- In multidimensional array the consecutive arrays are not stored in contiguous memory locations.
- If you want to take the address of a person as an input, you will mostly prefer `scanf()`.
- The efficient declaration in terms of memory for multidimensional string is `*mul[]` than `mul[20][30]`.
- The string is declared with `int name[]`;
- "1ABCDE2345" is a valid string.
- "\$ABCDE4567" is an invalid string.
- NULL character appears at first if the string is reversed.
- `strlen()` counts the number of characters including `\0` (null).
- `strcmp()` compares two arrays elements by elements.
- 'ABCD' is a valid string.
- `strcpy()` copies string from the destination to source string.

III Match the functions/words given in Group A with meanings in Group "B":

1.

Group A		Group B	
Sr. No.	Statement	Sr. No.	Output
1.	<code>printf("%s\n", text);</code>	A	PRABHAKAR
2.	<code>printf("%.5s\n", text);</code>	B	PRAB
3.	<code>printf("%.8s\n", text);</code>	C	PRABHAKAR
4.	<code>printf("%.15s\n", text);</code>	D	PRABHAKA
5.	<code>Printf("%-10.4s\n", text);</code>	E	PRABH
6.	<code>Printf("%11s", text);</code>	F	PRABHAKAR

2.

Group A		Group B	
Sr. No.	Functions	Sr. No.	Working
1	strlen()	A	Duplicate the string
2	strcpy()	B	Copies the string
3	strdup()	C	Returns length of the string
4	strrev()	D	Reverse the string

IV Selecting the appropriate option from the multiple choices given below:

1. The string always ends with

- (a) '\0' character
- (b) '\\' character
- (c) '0\' character
- (d) None of the above

2. What will be the output of the program

```
void main()
{
    char nm[]={ 'A', 'N', 'S', 'I', '\0',
                'C', '\0' };
    int x=0;
    clrscr();
    while (nm[x]!='\0')
        printf ("%c", nm[x++]);
}
```

- (a) ANSI
- (b) ANSI0C
- (c) ANSIC
- (d) None of the above

3. What will be the size of character array

```
void main()
{
    char x[]={ 's', 'a', '\0' };
    printf ("%n %d", sizeof(x));
}
```

- (a) 3
- (b) 2
- (c) 0
- (d) None of the above

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

```
# include <string.h>
void main()
{
    char x[]="a1b2c3d4e5f6g7h8i9j0";
    int t=0;
    clrscr();
    for(t=1; x[t]!='\0' &&
        t<=strlen(x); t+=2)
        printf ("%c", x[t]);
}
```

- (a) 1234567890
- (b) abcdefghij
- (c) a1b2c3d4e5f6g7h8i9j0
- (d) None of the above

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

```
void main()
{
    char txt[]="12345\0abcdef";
    clrscr();
    printf ("%s", txt);
}
```

- (a) 12345
- (b) abcdef
- (c) 12345\0abcdef
- (d) None of the above

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

```
void main()
{
    char txt[]="ABCDEFGH\0IJKL";
    clrscr();
    printf ("%s %d", txt, sizeof
        (txt));
}
```

- (a) ABCDEF 14
- (b) ABCDEF\0GHIJKL 14

- (c) ABCDEF 7
- (d) None of the above

V Attempt the following programming exercises:

- Write a program to arrange a set of fruit names given below in descending order (reverse alphabetic). (mango, banana, apple, orange, grapes, coconut, water melon and papaya).
- Write a program to arrange the following names in the alphabetic order. The sorting is to be done on the first three characters of the first name.
(Ashok, Alok, Akash, Amit, Amol, Anil, Ashish and Anand).
- Write a program to enter some text and display the text in reverse order. (Example: 'I am happy' will be displayed as 'happy am I').
- Write a program to enter five full names of persons. Display their names, initials and last names.
- Write a program to enter text through keyboard. Convert first character of each word in capital and display the text.
- Write a program to enter some text through the keyboard. Insert dot (.) after every three words in the text. The first character after every dot should be converted to capital.
- Write a program to enter some text through the keyboard. Count the number of words that starts from 'w'. Display such words and count them.
- Write a program to print the entered word with all possible combinations.
- Write a program to encrypt the text 'INDIA'. The output should be 'KPFKC'. ('A' is to be replaced with 'C', 'B' with 'D' and 'C' with 'E' and so on.)
- Write a program to decrypt the text 'KPFKC' to get original string 'INDIA'.

VI Answer the following questions:

- What are strings? How are they declared?
- What is the NULL character? Why is it important?
- Is it possible to initialize the NULL character in the string?
- Why is it necessary to count NULL characters while declaring string?
- What is the difference between the functions `strcmp()` and `stricmp()`?
- What is the use of `strrev()` and `strlen()` functions?
- What is the use of `strcpy()` and `strdup()` functions?
- What is the difference between `strcpy()` and `strncpy()` functions?
- What is the difference between `NULL`, `'\0'` and `0`?
- Describe any of the memory functions.
- Describe any three string conversion functions.
- What is the use of `atoi()` function?

VII What will be the output/s of the following program/s?

- ```
#include <string.h>
void main()
{
 char a1[12]={ "MAHARASHTRA" };
 clrscr();
 printf("%s", a1);
 getch();
}
```

**2.**

```
include <string.h>
void main()
{
 char a1[12]="MAHARASHTRA";
 char *a2[13]={ "MADHYAPRADESH"};
 clrscr();
 printf("%s", a1);
 printf("%s", *a2);
 getch();
}
```

**3. Program to find the length of the string.**

```
void main()
{
 char text[20];
 int len;
 clrscr();
 printf("type text below.\n");
 gets(text);
 len=strlen(text);
 printf("length of the string
is=%d", len);
}
```

#### 4. Program to copy the string using strcpy function

```
void main()
{
 char name1[10]={'P','r','i','y','a'},name2[10];
 clrscr();
 printf("\n Original name=%s",
 name1);
 strcpy(name2,name1);
 printf("\n Copied name
is=%s",name2);
 getch();
}
```

## 5. Without strcpy

```
void main()
{
```

```
char main[10]="abcd",dup[10];
int i;
clrscr();
for(i=0;i<10;i++)
dup[i]=main[i];
printf("Original String :
%s",main);
printf("\nDuplicate String :
%s",dup);
getche();
```

**6.**

```
void main()
{
 char group[15]={'I','n','d','i','a'};
 int len=0,i;
 clrscr();
 for(i=0;group[i]!='\0';i++)
 len++;
 printf("Length of the string is=%d", len);
 getch();
}
```

## 7. Program on comparison of two strings

```
void main()
{
 char str1[20]={'C','o','m','p',
 'u','t','e','r'},
 str2[20]={'C','o','m','p','u',
 't','e','r'};
 clrscr();
 if((strcmp(str1,str2))==0)
 printf("Length of the strings
 are same \n");
 else
 printf("Length of the strings
 are not same \n");
 getch();
}
```

### VIII Find the bug/s in the following programs:

**1.**

```
include <string.h>
void main()
{
 char a1[]={"KOLKATA"};
 int i;
 clrscr();
 while(a1[i++]!='\0')
```

```
printf("%c", a1[i]);
getche();
```

2.

```
include <string.h>
void main()
{
```



- ```

char a1[6]={'NAGPUR'};
clrscr();
printf("%c",a1);
getche();
}
3. # include <string.h>
void main()
{
    char *a[]="ROYAL";
    clrscr();
    printf("%s %u",*(a),&*a);
    getche();
}
4. # include <string.h>
void main()
{
    char a1[6]={"CHENNAI"};
    int i;
    clrscr();
    for(i=0;i<7;i++)
    printf("%c",a1[i]);
    ++i;
    getche();
}
5. void main()
{
    char text[15]="MOUNTAIN";
    int i;
    clrscr();
    for(i=7;i>=0;i--)
    printf("%.8d",text[i]);
    getche();
}
6. # include <string.h>
void main()
{
    char *a2[10]={"MADHYAPRADESH"};
    clrscr();
    printf("%s", *&(a2));
    getche();
}
7. void main()
{
    char str1[15]='snowy',
    str2[15]='sunny';
    clrscr();
    printf("\n Source String-
    :%s", str1);
    printf("\n Destination
    String-:%s",str2);
    strncpy(str2,str1,3);
    printf("\nDestination String
    -:%s",str2);
    getche();
}

```

ANSWERS

I Fill in the blanks:

Q.	Ans.
1. c. String It is a group of characters, digits and symbols. Actually it is an array of characters. So, more specific answer is String.	
2. b. '\0' Every string in C is terminated with '\0' (NULL) character.	
3. c. 1a3b5c7d9e In C the maximum characters that can fit into a string is less than its size by 1. Because, the string always terminates with '\0' (NULL) character.	
4. c. strstrstr() There is no function strtstrstr() in C.	

5. b. strcpy()
strcpy() function is used to copy a string from source to destination string.
6. a. 5
In C the string i.e. character array is terminated by '\0' (NULL) character. If the '\0' is specified like in declaration the compiler doesn't insert more '\0'. Also, the strlen() function gives the length of string without '\0' character.
7. d. None of the above three
Understand the sequence of functions. The outer function executes last. So we are trying to print integer with %S. It'll print something which we cannot predict.
8. b. string.h
In C the string.h header file contains the functions needed by the strings.
9. a. India
In character array the name of the array is same as the pointer to the first element.
10. c. both string and pointer
11. a. malloc()
12. b. malloc()
13. b. strdup()
14. a. strcat()

II True or False:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	T	2.	T	3.	F	4.	F	5.	T
6.	T	7.	F	8.	F	9.	F	10.	T
11.	F	12.	T	13.	F	14.	F	15.	F
16.	T	17.	F	18.	F.				

III Match the functions/words given in Group A with meanings in Group "B":

1.

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	F	2.	E	3.	D	4.	C	5.	B
6.	A								

2.

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	C	2.	B	3.	A	4.	D

IV Selecting the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	a	4.	a	5.	a
6.	a								

VII What will be the output/s of the following program/s?

Q.	Ans.
1.	MAHARASHTRA
2.	MAHARASHTRA MADHYAPRADESH
3.	My name is=Romali Length of the string is=6
4.	Original name=Priya Copied name is=Priya
5.	Original String : abcd Duplicate String : abcd
6.	Length of the string is=5
7.	Length of the strings are same

VIII Find the bug/s in the following programs:

Q.	Ans.
1.	'i' is to be initialized with some value say i=-1.
2.	Array initialization is incorrect and %S is to be used instead of %C.
3.	ROYAL must be enclosed with double quote.
4.	char array is to be initialized with 7 instead of 6.
5.	Replace c instead of d in printf statement.
6.	'&' operator to be deleted.
7.	Double quotation mark is needed in char statement.

CHAPTER

9

Pointers

Chapter Outline

- 9.1** Introduction
- 9.2** Features of Pointers
- 9.3** Pointers and Address
- 9.4** Pointer Declaration
- 9.5** The void Pointers
- 9.6** Wild Pointers
- 9.7** Constant Pointers
- 9.8** Arithmetic Operations with Pointers
- 9.9** Pointers and Arrays
- 9.10** Pointers and Two-Dimensional Arrays
- 9.11** Pointers and Multi-Dimensional Arrays
- 9.12** Array of Pointers
- 9.13** Pointers to Pointers
- 9.14** Pointers and Strings

9.1 INTRODUCTION

Pointer is a special data-type in C and it is widely used by programmers. Most of the computer language learners feel that the pointer is a puzzling topic and very difficult to understand. However, the pointer enables fast and straightforward execution of a program. With pointers, memory is used most efficiently. C gives more importance to pointers. Hence, it is important to know the operation and applications of pointers. Pointers are used as a tool in C and if you fail to understand it, you will be losing the power of C.

In C, variables are used to hold data values during the execution a program. Every variable when declared occupies certain memory location/s. For example, integer-type variable takes two bytes of memory, character type one byte and float type four. With pointers, one can manipulate memory addresses. It is possible to access and display the address of a memory location of a variable using the '&' operator. Memory is arranged in a series of bytes. These series of bytes are numbered from zero onwards. The number specified to a cell is known as the memory address. Pointer variable stores the memory address of any type of variable. The pointer variable and normal variable should be of the same type. The pointer is denoted by an asterisk (*) symbol.

A byte is nothing but a combination of eight bits as shown in Figures 9.1 and 9.2. The binary numbers 0 and 1 are known as bits. Each byte in the memory is specified with a unique (matchless) memory address. The memory address is an unsigned integer starting from zero to uppermost addressing capacity of the microprocessor. The number of memory locations pointed by a pointer depends on the type of pointer. The programmer should not take care and worry about addressing procedure of variables. The compiler knows and performs the procedure of a pointer. The pointers are either 16 bits or 32 bits long.

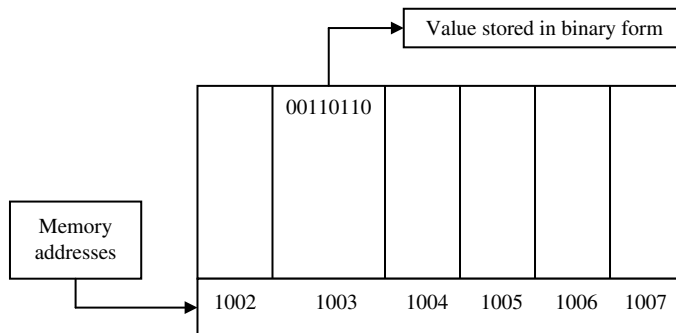


Figure 9.1 Memory picture

The allocation of memory during a program run time is called the dynamic memory allocation. Such type of memory allocation is essential for data structures and can efficiently handle it using pointers. Another reason to use pointers is in arrays. Arrays are used to store more values. Actually, the name of array is a pointer. One more reason to use pointers is command-line arguments. Command-line arguments are passed to programs and stored in an array of pointers `argv[ ]`.

Pointer: Pointer is a special variable that stores the address of another variable. Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variables but is always denoted by an asterisk (*) operator.

9.2 FEATURES OF POINTERS

- (i) Pointers are a fundamental tool in developing code in C.
- (ii) Pointers save the memory space.
- (iii) The memory is accessed efficiently with the pointers. The pointer assigns the memory space and also releases it. Pointers are used to allocate memory dynamically.
- (iv) Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
- (v) Access elements of an array of any data type irrespective of its subscript range.
- (vi) Pointers are used for file handling.
- (vii) They allow us to create linked lists and other algorithmically oriented data structures.
- (viii) Execution time with pointer is faster because data is manipulated with the address.

9.3 POINTERS AND ADDRESS

The theory of the pointer is not complicated. The computer memory is made by the semi-conductor technology. Memory comprises binary cells. Each cell has a capacity to store a bit either 0 or 1. Byte means a unit of 8 bits. Every such byte has a unique memory address as shown in Figure 9.2.

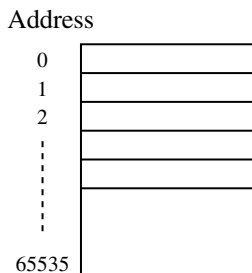


Figure 9.2 *Memory address*

When a variable of any data type is declared, memory according to its data type is reserved. For example:

```
int x,y;
char c;
```

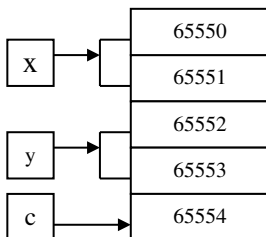


Figure 9.3 *Memory allocated*

As shown in Figure 9.3, memory locations are reserved for the variables *x*, *y* and *c*. For an integer variable two bytes each and for character variable one byte is reserved. The locations 65550 and 65551 are used to store *x*, and 65552 and 65553 are used to store *y* and so on. The starting address for *x* is 65550 and for *y*, it is 65552. The character needs only one byte and its location shown in Figure 9.3 is 65554.

9.4 POINTER DECLARATION

Pointer variables can be declared as follows.

Example:

```
int *x;
float *f;
char *y;
```

- (i) In the first statement, 'x' is an integer pointer and tells the compiler that it holds the address of any integer variable. In the same way, 'f' is a float pointer which stores the address of any float variable and 'y' is a character pointer that stores the address of any character variable.
- (ii) The indirection operator (*) is also called the dereferencing operator. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.
- (iii) Normal variable provides direct access to their own values, whereas a pointer provides indirect access to the values of the variable whose address it stores.
- (iv) The indirection operator (*) is used in two distinct ways with pointers, declaration and dereference.
- (v) When a pointer is declared, a star (*) indicates that it is a pointer and not a normal variable.
- (vi) When the pointer is dereferenced, the indirection operator indicates that the value at that memory location stored in the pointer is to be accessed rather than the address itself.
- (vii) Also note that * is the same operator that can be used as the multiplication operator. The compiler knows which operator to call, based on the context.
- (viii) '&' is the address operator and represents the address of the variable. %u is used with the printf() for printing the address of a variable. The address of any variable is a whole number. The operator '&' immediately preceding the variable returns the address of the variable. In the below given example, '&' is immediately preceding the variable 'num' which provides the address of the variable.

➤ 9.1 Write a program to display the address of the variable.

```
void main()
{
    int num;
    clrscr();
    printf("Enter a Number = ");
    scanf("%d", &num);
    printf("Value of num =%d\n", num);
    printf("Address of num=%u\n", &num);
    getch();
}
```

OUTPUT:

```
Enter a Number = 20
Value of num = 20
Address of num = 4066
```

Explanation:

The physical memory location of a variable is system-dependent; hence, the address of the variable cannot be predicted immediately. In the above example, the address of the variable 'num' is 4066. Here, in the below given Figure 9.4, three blocks are shown related to the above program. The first block contains variable name. The second block represents the value of the variable. The third block is the address of the variable 'num' where 20 is stored. Here, 4066 is the memory address.

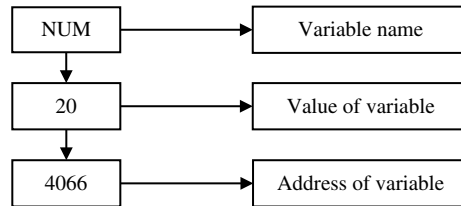


Figure 9.4 Variable value and its address

➤ 9.2 Write a program to display the addresses of different variables together with their values.

```

void main()
{
    char c;
    int i;
    float f;
    clrscr();
    printf("Enter alphabet, number, float=");
    scanf("%c %d %f",&c,&i,&f);
    printf("Value of c=%c i=%d f=%f\n",c,i,f);
    printf("\nAddress of(c)%C =%u",c,&c);
    printf("\nAddress of(i)%d =%u",i,&i);
    printf("\nAddress of(f)%.2f =%u",f,&f);
    getch();
}
  
```

OUTPUT:

```

Enter alphabet, number, float = C 20 2.5
Address of(c)c = 4061
Address of(i)20 = 4062
Address of(f)2.5 = 4064
  
```

Explanation:

The program mentioned above declares three variables of different data types. Their values are displayed together with respective addresses. Memory addresses are unsigned integers. From the above example, a programmer can understand that the difference between two successive addresses is not constant due to different data types.

- 9.3 Write a program to show pointer of any data type that occupies the same space.

```
void main()
{
    clrscr();
    printf("\t char %d byte & its pointer %d bytes\n", sizeof(char), sizeof(char*));
    printf("\t int %d byte & its pointer %d bytes\n", sizeof(int), sizeof(int*));
    printf("\t float %d byte & its pointer %d bytes\n", sizeof(float), sizeof(float*));
}
```

OUTPUT:

```
char 1 byte & its pointer 2 bytes
int 2 byte & its pointer 2 bytes
float 4 byte & its pointer 2 bytes
```

Explanation:

In the above program, using `sizeof()` operator size of data type and its pointer are calculated and displayed. It can be observed from the above program that data type requires different number of bytes for storage whereas pointer of any data type requires two bytes. On linux platform size of pointers is four bytes.

- 9.4 Write a program to display the value of variables and its location using pointer.

```
void main()
{
    int v=10, *p;
    clrscr();
    p=&v;
    printf("\n Address of v = %u", p);
    printf("\n Value of v = %d", *p);
    printf("\n Address of p= %u", &p);
}
```

OUTPUT:

```
Address of v = 4060
Value of v = 10
Address of p = 4062
```

Explanation:

In the above program, 'v' is an integer variable and its value is 10. The variable 'p' is declared as a pointer variable. The statement `p = &v` assigns address of 'v' to 'p' i.e. 'p' is the pointer to variable 'v'. To access the address and value of 'v', pointer 'p' can be used. The value of 'p' is nothing but an address of variable 'v'. To display the value stored at that location `*p` is used. The pointer variables also have addresses and are displayed using '&' operator. The statement used for finding pointer address, in this example, the statement used is `printf("\n Address of p = %u", &p);` (see Figure 9.5).

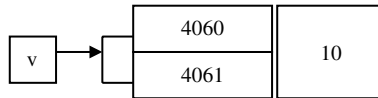


Figure 9.5 Variable, addresses, and value

The value of 'v' is 10 and it is stored at location 4060. The address of 'v' 4060 is assigned to variable 'p'. The address of pointer variable 'p' is 4062. Here, 'p' is the pointer-type variable, which points to the variable 'v'. Hence, it must be declared as `int v = 10;` and pointer variable as `int *p;`

- 9.5 Write a program to print the value of variables in different ways. How would you use '*' and '&' operators for accessing the values:

```
void main()
{
    int a,*pa;
    float b,*pb;
    clrscr();
    printf("Enter Integer & float Value :");
    scanf("%d %f",&a,&b);
    pa=&a;
    pb=&b;
    printf("\n Address of a=%u",pa);
    printf("\n Value of a=%d",a);
    printf("\n Value of a=%d",*(&a));
    printf("\n Value of a=%d",*pa);
    printf("\nAddress of b=%u",pb);
    printf("\n Value of b=%0.2f",b);
    printf("\n Value of b=%0.2f",*(&b));
    printf("\n Value of b=%0.2f",*pb);
}
```

OUTPUT:

```
Enter Integer & float Value : 4 2.2
Address of a=4054
Value of a=4
Value of a=4
Value of a=4
Address of b=4060
Value of b=2.20
Value of b=2.20
Value of b=2.20
```

Explanation:

The output of the program is as shown above. The program illustrates for accessing the values of variables 'a' and 'b' using pointers. The address of 'pa' is 4054 and it points to 4. The address of 'pb' is 4060, and it points to 2.2.

Thus, from the above example, we have noticed the equivalencies as given below.

. a=*(&a)=*pa

Explanation of *(&a): Here, &a is used to get the address of the variable 'a'. The content stored in this location is accessed by '*'. The declaration *(&a) acts as *pa.

A simple example is given to understand the above concepts.

- 9.6 Write a program to print the value of a variable using different pointer notations.

```
void main()
{
    int v=10,*p;
    clrscr();
    p=&v;
    printf("\n v = %d v = %d v = %d",v,*(&v),*p);
}
```

OUTPUT:

v = 10 v = 10 v = 10

Explanation:

In the above program to display the value of variable 'v', three different syntaxes are used in the printf() statement.

- 9.7 Write a program to print an element and its address using pointer.

```
void main()
{
    int i,*k;
    clrscr();
    printf("Enter a number : ");
    scanf("%d",&i);
    k=&i;
    printf("\nAddress of i is %u",k);
    printf("\nValue of i is %d",*k);
}
```

OUTPUT:

Enter a number : 15
Address of i is 4062
Value of i is 15

Explanation:

In the above program, the address of variable 'i' is assigned to pointer variable 'k' with statement k=&i;. Hence, 'k' is pointing to 'i'. The value of the variable 'i' is displayed using pointer (*k) with the statement printf("\nValue of i is %d",*k);

- 9.8 Write a program to add two numbers through variables and their pointers.

```
void main()
{
```

```

int a,b,c,d,*ap,*bp;
clrscr();
printf("Enter Two Numbers :");
scanf("%d %d",&a,&b);
ap=&a;
bp=&b;
c=a+b;
d=*ap*bp;
printf("\nSum of A & B Using Variable :%d",c);
printf("\nSum of A & B Using Pointers :%d",d);
}

```

OUTPUT:

```

Enter Two Numbers : 8 4
Sum of A & B Using Variable : 12
Sum of A & B Using Pointers : 12

```

Explanation:

In the above program, the sum of two numbers is obtained in two ways.

- (i) Adding variables 'a' and 'b' directly.
- (ii) Through pointers of 'a' and 'b' i.e. '*ap*' and '*bp*', respectively.

The addresses of 'a' and 'b' are stored in '*ap*' and '*bp*', respectively. Thus, adding '**ap*' and '**bp*' gives an addition of 'a' and 'b'. Here, '*ap*' and '*bp*' mean addresses of 'a' and 'b' and '**ap*' and '**bp*' mean the values at memory locations. Hence, the equation used here is $d = *ap + *bp$; and not $d = ap + bp$.

➤ 9.9 Write a program to assign the pointer value to another variable.

```

void main()
{
    int a=5,b,*c;
    c=&a;
    b=*c;
    clrscr();
    printf("\nMemory location of 'a'=%u",c);
    printf("\nThe value of a=%d & b=%d",a,b);
}

```

OUTPUT:

```

Memory location of 'a'= 4062
The value of a=5 & b=5

```

Explanation:

In the above program, the pointer variable 'c' is assigned the memory location of 'a'. The value stored in this memory location is assigned to 'b' through pointer c.

➤ 9.10 Write a program to assign the value of 'b' to 'a' through pointers. Show the effect of addition before and after the assignment of the value of 'b' to 'a'.

```

void main()
{
    int a,b,*pa,*pb;
    clrscr();
    printf("Enter Value of 'a' & 'b'. :");
    scanf("%d %d",&a,&b);
    pa=&a;
    pb=&b;
    printf("\nValue of a=%d & b=%d",a,b);
    printf("\nAddress of a=%u",pa);
    printf("\nAddress of b=%u",pb);
    printf("\n\nAddition of 'a' & 'b' : %d",*pa+*pb);
    pa=pb;
    printf("\n*pa=%d *pb=%d",*pa,*pb);
    printf("\nnpa =%u pb=%u",pa,pb);
    printf("\nAddition of *pa +*pb : %d",*pa+*pb);
    getch();
}

```

OUTPUT:

```

Enter Value of 'a' & 'b'. : 8 4
Value of a=8 & b=4
Address of a=4056
Address of b=4058
Addition of 'a' & 'b' : 12
*pa=4 *pb=4
pa=24350 pb=24350
Addition of *pa & *pb : 8

```

Explanation:

In the above program, addresses of variables 'a' and 'b' are assigned to pointers 'pa' and 'pb'. The statement `pa = pb;` means the value of pointer 'pb' is assigned to pointer 'pa'. Now, both the pointers have the same value and they point to the same variable 'b'. Hence, the addition is 8.

9.5 THE void POINTERS

Pointers can also be declared as void types. void pointers cannot be dereferenced without explicit type conversion. This is because being void the compiler cannot determine the size of the object that the pointer points to. Though void pointer declaration is possible, void variable's declaration is not allowed. Thus, the declaration `void p` will display an error message 'Size of 'p' is unknown or zero' after compilation.

A pointer points to an existing entity. A void pointer can point any type of variable with proper type casting. The size of a void pointer displayed will be two. When a pointer is declared as void two bytes are allocated to it. Later using type casting, a number of bytes can be allocated or deallocated. void variables cannot be declared because memory is not allocated to them and there is no place to store the address. Therefore, void variables cannot serve the job actually they are made for.

- 9.11 Write a program to declare a void pointer. Assign the address of `int`, `float` and `char` variables to the void pointer using the typecasting method. Display the contents of various variables.

```
int p;
float d;
char c;
void *pt = &p;
void main
{
    clrscr();
    *(int *) pt = 12;
    printf("\n p= %d", p);
    pt=&d;
    *(float *)pt = 5.4;
    printf("\n r=%g", d);
    pt=&c;
    *(char*)pt='S';
    printf("\n c=%c", c);
}
```

OUTPUT:

```
p=12
r=5.4
c=S
```

Explanation:

In the above example, variables `p`, `d` and `c` are variables of types `int`, `float` and `char`, respectively. Pointer `pt` is a pointer of type `void`. These entire variables are declared before `main()`. The pointer is initialized with the address of integer variable `p`, i.e. the pointer `pt` points to variable `p`. The statement `*(int *) pt = 12` assigns the integer value 12 to pointer `pt` i.e. to a variable `p`. The contents of variable `p` are displayed using the succeeding statement. The declaration `*(int *)` tells the compiler the value assigned is of the integer type. Thus, the assignment of `float` and `char` types is carried out. The statements `*(int *) pt = 12`, `*(float *)pt =5.4` and `*(char*) pt='S'` help the compiler to exactly determine the size of the data type.

9.6 WILD POINTERS

Pointers are used to store memory addresses. An improper use of pointer creates many errors in the program. Hence, pointers should be handled cautiously. When a pointer points to an unallocated memory location or data value whose memory is de-allocated, such a pointer is called a wild pointer. The wild pointer generates garbage memory location and dependent reference. The pointer becomes wild due to the following reasons:

- (i) Pointer declared but not initialized
- (ii) Pointer alternation
- (iii) Accessing the destroyed data

When a pointer is declared and not initialized, it holds an unauthorized address. It is very difficult to manipulate such pointers.

➤ 9.12 Write a program to show the wild pointer and its output.

```
void main()
{
    int k,*x;
    /*clrscr();*/
    for(k=0;k<=3;k++)
        printf("%u",x[k]);
}
```

OUTPUT:

7272 24330 30559 27753

Explanation:

In this program, pointer *x* is not initialized. The successive locations are displayed.

- (i) The forgetful assignment of a new memory location in a pointer is called the pointer alternation. This happens when the wild pointer accesses the location of the wild pointer. The wild pointer converts the legal pointer to the wild pointer.
- (ii) Sometimes the pointer attempts to access the data that has no longer life.

9.7 CONSTANT POINTERS

The address of the constant pointer cannot be modified.

```
char* const str="Constant";
```

In the above example, it is not possible to modify the address of the pointer *str*. The following operations will generate error messages:

```
str='san' /* cannot modify a constant object */
++str /* cannot modify a constant object */
```

If pointer constant is pointing to a variable, we can change the value of the actual variable but not through the constant pointer. The following valid and invalid operations are given:

```
k=5 /* possible */
*pm=5 /* invalid */
pm++ /* possible */
*pm++ /* invalid */
*pm=5 /* invalid */
```

➤ 9.13 Write a program to declare a constant pointer and modify the value.

```
void main()
{
    int k=10;
    int const *pm=&k;
```

```
clrscr();
k=40;
/* *pm=50; invalid operation */
printf("k=%d",k);
}
```

OUTPUT:

k=40

Explanation:

In the above program, variable `k` is an integer and `pm` is a constant pointer of the same type. The value of `k` cannot be changed through the pointer `pm`, because `pm` is a constant pointer. The value of a variable can be changed through itself only. Any constant entity cannot permit any modification.

9.8 ARITHMETIC OPERATIONS WITH POINTERS

Arithmetic operations on pointer variables are also possible. Increment, decrement, prefix and postfix operations can be performed with the pointers. The effects of these operations are shown in Table 9.1.

Table 9.1 *Pointer and arithmetic operation*

<i>Data Type</i>	<i>Initial Address</i>	<i>Operation</i>		<i>Address after Operations</i>		<i>Required Bytes</i>
int i=2	4046	++	—	4048	4044	2
char c='x'	4053	++	—	4054	4052	1
float f=2.2	4058	++	—	4062	4054	4
long l=2	4060	++	—	4064	4056	4

From Table 9.1, we can observe that, on increment of the pointer variable for integers, the address is incremented by two, i.e. 4046 is the original address and on increment its value will be 4048 because integers require two bytes.

Similarly, characters, floating point numbers and long integers require 1, 4 and 4 bytes, respectively. After the effect of increment and decrement the memory locations are shown in Table 9.1.

- 9.14 Write a program to show the effect of increment on pointer variables. Display the memory locations of integer, character and floating point numbers before and after the increment of pointers.

```
void main()
{
    int x,*x1;
    char y,*y1;
    float z,*z1;
    clrscr();
```



```

printf("Enter integer, character, float Values \n");
scanf("%d %c %f",&x,&y,&z);
x1=&x;
y1=&y;
z1=&z;
printf("Address of x = %u\n",x1);
printf("Address of y = %u\n",y1);
printf("Address of z = %u\n",z1);
x1++;
y1++;
z1++;
printf("\nAfter Increment in Pointers\n");
printf("\nNow Address of x=%u\n",x1);
printf("Now Address of y=%u\n",y1);
printf("Now Address of z=%u\n",z1);
printf("\nSize of Integer: %d",sizeof(x));
printf("\nSize of Character: %d",sizeof(y));
printf("\nSize of Float: %d",sizeof(z));
}

```

OUTPUT:

```

Enter integer, character, float Values
2 A 2.2
Address of x = 4046
Address of y = 4053
Address of z = 4058
After Increment in Pointers
Now Address of x = 4048
Now Address of y = 4054
Now Address of z = 4062
Size of Integer : 2
Size of Character : 1
Size of Float : 4

```

Explanation:

Observe the output. 4046 is the address of integer 'x', 4053 is the address of character 'y' and 4058 is the address of floating point number 'z'. On the increment of the pointer the address of integer, character and float will be 4048, 4054 and 4062, respectively. This is because a pointer is incremented, it points to immediately next location of its type.

- 9.15 Write a program to show the effect of increment and decrement operators used as prefix and suffix with the pointer variable.

```

void main()
{
    int i, *ii;
    puts("Enter Value of i=");
    scanf("%d",&i);
}

```

```

    ii=&i;
    clrscr();
    printf("Address of i = %u\n",ii);
    printf("Address of i = %u\n",++ii);
    printf("Address of i = %u\n",ii++);
    printf("Address of i = %u\n",--ii);
    printf("Address of i = %u\n",ii--);
    printf("Address of i = %u\n",ii);
}

```

OUTPUT:

```

Enter Value of i= 8
Address of i = 4060
Address of i = 4062
Address of i = 4062
Address of i = 4062
Address of i = 4062
Address of i = 4060

```

Explanation:

- (i) The 1st `printf()` statement displays the address of `i = 4060`.
- (ii) The 2nd `printf()` statement prefix increment is done with pointer variable 'ii'; so before printing it is incremented. Hence, a pointer is an integer type, and an integer requires two bytes in memory. Hence, the address of `i=4062`.
- (iii) The 3rd `printf()` statement is opposite of the 2nd. Here, the address value is printed first and then it is incremented. Hence, it prints `i=4062` and after printing incremented address becomes 4064.
- (iv) In the 4th `printf()` statement, the address of '`i`' is decremented first and then printed. Hence, the address of '`i`' is 4062.
- (v) In the 5th `printf()` statement, the address of '`i`' is printed first and then decremented. On decrement, the address of '`i`' becomes 4060.
- (vi) The last `printf()` statement prints the initial address of '`i`'.

➤ 9.16 Write a program to perform different arithmetic operations using pointers.

```

void main()
{
    int a=25,b=10,*p,*j;
    p=&a;
    j=&b;
    clrscr();
    printf("\n Addition a+b = %d", *p+b);
    printf("\n Subtraction a-b = %d", *p-b);
    printf("\n Product a*b = %d", *p**j);
}

```

```

printf("\n Division a/b = %d", *p / *j);
printf("\n a Mod b = %d", *p % *j);
}

```

OUTPUT:

```

Addition a+b = 35
Subtraction a-b = 15
Product a*b = 250
Division a/b = 2
a mod b = 5

```

Explanation:

The various arithmetic operations can be performed on a pointer such as addition, subtraction, multiplication, division and mod. Here, the value stored at the address is taken into account for operations but not the address. The arithmetic operations are impossible with addresses. For example, two address locations are not possible to add. Various arithmetic operations performed with the above program are elaborated as given below:

- (i) Addition: A number can be added to a pointer or addition of two variables through pointers can also be possible. In the first `printf()` statement, the value of 'b' is added to 'a' through a pointer, i.e. `*p`. The result of addition is 35.
- (ii) Subtraction: A number can be subtracted from a pointer. Subtraction of two variables through pointers can also be possible. In the second `printf()` statement, value of 'b' is subtracted from 'a' through a pointer, i.e. `*p`. The result of subtraction is 15.
- (iii) Multiplication: Multiplication of two pointers or a multiplication of a number with pointer variable can be done. In the third `printf()` statement, multiplication of variables 'a' and 'b' is done through their pointers '`*p`' and '`*j`'.

Similarly, division and mod operations can be carried out as shown in the above program.

Please note that the following operations with addresses are not possible:

- (i) Addition of two addresses (pointers)
 - (ii) Multiplication of addresses or multiplication of address with a constant
 - (iii) Division of address with a constant
- 9.17 Write a program to compare two pointers. Display the message 'Two pointers have the same address' or 'Two pointers have different addresses'.

```

void main()
{
    int a=2,*j,*k;
    j=&a;
    k=&a;
    clrscr();
    if(j==k)
        printf("\n The Two Pointers have the same address.");
    else

```

```
printf("\n The Pointers have the different address.");
printf("\n Address of a(j)=%u", j);
printf("\n Address of a(k)=%u", k);
}
```

OUTPUT:

```
The Two Pointers have the same address.
Address of a(j)= 4056
Address of a(k)= 4056
```

Explanation:

The comparison between the two pointers is done in the above program. The pointer variable should be of the same data type while a comparison is made. The comparison of pointers can test if either address is identical or not. Here, in this program pointers 'j' and 'k' are of integer data types and points to the same integer variable 'a'. Hence, they contain the same address.

9.9 POINTERS AND ARRAYS

- (i) Array name by itself is an address or pointer. It points to the address of the first element (0th element of an array). The elements of the array together with their addresses can be displayed by using array name itself. Array elements are always stored in contiguous memory locations.

Programs in this regard are explained below.

- 9.18 Write a program to display elements of an array. Start element counting from 1 instead of 0.

```
void main()
{
    int x[]={2,4,6,8,10}, k=1;
    clrscr();
    while(k<=5)
    {
        printf("%3d", x[k-1]);
        k++;
    }
}
```

OUTPUT:

```
2 4 6 8 10
```

Explanation:

Array element counting always starts from '0'. The element number is added in the base address and each element of an array is accessed. If one is subtracted from base address of an array, it points to the prior address of 0th element. By adding one to its reduced base address, it is possible to start element counting from '1'.

- 9.19 Write a program to display an array element with their addresses using array name as a pointer.

```
void main()
{
    int x[5]={2,4,6,8,10},k=0;
    clrscr();
    printf("\nElement No. Element Address");
    while(k<5)
    {
        printf("\nx[%d] = \t%8d %9u",k,*(x+k),x+k);
        k++;
    }
}
```

OUTPUT:

Element No.	Element	Address
x[0]=	2	4056
x[1]=	4	4058
x[2]=	6	4060
x[3]=	8	4062
x[4]=	10	4064

Explanation:

In the above program, variable '*k*' acts as an element number and its value varies from 0 to 4. When it is added with an array name '*x*', i.e. with the address of the first element, it points to the consecutive memory location. Thus, the element number, element and their addresses are displayed.

OR

- 9.20 Write a program to display array elements with their addresses using an array name as a pointer.

```
void main()
{
    int num[4]={10,25,35,45},i;
    clrscr();
    printf("Element Address\n");
    for(i=0;i<4;i++)
    {
        printf("num[%d]=%d",i,*(num+i));
        printf("%8u\n",num+i);
    }
}
```

OUTPUT:

Element	Address
num[0] = 10	4062
num[1] = 25	4064
num[2] = 35	4066
num[3] = 45	4068

Explanation:

In the above program, the array name '*num*' itself acts as a pointer to the array *num*[]. The pointer '*num*' provides the address of the first array element and '**num*' gives the value stored at that address. When '*i*' is added with '*num*', the equations **(num+i)* and *num+i* show '*i*'th element and its location, respectively.

- 9.21 Write a program to access elements of an array through different ways using a pointer.

```
void main()
{
    int arr[5]={10,20,30,40,50},p=0;
    clrscr();
    for(p=0;p<5;p++)
    {
        printf("Value of arr[%d]=",p);
        printf("%d |",arr[p]);
        printf("%d |",*(arr+p));
        printf("%d |",*(p+arr));
        printf("%d |",p[arr]);
        printf("address of arr[%d]=%u\n",p,&arr[p]);
    }
}
```

OUTPUT:

```
Value of arr[0]=10 | 10 | 10 | 10 | address of arr[0]=4056
Value of arr[1]=20 | 20 | 20 | 20 | address of arr[1]=4058
Value of arr[2]=30 | 30 | 30 | 30 | address of arr[2]=4060
Value of arr[3]=40 | 40 | 40 | 40 | address of arr[3]=4062
Value of arr[4]=50 | 50 | 50 | 50 | address of arr[4]=4064
```

Explanation:

In the above program, elements are displayed using a different syntax.

- (i) *arr[p]*, (ii) **(arr+p)*, (iii) **(p+arr)*, (iv) *p[arr]*. The results of all of them would be the same.
- (i) *arr[p]*: This statement displays various array elements. Here, '*arr*' refers to the address and '*p*' refers to the element number.
- (ii) **(arr+p)*: The *arr+p* is the addition of constant with base address of the array. It shows the address of the *p*th element. The **(arr+p)* points to the *p*th element of the array.
- (iii) **(p+arr)*: This statement is the same as (ii).
- (iv) *p[arr]*: This statement is the same as (i). Here, '*p*' refers to the element number and '*arr*' refers to the base address. By varying '*p*' and '*arr*' the various elements of the array are displayed.

- 9.22 Write a program to find the sum all the elements of an array. Use the array name itself as a pointer.

```
void main()
{
    int sum=0,i=0,a[]={1,2,3,4,5};
    clrscr();
    printf("Elements Values Address\n\n");
    while(i<5)
    {
        printf("a[%d]\t%5d\t%8u\n",i,*(a+i),(a+i));
        sum=sum+*(a+i++);
    }
    printf("\nSum of Array Elements = %d",sum);
}
```

OUTPUT:

```
Elements Values Address
a[0]          1      4056
a[1]          2      4058
a[2]          3      4060
a[3]          4      4062
a[4]          5      4064
Sum of Array Elements = 15
```

Explanation:

In this program, the array name 'a' acts as a pointer and the variable 'i' is used for referring the element numbers. Using the `for` loop and expressions `*(a+i)` & `(a+i)` various elements and their addresses are displayed, respectively. In the 'sum' variable, the sum of all elements is obtained.

- 9.23 Write a program to display the sum of squares and cubes of array elements using pointer.

```
# include <math.h>
void main()
{
    int b[]={1,2,3,4,5},j,sumsq=0,sumc=0;
    clrscr();
    for (j=0;j<5;j++)
    {
        sumsq=sumsq+pow(*(j+b),2);
        sumc=sumc+pow(b[j],3);
    }
    printf("\nSum of Squares of array elements : %d",sumsq);
    printf("\nSum of Cubes of array elements : %d",sumc);
}
```

OUTPUT:

```
Sum of Squares of array elements : 55
Sum of Cubes of array elements : 225
```

Explanation:

In the above program, using the `pow()` function, square and cube of array elements are computed and added to variable 'sumsq' and 'sumc', respectively. Using the `printf()` statement, the sum of square and cube of array elements are displayed.

- 9.24 Write a program to copy the elements of one array to another using pointers.

```
void main()
{
    int so[]={10,20,30,40,50}, *pb, ds[5], i;
    pb=so;
    clrscr();
    for(i=0;i<5;i++)
    {
        ds[i]=*pb;
        pb++;
    }
    printf("Original Array Duplicated Array");
    for(i=0;i<5;i++)
        printf("\n\t%d\t\t %d", so[i],ds[i]);
}
```

OUTPUT:

```
Original Array Duplicated Array
    10          10
    20          20
    30          30
    40          40
    50          50
```

Explanation:

In the above program, pointer '*pb*' contains the base address of array `so[]`. In the `for` loop, pointer '*pb*' is assigned to the corresponding element of array `ds[]` and then incremented. After the increment, it points to the address of the next element of the array. Thus, all the elements are copied to `ds[]` array.

- 9.25 Write a program to copy one array into another array. The order of elements of the second array should be opposite of the first array.

```
void main()
{
    int arr1[]={15,25,35,45,55}, arr2[5], i;
    clrscr();
    printf("\nOriginal Array Duplicate Array");
    for(i=0;i<5;i++)
    {
        arr2[i]=*(arr1+4-i);
        printf("\n\t%d \t\t %d", arr1[i],arr2[i]);
    }
}
```


OUTPUT:

Original Array	Duplicate Array
15	55
25	45
35	35
45	25
55	15

Explanation:

The logic of the program is the same as the previous one. Here, instead of starting from the base address of the array, the address of the last element of the array is selected first. Using the `for` loop, the addresses are read in the reverse order and their contents are copied to the destination array `arr2[]`.

9.10 POINTERS AND TWO-DIMENSIONAL ARRAYS

A matrix can represent two-dimensional elements of an array. Here, the first argument is row number and second the column number. To display the address of 1st element of two-dimensional array using a pointer, it is essential to have '`&`' operator as prefix with an array name followed by element number; otherwise the compiler shows an error.

➤ 9.26 Write a program to display array elements and their addresses using pointers.

```
void main()
{
    int i,j=1,*p;
    int a[3][3]={1,2,3},{4,5,6},{7,8,9}};
    clrscr();
    printf("\tElements of An Array with their addresses\n\n");
    p=&a[0][0];
    for(i=0;i<9;i++,j++)
    {
        printf("%5d [%5u ]",*(p),p);
        p++;
        if(j==3)
        {
            printf("\n");
            j=0;
        }
    }
}
```

OUTPUT:

Elements of An Array with their addresses.

1	[4052]	2	[4054]	3	[4056]
4	[4058]	5	[4060]	6	[4062]
7	[4064]	8	[4066]	9	[4068]

Explanation:

In the above program, the two-dimensional array is declared and initialized. The base address of the array is assigned to integer pointer '*p*'. While assigning the base address of the two-dimensional array, '&' operator is to be prefixed with the array name followed by element numbers. Otherwise, the compiler shows an error. The statement `p=&a[0][0]` is used in this context. The pointer '*p*' is printed and incremented in the `for` loop till it prints the entire array elements. The `if` statement splits a line when three elements in each row are printed.

OR

- 9.27 Write a program to display array elements and their address. Use the array name itself as a pointer.

```
void main()
{
    int i;
    int a[][3]={1,2,3},{4,5,6},{7,8,9}};
    clrscr();
    printf("\tElements of An Array with their addresses.\n\n");
    for(i=0;i<9;i++)
    {
        printf("%u",&a[0][0]+i);
        printf("[%d]",*(&a[0][0]+i));
        if(i==2 || i==5)
            printf("\n");
    }
}
```

OUTPUT:

Elements of An Array with their addresses.

```
1 [4052] 2 [4054] 3 [4056]
4 [4058] 5 [4060] 6 [4062]
7 [4064] 8 [4066] 9 [4068]
```

Explanation:

The logic of the program is the same as the previous one. The only difference is that the array name itself is used as a pointer. The `if` statement inserts a line after displaying every three elements.

9.11 POINTERS AND MULTI-DIMENSIONAL ARRAYS

Array is a contiguous block of memory where multiple values are stored, i.e. elements are stored one after the other. Array name itself is a pointer (address) where the first value of the array is stored. Successive values are stored by incrementing the array name. In this multi-dimensional array also the addresses of its elements are stored contiguously. The same effect can be observed with the following program.

Normally, a two-dimensional array can be represented in the following way:

```
data-type  name_of_array[row_size][col_size];
```

Example:

```
int a[2][2];
```

Two-dimensional array with pointer notation is as follows:

```
data_type (*name_of_array)[col_size];
```

Example:

```
int (*a)[2];
```

This example represents a two-dimensional integer pointer array 'a', where each row contains two integer elements. The 'a' points to first two elements of the array which is nothing but first row of a[2][2]. Similarly, (a+1) points to the second row containing two elements. In two-dimensional array, the value of ith row and jth column is to be obtained by the following expression:

```
*(*(a+i)+j)
```

Following are a few examples on pointer and multi-dimensional arrays.

➤ 9.28 Program to display addition of the elements of three-dimensional arrays.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[2][2][2]={1,2,3,4,5,6,7,8};
    int i,j,k,b[2][2][2]={1,2,3,4,5,6,7,8};
    int c[2][2][2];
    clrscr();
    printf("\n Addition of Three Dimensional arrays as follows:");
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<2;k++)
            {
                *(*(c+i)+j)+k=*(*(a+i)+j)+k+*(*(b+i)+j)+k;
            }
        }
    }
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<2;k++)
            {
                printf("%d", *(*(c+i)+j)+k);
            }
        }
    }
    getch();
}
```

OUTPUT:

Addition of Three Dimensional arrays as follows: 2 4 6 8 10 12 14 16

- 9.29 Program to display addresses of multi-dimensional array and compute the addition of its elements.

```
int main()
{
int a[2][2][2]={1,2,3,4,5,6,7,8},i,j,k,sum=0;

clrscr();
printf("Addresses of elements of an array are as follows:\n");

for(i=0;i<2;i++)
    for(j=0;j<2;j++)
        for(k=0;k<2;k++)
        {
            printf("%u ",&a[i][j][k]);

            sum=sum+*(*(a+i)+j)+k);
        }
printf("\n Sum of all the elements of array = %d",sum);
return 0;
}
```

OUTPUT:

Addresses of elements of an array are as follows:
65506 65508 65510 65512 65514 65516 65518 65520
Sum of all the elements of array = 36

Explanation:

By refereeing to the first element using pointer (name of the array), it is easy to identify the other elements of multi-dimensional array. Successive elements are accessed one by one and the same is added with the SUM variable. Addition of all the elements of an array is done in the above program.

9.12 ARRAY OF POINTERS

So far, we have studied the arrays of different standard data types such as `int`, `float`, `char`. In the same way, the 'C' language also supports the array of pointers. It is nothing but a collection of addresses. Here, we store the addresses of variables for which we have to declare the array as pointer.

- 9.30 Write a program to store addresses of different elements of an array using array of pointers.

```
void main()
{
    int *arrp[3];
    int arr[3]={5,10,15},k;
    for(k=0;k<3;k++)
        arrp[k]=arr+k;
    clrscr();
    printf("\n\tAddress Element\n");
    for(k=0;k<3;k++)
    {
        printf("\t%u", arrp[k]);
        printf("\t%d \n", *(arrp[k]));
    }
}
```

OUTPUT:

```
Address Element
4060      5
4062     10
4064     15
```

Explanation:

In the above program, `*arrp[3]` is declared as an array of pointer. Using the first `for` loop, the addresses of various elements of array '`arr[]`' are assigned to '`*arrp[]`'. The second `for` loop picks up addresses from '`*arrp[]`' and displays the value present at that locations. Here, each element of '`*arrp[]`' points to respective element of array '`arr[]`' (see Table 9.2).

Table 9.2 Array of pointers in memory

<i>Element No.</i>	<i>Array of Values</i>	<i>Element No.</i>	<i>Array of Addresses</i>
arr[0]	5	arrp[0]	4060
arr[1]	10	arrp[1]	4062
arr[2]	15	arrp[2]	4064

- 9.31 Write a program to display the address of elements and pointers.

```
void main()
{
    int a[5]={0,1,2,3,4};
    int *p[5],i;
    for(i=0;i<5;i++)
        p[i]=a+i;
```

```

clrscr();
for(i=0;i<5;i++)
{
    printf("\n\t%d at location",*(p+i));
    printf("\t%u at location",*(p+i));
    printf("%u",p+i);
}
printf("\n\n Integer requires 2 bytes, pointer require 2 bytes");
}

```

OUTPUT:

```

0 at location 4036 at location 4046
1 at location 4038 at location 4048
2 at location 4040 at location 4050
3 at location 4042 at location 4052
4 at location 4044 at location 4054
Integer requires 2 bytes, pointer require 2 bytes

```

Explanation:

In the above program, the first `for` loop assigns addresses of the elements of integer array to the pointer array. The first `printf()` statement prints elements, the second displays addresses of the element and the third displays addresses of the address, i.e. the address of the pointer. Thus, it is clear from the above example that integer requires two bytes and the pointer requires two bytes.

9.13 POINTERS TO POINTERS

A pointer is known as a variable containing the address of another variable. The pointer variables also have an address. The pointer variable containing the address of another pointer variable is called the pointer to pointer. This chain can be continued to any extent. The below given program illustrates the concept of the pointer to pointer.

➤ 9.32 Write a program to print the value of a variable through pointer and pointer to pointer.

```

void main()
{
    int a=2,*p,**q;
    p=&a;
    q=&p;
    clrscr();
    printf("\n Value of a=%d Address of a=%u",a,&a);
    printf("\n Through *p Value of a=%d Address of a=%u",*p,p);
    printf("\n Through **q Value of a=%d Address of a=%d",**q,*q);
}

```

OUTPUT:

```

Value of a=2 Address of a=4056
Through *p Value of a=2 Address of a=4056
Through **q Value of a=2 Address of a=4056

```

Explanation:

In the above program, variable 'p' is declared as a pointer. The variable 'q' is declared as a pointer to another pointer. Hence, they are declared as '*p' and '**q', respectively. The address of a variable 'a' is assigned to the pointer 'p'. The address of pointer 'p' is assigned to 'q'. The variable 'q' contains the address of pointer variable. Hence, 'q' is a pointer to pointer. The program displays the value and address of variable 'a' using variable itself and pointers 'p' and 'q'. Table 9.3 illustrates the contents of different variables used in the program.

Table 9.3 *Pointer to pointer*

Variable	Pointer (*)	Pointer of Pointer (**)
A	P	q
2	4056	4058

- 9.33 Write a program to use different levels of array of pointer to pointer and display the elements.

```
void main()
{
    int k;
    int a[]={1,2,3};
    int *b[3];
    int **c[3];
    int ***d[3];
    int ****e[3];
    int *****f[3];
    clrscr();
    for(k=0;k<3;k++)
    {
        b[k]=a+k;
        c[k]=b+k;
        d[k]=c+k;
        e[k]=d+k;
        f[k]=e+k;
    }
    for(k=0;k<3;k++)
    {
        printf("%3d", *b[k]);
        printf("%3d", **c[k]);
        printf("%3d", ***d[k]);
        printf("%3d", ****e[k]);
        printf("%3d\n", *****f[k]);
    }
}
```

OUTPUT:

```
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
```

Explanation:

In the above example, the addresses of one-array elements are assigned to another array. The second array assigned addresses to third one and so on. In the successive `printf()` statements, `*` incremented by one for getting higher level value.

9.14 POINTERS AND STRINGS

- 9.34 Write a program to read string from keyboard and display it using character pointer.

```
void main()
{
    char name[15], *ch;
    printf("Enter Your Name :");
    gets(name);
    ch=name ;
    /* store base address of string name */
    while (*ch!='\0')
    {
        printf("%c", *ch);
        ch++;
    }
}
```

OUTPUT:

```
Enter Your Name: KUMAR
KUMAR
```

Explanation:

Here, the address of the 0th element is assigned to character pointer `'ch'`. In other words, the base address of string is assigned to `'ch'`. The pointer `'*ch'` points to the value stored at that memory location and it is printed through the `printf()` statement. After every increment of `'ch'` the pointer goes to the next character of the string. When it encounters the NULL character, the `while` loop terminates the program.

- 9.35 Write a program to find the length of a given string including and excluding spaces using pointers.

```
void main()
{
    char str[20], *s;
    int p=0, q=0;
    clrscr();
    printf("Enter String :");
    gets(str);
    s=str;
    while(*s!='\0')
    {
        printf("%c", *s);
        p++;
    }
```



```

    s++;
    if(*s==32) /* ASCII equivalent of ' ' (space) is 32 */
        q++;
}
printf("\nLength of String including spaces : %d",p);
printf("\nLength of String excluding spaces : %d",p-q);
}

```

OUTPUT:

```

Enter String: POINTERS ARE EASY
POINTERS ARE EASY
Length of String including spaces: 17
Length of String excluding spaces: 15

```

Explanation:

The above program is the same as the previous one. Here, the counter variables 'p' and 'q' are incremented to count the number of characters and spaces found in the string. The ASCII value of space is 32. Thus, at the end of the program both the variables are printed.

- 9.36 Write a program to interchange elements of character array using pointer.

```

void main()
{
    char *names[]=
    {
        "kapil",
        "manoj",
        "amit",
        "amol",
        "pavan",
        "mahesh"
    } ;
    char *tmp;
    clrscr();
    printf("Original : %s %s",names[3],names[4]);
    tmp=names[3];
    names[3]=names[4];
    names[4]=tmp;
    printf("\nNew : %s %s",names[3],names[4]);
}

```

OUTPUT:

```

Original : amol pavan
New : pavan amol

```

Explanation:

In the above program, the character array *names[] is declared and initialized. Another character pointer *tmp is declared. The destination name that is to be replaced is assigned to

the variable `'*tmp'`. The destination name is replaced with the source name and the source name is replaced with the `*tmp` variable. Thus, using simple assignment statements, two names are interchanged.

- 9.37 Write a program to read two strings through the keyboard. Compare these two strings character by character. Display the similar characters found in both the strings and count the number of dissimilar characters.

```
# include <string.h>
void main()
{
    char str1[20],str2[20],*a,*b;
    int c=0, l=0;
    clrscr();
    printf("\n Enter First String :");
    gets(str1);
    printf("\n Enter Second String :");
    gets(str2);
    a=str1;
    b=str2;
    printf("\n Similar Characters Found in Both String.");
    while(*a!='\0')
    {
        if(stricmp(*a,*b)==0)
        {
            printf("\n\t%c \t%c",*a,*b);
            l++;
        }
        else
            c++;
        a++;
        b++;
    }
    if(c==0)
        printf("\n The String are Identical.");
    else
        printf("\nThe Strings are different at %d places.",c);
    printf("\n The String Characters are similar at %d places.",l);
}
```

OUTPUT:

```
Enter First String : SUNDAY
Enter Second String : MONDAY
Similar Characters Found in Both String.
```

N	N
D	D
A	A
Y	Y

```
The Strings are different at 2 places.
The String Characters are similar at 4 places.
```

Explanation:

In the above program, two strings are entered in the character arrays `str1[]` and `str2[]`. Their base addresses are assigned to pointers `'a'` and `'b'`. In the `while` loop, two pointers are compared using the `strcmp()` function. If the characters of two strings are the same the counter variable `'l'` is incremented and characters are printed. Otherwise, the counter `'c'` is incremented. This job is done in the `if` statement. The character pointers `'a'` and `'b'` are incremented throughout the `while` loop to obtain the successive characters from both the strings. The last `if` statement displays the messages giving string are different or identical depending on the value of `'C'`.

- 9.38 Write a program to enter three characters using pointers. Use the `memcmp()` function for comparing the three characters. In case the entered characters are the same display the message 'the characters are the same', otherwise indicate their appearance before or after one another or display the status of characters in alphabetic. (The `memcmp()` function compares a specified number of characters from two buffers.)

```
# include <string.h>
# include <process.h>
void main()
{
    char x,y,z,*xp,*yp,*zp;
    int stat=0;
    clrscr();
    printf("Enter Three Characters");
    scanf("%c %c %c",&x,&y,&z);
    xp=&x,yp=&y,zp=&z;
    stat = memcmp(yp, xp, strlen(yp));
    if(*xp==*yp)
    {
        printf("\n1st and 2nd Character are same.\n");
        goto next;
    }
    if(stat > 0)
        printf("2nd Character appears after the 1st Character in Alphabetic.\n");
    else
        printf("2nd Character appears before the first Character in Alphabetic \n");
    next:
    stat = memcmp(yp,zp, strlen(yp));
    if(*yp==*zp)
    {
        printf("\n2nd and 3rd Character are same.");
        exit(1);
    }
    if(stat > 0)
        printf("2nd Character appears after the 3rd Character in Alphabetic. \n");
    else
```

```

    printf("2nd Character appears before 3rd Character in  

    Alphabetic.\n");
}

```

OUTPUT:

```

Enter Three Character C C A
1st and 2nd Character are same.
2nd Character appears after the 3rd Character in Alphabetic.

```

Explanation:

In the above program, three characters are entered in the character variables 'x', 'y' and 'z' and their base addresses are stored in the pointers 'xp', 'yp' and 'zp', respectively. The function `memcmp()` is used to compare two pointers for a specified length. If the first two characters are the same, a message is displayed and control goes to the *next* label. In the *next* label, the second and third characters are compared. Using the value of the variable 'stat' locations of characters are decided.

- 9.39 Write a program to compare two strings irrespective of case. Compare the characters at the specific position. If they are the same display 'the characters are the same at that position'.

```

void main()
{
    char *buf1 = "computer";
    char *buf2 = "comp ter";
    int stat;
    stat = memcmp(buf1, buf2, 4);
    clrscr();
    printf("The Characters up to 4th position are");
    if (stat) /* if stat is non zero then prints 'not' otherwise
    'same' .t */
    printf("not");
    printf("same\n");
}

```

OUTPUT:

```

The Characters up to 4th position are same.

```

Explanation:

The `memcmp()` function compares two strings for a specified number of characters. It returns zero if both the strings are the same up to a specified length of characters. Otherwise, the non-zero value will be returned. Depending upon the value it returns, the if statement displays respective messages.

- 9.40 Write a program to read a string. Print the string up to the first occurrence of the character entered through the keyboard.

```
# include <string.h>
void main()
{
    char src[20],dest[50],*ptr,*sr,f;
    clrscr();
    printf("\n Enter a String :");
    gets(src);
    printf("\n Enter a Character to find in the text :");
    scanf("%c",&f);
    sr=src;
    ptr = memccpy(dest, sr, f, strlen(sr));
    if(ptr)
    {
        *ptr = '\0';
        printf("String up to that Character : %s\n", dest);
    }
    else
        printf("The character wasn't found\n");
}
```

OUTPUT:

```
Enter a String : FUNCTIONS
Enter a Character to find in the text : T
String up to that Character : FUNCT
```

Explanation:

The `memccpy()` function copies the number of characters from the source string up to the first occurrence of a given character. It returns a pointer if the given character is found, otherwise it returns `NULL`.

- 9.41 Write a program to read two strings through the keyboard. Replace the contents of the second string with the first string. The length of the first string should be less than that of the second string.

```
# include <string.h>
void main()
{
    static char src[20],dest[20];
    char *ptr;
    clrscr();
    printf("\nEnter a Source String:");
```

```

gets(src);
printf("\nEnter a Destination String :");
gets(dest);
printf("\n\nDestination before memcpy: %s\n", dest);
ptr = memcpy(dest, src, strlen(src));
if(ptr)
printf("Destination after memcpy : %s\n", dest);
else
printf("memcpy failed\n");
}

```

OUTPUT:

```

Enter a Source String: Tomorrow
Enter a Destination String: Today is Sunday
Destination before memcpy : Today is Sunday
Destination after memcpy : Tomorrow is Sunday

```

Explanation:

The function `memcpy()` works similarly as in the previous program. It copies the source string to the destination string. Both the strings are displayed.

- 9.42 Write a program to display the string through their pointer.

```

void main()
{
    char *c,*m;
    c="Central Processing Unit";
    m="Math Co- Proccesser";
    clrscr();
    printf("'c' is pointing the string '%s'\n",c);
    printf("'m' is pointing the string '%s'\n",m);
    getch();
}

```

OUTPUT:

```

'c' is pointing the string 'Central Processing Unit'
'm' is pointing the string 'Math Co-Processor'

```

Explanation:

In the above program, two strings are initialized to the character pointers 'c' and 'm'. When printed using '%s' control string in `printf()` statement, entire strings are displayed.

SUMMARY

This chapter describes the most important feature of the C language, i.e. pointer. In this chapter, we have discussed declaration and initialization of pointers. You have studied how to access variables using their pointers. After having gone through the topic you are familiar with the effect of unary operators on pointers of different data types as well as arithmetic operations with pointers. You are also familiar with array of pointers and relation of pointer with arrays of different dimensions. Now you also know how to make a chain of pointers, i.e. how one-pointer points to another pointer. Finally, the association of strings with a pointer is explained.

EXERCISES

I True or false:

1. Every variable is associated with an address in memory.
2. All the addresses in the memory are unique.
3. The address of a variable cannot be stored in another variable.
4. The pointer is a special type of variable which holds the value of address of other variable.
5. The pointer is a variable and cannot be made constant.
6. The size of a pointer variable is the same as that of standard size of its data type.
7. One type of pointer cannot hold the data with other data type.
8. The pointer variable is also stored somewhere in memory.
9. We cannot store the address of the pointer variable.
10. If we want to change the value of variable passed as an argument to the function inside that function then we should pass the address of the variable.
11. If we increment pointer, it will point to the next byte in memory.
12. The segment `char *a[]={ "ABC", "def" };` is invalid.
13. Pointer of any data type occupies the same memory space.
14. With indirection operator (*), the value of the variable stored at some address can be accessed.
15. The allocation of memory during program run time is called dynamic memory allocation.
16. With pointer, data is manipulated with the address.
17. Pointers are not associated with multi-dimensional arrays.
18. Pointers are used to allocate memory dynamically.
19. Value of the `CONST` pointer `*pm` can be changed.
20. Arithmetic operations on pointer variables are not possible.

II Match the functions /words given in Group A with meanings in Column B:

Sr. No	Data Type	Sr. No	Required Bytes
1	<code>int i=2</code>	A	4
2	<code>char c='x'</code>	B	4
3	<code>float f=2.2</code>	C	1
4	<code>long l=2</code>	D	2

Sr. No	Declaration	Sr. No	Declaration
1	int *x;	A	Array
2	float **x;	B	Pointer of pointer
3	long *x	C	Integer pointer
4	int x[5]	D	Long pointer

III Select the appropriate option from the multiple choices given below:

- Which of the following statement is true after the execution of following program?

```
int a[5]={2,3}, *c;
c=a;
(*c)--;
```

 - the value of a[0] will be 1
 - the value of a[0] will be 2
 - the value of a[1] will be 2
 - None of the above
- The fastest way to exchange two rows in a two-dimensional array is
 - exchange the addresses of each element in the two rows
 - exchange the elements of the two rows
 - store the addresses of the rows in an array of pointers and exchange the pointers
 - None of the above
- Which is the correct way to declare a pointer?
 - int *ptr
 - *int ptr
 - int ptr*
 - int_ptr x
- What will be the result of the following program?

```
void main()
{
    int a=8,b=2,c,*p;
    c=(a=a+b,b=a/b,a=a*b,b=a-b);
    p=&c;
    clrscr();
    printf("\n %d",++*p);
}
```

 - 50
 - 46
 - 36
 - 40
- What will be the resulting string after the execution of following program?

```
#include<string.h>
void main()
{
    char *str1, *str2, *str3;
    str1="The Capital of India is";
    str2="!!ihleD weN";
    str3="Bangalore";
    strncat(str1, strrev(str2),
    strlen(str3));
    clrscr();
    puts(str1);
}
```

 - The Capital of India is Mumbai
 - The Capital of India is New Delhi
 - The Capital of India is Bangalore
 - None of the above
- What will be the values of variables a and b after execution ?

```
#include<string.h>
void main()
{
    int a,*b=&a,**c=&b;
    a=5;
    **c=15;
    *b= **c;
    clrscr();
    printf("A=%d, B=%d",a,*b);
}
```

 - A=15, B=15
 - A=15, B=5
 - A=15, B=16
 - None of the above

7. What will be the value of the variables *a1* and *a2* after execution?

```
void main()
{
    int a1,a2,c=3,*pt;
    pt=&c;
    a1=3*(c+5);
    a2=3*( *pt+5);
    printf("\n a1=%d, a2=%d", a1,a2);
}
```

- (a) $a_1=24, a_2=24$
- (b) $a_1=12, a_2=24$
- (c) $a_1=12, a_2=20$
- (d) None of the above

8. What will be the value of *x* after execution of the following program?

```
void main()
{
    int x,*p;
    p=&x;
    *p=2;
    clrscr();
    printf("\nValue of x=%d",x);
}
```

- (a) $x=2$
- (b) $x=0$
- (c) $x=65504$
- (d) None of the above

IV Attempt the following programs:

- Write a program to accept a string using character pointer and display it.
- Write a program to calculate square and cube of the entered number using pointer of the variable containing entered number.
- Write a program to display all the elements of an array using pointer.
- Write a program to allocate memory for 10 integers.
- Write a program to demonstrate the use of `realloc()` functions.

V Answer the following questions:

- What are pointers? Why are they important?
- Explain features of pointers.
- Explain pointer of any data type that requires four bytes.
- Explain the use of (*) indirection operator.
- Explain the effect of ++ and — operators with pointer of all data type.
- What is an array of pointer? How is it declared?
- Explain the relation between an array and a pointer.
- Why addition of two pointers is impossible?
- Which are the possible arithmetic operations with pointers?
- Explain comparison of two pointers.
- How one pointer points to another pointer?
- How will you recognize pointer to pointer? What does the number of '*'s indicate?
- How strings are stored in pointer variables? Is it essential to declare length?
- What is base address? How is it accessed differently for one and two-dimensional arrays?
- Elaborate the address stored in the pointer and the value at that address.
- Why element counting of arrays always starts from '0'?
- Write a program to read and display a two-dimensional array of 5 by 2 numbers. Reduce the base address of an array by one and start element counting from one.

18. Explain tiny and large memory models.
19. What are the uses of `malloc()` and `free()` functions?
20. Explain the concept of dynamic memory allocation.
21. Describe the various functions used for memory allocation.

VI What is/are the output/s of the following programs?

1. Use pointer and display value and address of an element

```
void main()
{
    int num=10, *p;
    p=&num;
    clrscr();
    printf("%d", *p);
    printf("%d", num);
    printf("%u", &num);
    printf("%u", p);
    getch();
}
```

2. Program on display of a string

```
void main()
{
    char c[]="India";
    char *p;
    p=c;
    clrscr();
    printf("%s", p);
}
```

3. Display two strings

```
void main()
{
    char *c="India";
    char *p="Bharat";
    char *k;
    k=p;
    p=c;
    clrscr();
    printf("%s", p);
    printf("%s", k);
}
```

4. Use pointer and display values of an element by various ways

```
void main()
{
    int v=10, *p;
```

```
    p=&v;
    clrscr();
    printf("v=%d v=%d v=%d", v, *p,
    *(&v));
}
```

5. Use pointers and display addition of two numbers with and without pointers

```
void main()
{
    int a=5, b=7, c, d, *ap, *bp;
    clrscr();
    ap=&a;
    bp=&b;
    c=a+b;
    d=*ap+*bp;
    printf("\n Sum of A & B : %d", c);
    printf("\n Sum of A & B : %d", d);
}
```

6. Addition, Subtraction, Multiplication, Division & Modulus operation on pointers.

```
void main()
{
    int a=25, b=10, *p, *j;
    p=&a; j=&b;
    clrscr();
    printf("%d %d %d %d %d", *p+b, *p-
    b, *p * *j, *p / *j, *p % *j);
}
```

7. Display numbers

```
void main()
{
    int x[]={1, 2, 3, 4, 5, 6, 7}, k=1;
    clrscr();
    while(k<=5)
    {
        printf("%2d", k[x-1]); k++;
    }
}
```

VII Find the bug/s in the following programs:

1.

```
void main()
{
    char *c;
    float x=10;
    c=&x;
    printf("%d", *c);
}
```

2.

```
void main()
{
    float x=10.25, *f; f=x; clrscr();
    printf("%g", *f);
}
```

3.

```
void main()
{
    float x=10.25, *t, *f;
    t=&x;
    f=&t;
    clrscr();
    printf("%g", **f);
}
```

4.

```
void main()
{
    char sa[]="How are you ?";
    char t, *f; f=&sa; clrscr();
    printf("%s", f);
}
```

5.

```
void main()
{
    int *t, x; t=&x; x=11;
    *t++;
    clrscr();
    printf("%d", *t);
}
```

6.

```
void main()
{
    int t[]={1, 2, 3, 4, 5};
    int *p, *q, *r; p=t; q=p[1];
    r=p[2];
    clrscr();
    printf("%d %d %d", *p, *q, *r);
}
```

7.

```
void main()
{
    int num[2][3]={1, 2, 3, 4, 5}, *k;
    k=&num;
    clrscr();
    printf("%d", *k);
}
```

8.

```
void main()
{
    int x=5, y=8, z;
    int *px, *py, *pz; px=&x; py=&y; p
    z=&z; pz=*px+*py;
    clrscr();
    printf("%d", z);
}
```

9.

```
void main()
{
    int x=5;
    int *px=&x, *py;
    clrscr();
    printf("%d", px);
}
```

ANSWERS

I True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	T	2.	T	3.	F	4.	T	5.	F
6.	F	7.	F	8.	T	9.	F	10.	T
11.	F	12.	F	13.	T	14.	T	15.	T
16.	T	17.	T	18.	T	19.	F	20.	F

II Match the functions/words given in Group A with meanings in Column "B":

Q.	Ans.	Q.	Ans.
1.	D	1.	C
2.	C	2.	B
3.	A,B	3.	D
4.	A,B	4.	A

III Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	a	4.	b	5.	b
6.	a	7.	a	8.	a				

VI What is/are the output/s of the following programs?

Q.	Ans.	Q.	Ans.
1.	10 10 65496 65496	5.	Sum of A & B : 12 Sum of A & B : 12
2.	India	6.	35 15 250 2 5
3.	India Bharat	7.	1 2 3 4 5
4.	v=10 v=10 v=10		

VII Find the bug/s in the following programs:

Q.	Ans.
1.	pointer pointing to a variable should be of same type.
2.	& is missing is assignment statement.

3. `**f` should be declared as pointer of pointer.
4. No Bug.
5. `t++` dose not work (`++*t`) works.
6. `&` is required in `q=p[1];` & `r=p[2];`.
7. `k=#` would be `k=&num[0][0];`.
8. In the assignment statement `*pz` required.
9. `%u` is required for formatting address.

CHAPTER

10 Functions

Chapter Outline

- 10.1** Introduction
- 10.2** Basics of a Function
- 10.3** Function Definition
- 10.4** The return Statement
- 10.5** Types of Functions
- 10.6** Call by Value and Reference
- 10.7** Function Returning More Values
- 10.8** Function as an Argument
- 10.9** Function with Operators
- 10.10** Function and Decision Statements
- 10.11** Function and Loop Statements
- 10.12** Functions with Arrays and Pointers
- 10.13** Passing Array to a Function
- 10.14** Nested Functions
- 10.15** Recursion
- 10.16** Types of Recursion
- 10.17** Rules for Recursive Function
- 10.18** Direct Recursion
- 10.19** Indirect Recursion
- 10.20** Recursion Versus Iterations
- 10.21** The Towers of Hanoi
- 10.22** Advantages and Disadvantages of Recursion
- 10.23** Efficiency of Recursion
- 10.24** Library Functions

10.1 INTRODUCTION

It is difficult to prepare and maintain a large-sized program. Moreover, the identification of the flow of data is hard to understand. The best way to prepare a programming application is to divide the bigger program into small pieces or modules and the same modules can be repeatedly called from the `main()` function as and when required. These small modules or subprograms are easily manageable. This method is called the divide and conquer method. In C, such small modules are called functions. Figure 10.1 describes the method of developing a program.

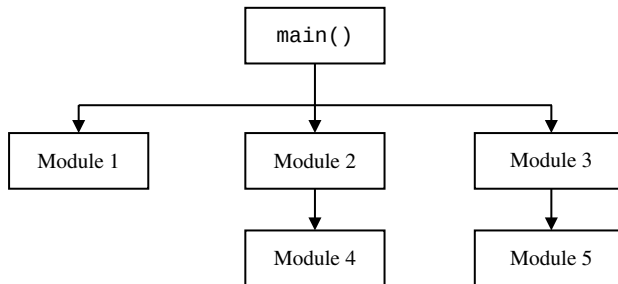


Figure 10.1 A program divided in multiple functions

The programs written in the C language are highly dependent on functions. The C program is nothing but a combination of one or more functions. Every C program starts with the user-defined function `main()`. Each time when a new program is started, `main()` function must be defined. The `main()` function calls another functions to share the work. The `main` program can supply data to a function. A specific operation is performed on the data and a value is returned to the calling function.

The C language supports two types of functions: (1) library functions and (2) user-defined functions. The library functions are pre-defined set of functions. Their task is limited. A user should not worry to understand the internal working of these functions. One can only use the functions but cannot change or modify them. For example, `sqrt(81)` gives result 9. Here, the user need not worry about its source code, but the result should be provided by the function.

The user-defined functions are totally different. The functions defined by the user according to his/her requirement are called user-defined functions. A user can modify the functions according to the requirement. A user certainly understands the internal working of the function. The user has full scope to implement his/her own ideas in the function. Thus, the set of such user-defined functions can be useful to another programmer. One should include the file in which the user-defined functions are stored to call the function in the program. For example, let `square(9)` be a user-defined function that gives the result 81. Here, the user knows the internal working of the `square()` function, as its source code is visible. This is the major difference between the two types of functions.

10.2 BASICS OF A FUNCTION

A function is a self-contained block or a sub-program of one or more statements that perform a special task when called.

10.2.1 | Why Use Functions?

- (a) If we want to perform a task repetitively, then it is not necessary to re-write the particular block of the program again and again. Shift the particular block of statements in a user-defined function. The function defined can be called any number of times to perform the task.

- (b) Using functions, large programs can be reduced to smaller ones. It is easy to debug and find out the errors in it. It also increases readability.

10.2.2 | How a Function Works?

- (a) Once a function is defined and called, it takes some data from the calling function and returns a value to the called function.
- (b) The detail of inner working of a function is unknown to the rest of the program. Whenever a function is called, control passes to the called function and working of calling function is paused. When the execution of a called function is completed, control returns back to the calling function and executes the next statement.
- (c) The values of actual arguments passed by the calling function are received by the formal arguments of the called function. The number of actual and formal arguments should be the same. Extra arguments are discarded if they are defined. If the formal arguments are more than the actual arguments, then the extra arguments appear as garbage. Any mismatch in the data type will produce the unexpected result.
- (d) The function operates on formal arguments and sends back the result to calling function. The `return()` statement performs this task.

10.3 FUNCTION DEFINITION

Function definition is as per the format given below.

```
return_type function_name (argument/parameter list)
{
    local variable declaration;
    Statement1;
    Statement2;
    return(value);
} /*Body of the function definition*/
```

The following code illustrates the working of a function with its necessary components (Table 10.1).

Table 10.1 Working of a function

<pre>int abc(int, int, int); void main() { int x, y, z; ----- ----- abc(x,y,z); /* Function Call */ { -----> Actual arguments } ----- ----- }</pre>	

(Continued)

```

int abc(int l, int k, int j) Function definition
{
    -----
    -----
    return(); return value
}

```

→ Formal arguments

- (a) The **return** type of the function can be `int`, `float`, `double`, `char`, `void`, etc., depending upon what is to be returned from called function to the calling function. By default in ANSI C, function returns `int`, if return type is not mentioned.
- (b) **Actual argument:** The arguments of calling functions are actual arguments. In Table 10.1, variables 'x', 'y' and 'z' are actual arguments.
- (c) **Formal argument:** The arguments of the called function are formal arguments. In Table 10.1, variables 'l', 'k' and 'j' are formal arguments.
- (d) **Function name:** A function name follow the same rule as we use for naming a variable.

Example:

```
sum (int a, int b);
```

where `SUM()` is a user-defined function. This is a function call and 'a' and 'b' are integer arguments. The function call must be ended by a semi-colon (;).

- (e) **Argument/parameter list:** The argument list means variable names enclosed within the parentheses. They must be separated by a comma (.). These formal arguments (consignment) receive values from the actual argument for performing the communication between consignee and consignor functions.
- (f) **Function call:** A compiler executes the function when a semi-colon (;) is followed by the function name. A function can be called simply using its name like other C statement, terminated by a semi-colon (;).

Consider the following example.

➤ 10.1 Write a program to show how user-defined function is called.

```

int add(int,int); /* function prototype */
void main()
{
    int x=1,y=2,z;
    z=add(x,y); /*      FUNCTION call/
    printf("z=%d",z);
}
/* FUNCTION DEFINITION */
int add(int a, int b)
{
    return(a+b);
}

```

OUTPUT:

```
z=3
```

Explanation:

In the above program, values of 'x' and 'y' are passed to function `add()`. Formal arguments 'a' and 'b' receive the values. The function `add()` calculates the addition of both the variables and returns result. The result is collected by the variable 'z' of the `main()` function which is printed through the `printf()` statement.

- 10.2 Write a program to define a user-defined function. Call them at different places.

```
void y();
void y()
{
    printf("Y");
}
void main()
{
    void a(),b(),c(),d();
    clrscr();
    y();
    a();
    b();
    c();
    d();
}
void a()
{
    printf("A");
    y();
}
void b()
{
    printf("B");
    a();
}
void c()
{
    a();
    b();
    printf("C");
}
void d()
{
    printf("D");
    c();
    b();
    a();
}
```

OUTPUT:

Y A Y B A Y A Y B A Y C D A Y B A Y C B A Y A Y

Explanation:

From the above program we can conclude that.

- (i) The `main()` function can call any other function defined before or after the `main()` function.
- (ii) Any other user-defined function can call the `main()` function.
- (iii) The user-defined function can call only those user-defined functions which are defined before it, i.e. the function `a()` can call only `y()` and not `b()`, `c()` and `d()`. The function `b()` cannot call functions `c()` and `d()`. The function `d()` can call all the functions because it is the last function.
- (f) Local and global variables: There are two kinds of variables: (i) local and (ii) global.
- (i) Local variable: The local variables are defined within the body of the function or block. The variable defined is local to that function or block only. Other functions cannot access these variables. The compiler shows errors in case other functions try to access the variables.

Example:

```
value(int k, int m)
{
    int r,t;
}
```

Here '*r*' and '*t*' are the local variables, which are defined within the body of the function `value()`. The same variable names may be defined in different functions. They are local to that particular function.

Programs on local and global variables are illustrated below.

- 10.3 Write a program to show how similar variable names can be used in different functions.

```
void fun(void);
void main()
{
    int b=10,c=5;
    clrscr();
    printf("\n In main() B=%d C=%d",b,c);
    fun();
}
void fun()
{
    int b=20,c=10;
    printf("\nIn fun() B=%d C=%d",b,c);
}
```

OUTPUT:

```
In main() B=10 C=5
In fun()  B=20 C=10
```

Explanation:

In the above program, two functions are used. One is `main()` and the other user-defined function is `fun()`. The variables '*b*' and '*c*' are defined in both the functions. Their effect is only within the function in which they are defined. Both the functions print local values of '*b*' and '*c*' assigned in their functions.

We can also declare the same variable names for actual and formal arguments. The compiler will not get confused due to same variable names. The scope of every auto variable is local to the block in which they are defined.

- (ii) Global variables: Global variables are defined outside the `main()` function. Multiple functions can use them. The example is illustrated below for understanding.

➤ 10.4 Write a program to show the effect of global variables on different functions.

```
void fun(void);
int b=10,c=5;
void main()
{
    clrscr();
    printf("\nIn main() B=%d C=%d",b,c);
    fun();
    b++;
    c--;
    printf("\n Again In main() B=%d C=%d",b,c);
}
void fun()
{
    b++;
    c--;
    printf("\nIn fun() B=%d C=%d",b,c);
}
```

OUTPUT:

```
In main()          B=10 C=5
In fun()           B=11 C=4
Again In main()   B=12 C=3
```

Explanation:

In the above program, variables '*b*' and '*c*' are defined outside the `main()`. Hence, they are global. They can be called by any other function. The same variables '*b*' and '*c*' are used throughout the program. In `main()`, values of '*b*' and '*c*' are printed.

The function `fun()` is called; '*b*' is incremented and '*c*' is decremented. The values of '*b*' and '*c*' are printed. The control then returns to the function `main()`.

Again '*b*' is incremented and '*c*' is decremented. The values of '*b*' and '*c*' are printed. Here, both the functions use the same variables.

- (g) Return value: It is the outcome of the function. The result obtained by the function is sent back to the calling function through the `return` statement. The `return` statement returns one value per call. The value returned is collected by the variable of the calling function.

10.4 THE return STATEMENT

The user-defined function uses the `return` statement to return the value to the calling function. Exit from the called function to the calling function is done by the use of the `return` statement. When the `return` statement is executed without argument, it always return 1.

➤ 10.5 Write a program to use the return statement in different ways.

```

int pass(int);
void main()
{
    int x,y;
    clrscr();
    printf("Enter value of x :");
    scanf("%d",&x);
    if(x==1 || x<0)
        return 0;
    y=pass(x);
    switch(y)
    {
        case 1:
            printf("The value returned is %d",y);
            break;
        default :
            printf("The Cube of %d is : %d",x,y);
    }
    return 0;
}
int pass(a)
{
    if(a==0)
        return 0;
    else
        return(a*a*a);
}

```

OUTPUT:

```

Enter value of x : 5
The Cube of 5 is :125

```

- (a) **return 0:** This statement returns zero to the operating system if the value entered by the user is 1 or negative.
- (b) **return:** The value of 'x' is passed to the function `pass()`. If the value is zero then the **return** statement returns 1. The **return** statement is used to exit from the function `pass()`.
- (c) **return (a*a*a):** In a function `pass()`, when the **if** statement finds the value of 'x' as non-zero then **else** block is executed and the **return** statement returns the cube of the entered number.

The return statement can be used in the following ways.

- (a) `return(expression);`

Example:

```
return(a+b+c);
```

If such a statement is executed, the expression within the parentheses is first solved and the result obtained is returned.

- (b) A function may use one or more **return** statements. It is used when we want to return a value depending upon certain conditions.

Example:

The **if statement**

```
if(a>b)
return(a);
else
return(b);
```

The **switch() statement**

```
switch(x)
{
    case 1 :
        return(1);
        break;
    case 2 :
        return(1*2);
        break;
    default :
        return(0);
}
```

- (c) **return(&p);**

If above syntax is used, it returns the address of the variable.

- (d) **return(*p);**

The above statement returns the value of a variable through the pointer.

- (e) **return(sqrt(r));**

If such a format is used, when control reaches to the return statement, control again passes to function **sqrt()**. The return statement collects the result obtained from **sqrt()** function and returns it to the calling function. Here, the return statement calls the function **sqrt()**.

- (f) **return(float(square(2.8)));**

All functions return by default integer value. To force the function return other type of value, specify the data type to be used. When a value is returned, it automatically converts to the function data type. In the above example, the return value will be converted to **float** type.

From the above discussion we can conclude the following:

- (1) When the **return** statement is encountered in the function the control sends back to the calling function and next statement following **return** statement if present will not be executed.
- (2) The absence of the return statement indicates that no value is returned. Such functions are called a **void**.

10.5 TYPES OF FUNCTIONS

Depending on arguments present, the return value sends the result back to the calling function. Based on this, the functions are divided into four types.

- (1) **Without arguments and return values (see Table 10.2).**

Table 10.2 Functions without arguments and return values

Calling Function	Analysis	Called Function
void main() { ----- ----- abc(); ----- ----- }	No arguments are passed. No values are sent back.	abc() { ----- ----- ----- }

- (1) Data is neither passed through the calling function nor sent back from the called function.
- (2) There is no data transfer between calling and called functions.
- (3) The function is only executed and nothing is obtained.
- (4) If such functions are used to perform any operation, they act independently. They read data values and print result in the same block.
- (5) Such functions may be useful to print some message, draw a line or spilt the line.

Example:

➤ 10.6 Write a program to display a message using user-defined function.

```
void main()
{
    void message();
    message();
}
void message()
{
    puts("Have a nice day");
}
```

OUTPUT:
Have a nice day

Explanation:

This program contains a user-defined function named `message()`. It requires no arguments and returns nothing. It displays only a message when called.

➤ 10.7 Write a program to display alphabets ‘A’, ‘B’ and ‘C’ using functions.

```
void main()
{
    void a(), b(),c();
    a();
    b();
    c();
}
```

```
void a()
{
    printf("\nA");
}
void b()
{
    printf("B");
}
void c()
{
    printf("C");
}
```

OUTPUT:

ABC

Explanation:

In the above example, three user-defined functions `a()`, `b()` and `c()` are defined. The `main()` function calls these three functions. These functions get called as per the calling sequence and messages are displayed.

(2) With arguments but without return values (Table 10.3).

Table 10.3 Function with arguments but without return values

Calling Function	Analysis	Called Function
void main() { ----- ----- abc(x); ----- ----- }	Argument(s) are passed. No values are sent back.	abc(y) { ----- ----- ----- } -----

- (1) In the above functions, arguments are passed through the calling function. The called function operates on the values. But no result is sent back.
- (2) Such functions are partly dependent on the calling function. The result obtained is utilized by the called function and there is no gain to the `main()`.

Example:

- 10.8 Write a program to pass date, month and year as parameters to a user-defined function and display the day in the format dd/mm/yy.

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

void main()
{
```



```

int  dat(int ,int ,int );
int  d,m,y;
clrscr();
printf("\nEnter Date: dd/mm/yy");
scanf("%d %d %d",&d,&m,&y);
dat(d,m,y);
getch();
}

int dat(int x, int y, int z)
{
printf("\nDate = %d/%d/%d",x,y,z);
return 0;
}

```

OUTPUT:

```

Enter Date: dd/mm/yy09 09 2014
Date = 9/9/2014

```

Explanation:

In this program, three value are passed to the `dat ( )` function. The `dat ( )` function receives arguments form `main ( )` and displays date. But it returns nothing.

- 10.9 Write a program to calculate square of a number using user-defined function.

```

void main()
{
    int j=0;
    void sqr(int);
    clrscr();
    for(j=1;j<5;j++)
        sqr(j);
}

void sqr(int k)
{
    printf("\n%d", k*k);
}

```

OUTPUT:

```

1
4
9
16

```

Explanation:

Here, in this program, the `main ( )` function passes one argument per call to the function `sqr ( )`. The function `sqr ( )` collects this argument and prints its square. The function `sqr ( )` is void.

➤ 10.10 Write a program to pass the value to `main()` function.

```
void main(int j)
{
    clrscr();
    printf("\n Number of command line arguments J = %d",j);
}
```

Explanation:

To execute this program, first create its executable file and execute the program from the command prompt. The above example illustrates the command line arguments.

```
C:\> main.exe 1 2 3
```

In the above example, `main.exe` is the file name containing the program code. The file is executed from the command prompt. Following the file name some arguments are written. The variable '`j`' contains the number of arguments supplied from the command prompt. Here, four arguments are supplied to the file name. Thus, the `main()` function receives the arguments and returns nothing.

(3) With arguments and return values (Table 10.4).

Table 10.4 Function with arguments and return values

<code>void main()</code>		<code>abc(y)</code>
<code>{</code>		<code>{</code>
<code>int z;</code>		-----
-----		<code>y++;</code>
<code>z=abc(x);</code>	Argument(s) are passed.	-----
-----		-----
-----		-----
<code>}</code>	Values are send back.	<code>return(y);</code>
		<code>}</code>

- (1) In the above example, the copy of the actual argument is passed to the formal argument, i.e. the value of '`x`' is assigned to '`y`'.
 - (2) The return statement returns the incremented value of '`y`'. The returned value is collected by '`z`'.
 - (3) Here, data is transferred between calling and called functions, i.e. communication between functions is made.
- 10.11 Write a program to pass values to a user-defined function and display the date, month and year.

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

void main()
```

```

{
int dat(int ,int ,int );
int d,m,y,t;
clrscr();
printf("Enter Date dd/mm/yy : ");
scanf("%d %d %d",&d,&m,&y);
t=dat(d,m,y);
printf("\nTomorrow = %d/%d/%d",t,m,y);
getch();
}

int dat(int x, int y, int z)
{
printf("\nToday      = %d/%d/%d",x,y,z);
return(++x);
}

```

OUTPUT:

```

Enter Date dd/mm/yy : 09 09 2014
Today      = 9/9/2014
Tomorrow = 10/9/2014

```

Explanation:

In the above program, three values date, month and year are passed to the function `dat()`. The function displays date. The function `dat()` returns the next date. The next date is printed in function `main()`. Here, function `dat()` receives arguments and return values.

- 10.12 Write a program to send values to user-defined function and receive and display the return value.

```

int main()
{
    int sum(int,int,int), a,b,c,s;
    clrscr();
    printf("Enter Three Numbers :");
    scanf("%d %d %d", &a,&b,&c);
    s=sum(a,b,c);
    printf("Sum = %d",s);
    return 0;
}

int sum(int x, int y, int z)
{
    return(x+y+z);
}

```

OUTPUT:

```

Enter Three Numbers : 7 5 4
Sum = 16

```

Explanation:

In the above program, the function `sum()` receives three values from the function `main()`. The `sum()` calculates the sum of three numbers and returns them to `main()`.

(4) Without arguments and but with return values (Table 10.5)

Table 10.5 Functions without arguments but with return values

main()		abc()
{		{
int z;		int y=5;
-----		-----
z=abc();	No Argument(s) are passed.	-----
-----		-----
-----	Values are sent back.	return(y);
}		}

- (1) In the above type of function, no argument(s) is passed through the `main()` function. But the called function returns values.
 - (2) The called function is independent. It reads values from keyboard or generates from initialization and returns the values.
 - (3) Here, both the calling and called functions partly communicate with each other.
- 10.13 Write a program to receive values from user-defined function without passing any value through `main()`.

```
int main()
{
    int sum(), a, s;
    clrscr();
    s=sum();
    printf("Sum = %d", s);
    return 0;
}

int sum()
{
    int x, y, z;
    printf("\n Enter Three Values :");
    scanf("%d %d %d", &x, &y, &z);
    return(x+y+z);
}
```

OUTPUT:

```
Enter Three Values : 3 5 4
Sum = 12
```

Explanation:

The user-defined function `sum()` is an independent function. The function `sum()` reads three values through the keyboard and returns their sum to the function `main()`. The `main()` only prints the sum.

- 10.14 Write a program to read values through the user-defined function and send back the result in the form of an address.

```
int main()
{
    int *sum(), *s;
    clrscr();
    s=sum();
    printf("Sum = %d", *s);
    return 0;
}

int *sum()
{
    int x,y,z,k;
    printf("\nEnter Three Values :");
    scanf("%d %d %d",&x,&y,&z);
    k=x+y+z;
    return(&k);
}
```

OUTPUT:

```
Enter Three Values : 1 3 5
Sum = 9
```

Explanation:

The function `sum()` is declared as a pointer function, i.e. the function declared as a pointer always returns reference. The reference returned by the function `sum()` is assigned to pointer `*s`. The pointer `*s` prints the sum.

10.6 CALL BY VALUE AND REFERENCE

There are two ways in which we can pass arguments to the function.

- (i) **Call by value:** In this type, the value of actual arguments is passed to the formal arguments and operation is done on the formal arguments. Any change in the formal argument made does not affect the actual arguments because formal arguments are the photocopy of the actual argument. Hence, when a function is called by the call by value method, it does not affect the actual contents of the arguments. Changes made in the formal arguments are local to the block of the called function. Once control returns back to the calling function, changes made vanish. The example given below illustrates the use of call by value.
- 10.15 Write a program to exchange values of two variables by using 'call by value' to the function.

```
int main()
{
    int x,y,change (int, int);
    clrscr();
    printf("\n Enter Values of X & Y :");
    scanf("%d %d",&x,&y);
```

```

    change(x,y);
    printf("\n In main() X=%d Y=%d",x,y);
    return 0;
}
change(int a, int b)
{
    int k;
    k=a;
    a=b;
    b=k;
    printf("\n In Change() X=%d Y=%d",a,b);
}

```

OUTPUT:

```

Enter Values of X & Y : 5 4
In Change() X=4 Y=5
In main()    X=5 Y=4

```

Explanation:

In the above program, we are passing values of actual argument 'x' and 'y' to the function `change()`. The formal arguments 'a' and 'b' of function `change()` receive these values(). The values are interchanged, i.e. the value of 'a' is assigned to 'b' and vice versa and printed. When the control returns back to the `main()`, the changes made in the function `change()` vanish. In the `main()`, the values of 'x' and 'y' are printed as they read from the keyboard. In the call by value method, the formal argument acts as a photocopy of the actual argument. Hence, changes made to them are temporary.

- (ii) Call by reference: In this type, instead of passing values, addresses (reference) are passed. Function operates on addresses rather than values. Here, the formal arguments are pointers to the actual argument. In this type, formal arguments point to the actual argument. Hence, changes made in the argument are permanent. The example given below illustrates the use of call by value.

➤ 10.16 Write a program to send a value by reference to the user-defined function.

```

int main()
{
    int x,y,change (int*, int*);
    clrscr();
    printf("\n Enter Values of X & Y :");
    scanf("%d %d",&x,&y);
    change(&x,&y);
    printf("\n In main() X=%d Y=%d",x,y);
    return 0;
}
change(int *a, int *b)
{
    int *k;
    *k=*a;
    *a=*b;
    *b=*k;
    printf("\n In Change() X=%d Y=%d", *a, *b);
}

```

OUTPUT:

```
Enter Values of X & Y : 5 4
In Change() X=4 Y=5
In main()   X=4 Y=5
```

Explanation:

In the above example, we are passing addresses of formal arguments to the function `change()`. The pointers in the `change()` receive these addresses, i.e. pointer points to the actual argument. Here, the `change()` function operates on the actual argument through the pointer. Hence, the changes made in the values are permanent. In this type of call, no `return` statement is required.

10.7 FUNCTION RETURNING MORE VALUES

So far, we know that the function can return only one value per call. We can also force the function to return more values per call. It is possible by the call by reference method. The given below example illustrates this fact.

➤ 10.17 Write a program to return more than one value from the user-defined function.

```
int main()
{
    int x,y,add,sub,change (int*, int*, int*, int*);
    clrscr();
    printf("\n Enter Values of X & Y :");
    scanf("%d %d",&x,&y);
    change(&x,&y,&add,&sub);
    printf("\n Addition : %d ",add);
    printf("\n Subtraction : %d ",sub);
    return 0;
}

change(int *a, int *b,int *c,int *d)
{
    *c=*a+*b;
    *d=*a-*b;
}
```

OUTPUT:

```
Enter Values of X & Y : 5 4
Addition : 9
Subtraction : 1
```

Explanation:

In this program the `return` statement is not used. Still function returns more than one value. Actually, no values are returned. Once the addresses of variables are available, we can directly access them and modify their contents. Hence, in the call by reference method, it is possible to directly modify the contents of the variable. The memory addresses of the variables are unique. There is no scope rule for memory address. If we declare the same variable for actual and formal arguments, their memory addresses will be different from each other.

- 10.18 Write a program to pass arguments to the user-defined function by value and reference.

```
int main()
{
    int k,m,other(int,int*);
    clrscr();
    printf("\n Address of k & m in main() : %u %u",&k,&m);
    other(k,&m);
    return 0;
}
other(int k,int *m)
{
    printf("\n Address of k & m in other() : %u %u",&k,m);
}
```

OUTPUT:

```
Address of k & m in main() : 65524 65522
Address of k & m in other() : 65518 65522
```

Explanation:

In the above example, we are passing value as well as reference; therefore it is a mix-call i.e. calls by value and reference. The first variable is passed by the call by value method and second by reference.

- (1) The variable 'k' of the functions `other()` and `main()` is different from each other. The compiler treats them different. Hence, their memory addresses are different.
- (2) We are passing the address of variable 'm'. It is received by the pointer '*m' of the `other()` function, i.e. pointer 'm' of function `other()` contains the memory location of variable 'm' of `main()`. The pointer 'm' points to variable 'm'. Hence, the address printed of 'm' is the same as in `main()`. This is the difference between these two calls.

10.8 FUNCTION AS AN ARGUMENT

Till now we passed values or address through the functions. It is also possible to pass a function as an argument.

- 10.19 Write a program to pass a user-defined function as an argument to another function.

```
#include<stdio.h>
#include<conio.h>
int doub(int m);
int square(int k);
void main()
{
    int y=2,x;
    clrscr();
    x=doub(square(y));
    printf("x=%d",x);
}
int doub(int m)
{
```



```

    int p;
    p=m*2;
    return(p);
}
int square(int k)
{
    int z;
    z=k*k;
    return(z);
}

```

OUTPUT:

x=8

Explanation:

In the above example, instead of passing a variable or its address, a function is passed as an argument. The innermost function `square()` is executed first and completed. Its return value is collected by the `doub()` function. The `doub()` function uses the `square()` function as an argument. It operates on the return value of `square()` function and makes the returned value double and returns to the `main()` function.

- 10.20 Write a program to use two functions as arguments for another function.

```

# include <math.h>
int main()
{
    int d,x(int,int),y(),z();
    d=x(z(), y());
    printf("\n z() - y()= %d",d);
    return 0;
}
int x(int a, int b)
{
    return(abs(a-b));
}
int y()
{
    int y;
    clrscr();
    printf("\n Enter First Number : ");
    scanf("%d",&y);
    return(y);
}
int z()
{
    int z;
    printf("\n Enter Second Number : ");
    scanf("%d",&z);
    return(z);
}

```

OUTPUT:

```
Enter First Number : 25
Enter Second Number : 50
z() - y()= 25
```

Explanation:

The above program consists of three user-defined functions `x()`, `y()` and `z()`. The functions `z()` and `y()` are passed through the function `x()`, i.e. they are used as arguments. They read values and return to function `x()`. The function `x()` carries subtraction and return to `main()`. The variable '`d`' prints difference between these two values. The `abs()` function is called through the `return()` statement to return only the positive value.

Note: `abs()` function returns absolute value of a given number.

Below given program can be used in place of the `abs()` function. It is a user-defined function, we call it as `uabs()`. 'U' stands for user-defined.

- 10.21 Write a program to return only the absolute value like the `abs()` function.

```
int main()
{
    int uabs(int),x;
    clrscr();
    printf("Enter a Value :");
    scanf("%d",&x);
    x=uabs(x);
    printf("\n X= %d",x);
    return 0;
}

int uabs(int y)
{
    if(y<0)
        return( y * -1);
    else
        return(y);
}
```

OUTPUT:

```
Enter a Value : -5
X = 5
```

Working of `uabs()` function.

The `uabs()` function receives values from the calling function. If the value is less than zero (0), i.e. negative, it is multiplied by `-1` to make it absolute, otherwise it returns as it is.

- 10.22 Write a program to calculate square and cube of an entered number. Use function as an argument.

```
# include <math.h>
int input();
int sqr(int m);
int cube(int m);
void main()
{
    int m;
    clrscr();
    printf("\n\tCube : %d",cube(sqr(input())));
}
int input()
{
    int k;
    printf("Number :");
    scanf("%d",&k);
    return k;
}
int sqr(int m)
{
    printf("\tSquare : %d",m*m);
    return m;
}
int cube(int m)
{
    return m*m*m;
}
```

OUTPUT:

```
Number : 2
Square : 4
Cube : 8
```

Explanation:

In the above program, three user-defined functions are defined. They are `input()`, `sqr()` and `cube()`.

1. The function `input()` prompts user to enter a number.
2. The function `sqr()` calculates the square of the entered number.
3. The function `cube()` calculates the cube of the entered number.

In the `printf()` statement, the `input()` function is used as an argument of `sqr()` function and again the function `sqr()` is used as an argument for the `cube()` function. These functions work as given below.

The function `input()` is executed first. It prompts the user to enter a number. The function `input()` returns the value which is the entered number. The return value of `input()` function is passed to the function `sqr()` and square is calculated. The `sqr()` function returns the number and it is passed to the function `cube()`. The function `cube()` calculates the cube of the original number.

10.9 FUNCTION WITH OPERATORS

The below given program illustrates the use of various operators with a function.

- (1) **The assignment operator (=):** The use of this operator is to assign some value to the variable. To assign return value of function to the variable following syntax is used.

Syntax:

```
x=sqr(2);
```

where 'x' is a variable and `sqr()` is a user-defined function. The value returned by the `sqr()` function is assigned to variable 'x'.

➤ 10.23 Write a program to assign the return value of a function to another variable.

```
void main()
{
    int input(int);
    int x;
    clrscr();
    x=input(x);
    printf("\nx=%d", x);
}
int input(int k)
{
    printf("\nEnter Value of x =");
    scanf("%d",&k);
    return(k);
}
```

OUTPUT:

```
Enter Value of x = 5
x = 5
```

Explanation:

In the above program, the user-defined function `input()` reads integer value through the keyboard. The read value is returned to the `main()` function. The variable 'x' in the `main()` function collects the returned value. The `printf()` statement prints the value of 'x'.

(2) Addition and Subtraction (+ & -):

Syntax:

```
x=(1-fun()+1);
```

where 'x' is an integer and `fun()` is a user-defined function.

➤ 10.24 Write a program to perform the addition and subtraction of numbers with the return value of function.

```
# include <math.h>
void main()
{
    int input(int);
    int sqr(int);
    int x;
    clrscr();
    x=sqr(1-input(x)+1);
    printf("\nSquare = %d", x);
}
```

```

}
int input(int k)
{
    printf("\nEnter Value of x =");
    scanf("%d",&k);
    return(k);
}
int sqr(int m)
{
    return(pow(m,2));
}

```

OUTPUT:

```

Enter Value of x = 5
Square = 9

```

Explanation:

In the above program, two user-defined functions are defined. The `input()` function reads the number through the keyboard. The `sqr()` function calculates the square of the number. The `input()` function is an argument of `sqr()` function.

The value returned by `input()` is subtracted from 2. Though we used ‘-’ operator only once; the equation built is $1-5+1$. The number 5 is the returned value of `input()`. The subtraction takes place two times. Hence, two times one is subtracted and instead of 5, the `sqr()` function receives number -3. Hence, square of -3 is 9.

(3) Multiplication and Division (* & /):

- 10.25 Write a program to perform multiplication and division of numbers with return value of function.

```

# include <math.h>
void main()
{
    int input(int);
    int sqr(int);
    int x;
    clrscr();
    x=sqr(5*input(x)/2);
    printf("\nSquare = %d",x);
}
int input(int k)
{
    printf("\nEnter Value of x =");
    scanf("%d",&k);
    return(k);
}
int sqr(int m)
{
    return(pow(m,2));
}

```

OUTPUT:

```
Enter Value of x = 5
Square = 144
```

Explanation:

The above program is the same as the previous one. Instead of addition and subtraction operations, multiplication and division operations are performed

- (4) **Increment and Decrement Operators (++ & --):** The uses of these operators are to increment or decrement the value of the variable. Here, they perform the same job with the return value of functions.

Syntax:

```
x=fun1(++(y=( fun2 (x))));
```

where 'x' and 'y' are integer and fun1() and fun2() are functions.

Consider the example following example.

- 10.26 Write a program to use (++) operator with the return value of a function.

```
# include <math.h>
void main()
{
    int input(int);
    int sqr(int);
    int x,y=0;
    clrscr();
    x=sqr(++(y=(input(x))));
    printf("\n Square = %d",x);
}
int input(int k)
{
    printf("\n Enter Value of x =");
    scanf("%d",&k);
    return(k);
}
int sqr(int m)
{
    return(pow(m,2));
}
```

OUTPUT:

```
Enter Value of x = 7
Square = 64
```

Explanation:

In the above example, the returned value of input() function is first incremented and passed to the function sqr(). The sqr() function gives the square of incremented value. We cannot

apply these operators (++ and --) directly to the function. We used an extra variable 'y'. The value returned is collected in variable 'y' and then 'y' is incremented.

Note: The decrement (--) operator works in the same way. It decrements the value.

(5) Mod (%) and ? Operators.

➤ 10.27 Write a program to use mod (%) with function.

```
int main()
{
    int j();
    if(j()%2==0)
        printf("\n Number is Even.");
    else
        printf("\n Number is Odd.");
    return 0;
}

int j()
{
    int x;
    clrscr();
    printf("\n Enter a Number :");
    scanf("%d",&x);
    return(x);
}
```

OUTPUT:

```
Enter a Number : 5
Number is Odd.
```

Explanation:

In the above program, the mod operation is directly applied to return the value of function `j()`. The function `j()` reads the number through the keyboard and returns it to `main()`. If the remainder is '0', the number is even, otherwise it is odd.

➤ 10.28 Write a program to use conditional operator (?) with function.

```
# include <math.h>
int main()
{
    int x,y(),z, sqr(int),cube(int);
    clrscr();
    printf("\n Enter a Number :");
    scanf("%d",&x);
    z=(x>y() ? sqr(x) : cube(x));
    printf("\n = %d",z);
    return 0;
}

int sqr(int x)
{
    printf("Square");
}
```

```

        return(pow(x,2));
    }
    int cube(int x)
    {
        printf("Cube");
        return(pow(x,3));
    }
    int y()
    {
        return(10);
    }

```

OUTPUT:

```

Enter a Number : 5
Cube = 125

```

Explanation:

There are three user-defined functions in this example. They are `y()`, `sqr()` and `cube()`. The return value of function `y()` is compared with value entered by the user, i.e. '`x`'. The condition is given in the ternary operator. The ternary operator decides which function to call depending on the condition. If `(x > y())` condition is true, i.e. '`x`' is larger the function `sqr()` is called, otherwise function `cube()` is called. The user-defined functions can be used with (< and >). Here, the function `y()` is used.

➤ 10.29 Write a program to compare two return values of functions.

```

int main()
{
    int a(),b();
    clrscr();
    if(a()==b())
        printf("\n Value of a() & b() are equal.");
    else
        printf("\n Value of a() & b() are unique.");
    return(0);
}

int a()
{
    int x;
    printf("\n Enter a Number a() :");
    scanf("%d",&x);
    return(x);
}

int b()
{
    int x;
    printf("\n Enter a Number b() :");
    scanf("%d",&x);
    return(x);
}

```


OUTPUT:

```
Enter a Number a() : 5
Enter a Number b() : 5
Value of a() & b() are equal.
```

Explanation:

Here, `a()` and `b()` are two user-defined functions. They read values through the keyboard and return to `main()` function. Both functions are called from the `if()` statement. First, `a()` is called. It reads an integer. Similarly, `b()` is called to perform the same task as `a()`. The `if` statement compares both return values and displays proper message.

(6) Arithmetic equations

- 10.30 Write a program to evaluate the equation $s = \text{sqr}(a() + b())$ using function.

```
int main()
{
    int s=0,a(),b(),sqr(int);
    clrscr();
    s=sqr(a()+b());
    printf("\n Square of Sum = %d",s);
    return 0;
}

int a()
{
    int a;
    printf("\n Enter value of a :");
    scanf("%d",&a);
    return(a);
}

int b()
{
    int b;
    printf("\n Enter value of b :");
    scanf("%d",&b);
    return(b);
}

int sqr(int x)
{
    return (x*x);
}
```

OUTPUT:

```
Enter value of a : 5
Enter value of b : 3
Square of Sum = 64
```

Explanation:

In the above program, functions `a()` and `b()` read integer '*a*' and '*b*' through the input device. The returned values of these function are added and this addition is passed to function `sqr()` to obtain the square the sum.

➤ 10.31 Write a program to evaluate the equation $y = x^1 + x^2 + \dots + x^n$ using function.

```
#include<math.h>
int main()
{
    int b(),x,y=0,z=0,n,a;
    clrscr();
    printf("Values of 'x' and 'n': ");
    scanf("%d %d",&x,&n);
    while(z++!=n)
    {
        a=pow(x,b());
        y=y+a;
        printf("%d +",a);
        if(z==n)
        {
            printf("\n Value of y = %d",y);
            return 0 ;
        }
    }
    return 0;
}
int b()
{
    static int m;
    return(++m);
}
```

OUTPUT:

```
Values of 'x' and 'n' : 3 3
3 + 9 + 27 +
Value of y = 39
```

Explanation:

In the above example, the statement $y = y + \text{pow}(x, b())$ within the `while` loop evaluates the series. The `pow()` function calculates the power of 'x' by returned value of function `b()`. The function `b()` is called for 'n' times. During each call, the value of variable 'm' is incremented by one. The variable 'm' is declared as static. The value of variable 'm' does not vanish when the function is not active.

10.10 FUNCTION AND DECISION STATEMENTS

There are two ways to use decision-making statements in C.

(1) Use of `if ... else` statement. For syntax see Table 10.6

Table 10.6 *if else statement with & without function*

Without Function	With Function
<pre>if(condition) Statement1; else Statement1; }</pre>	<pre>if(fun()) statement1; else statement2; where fun() is a function.</pre>
The <code>if ... else</code> statement is used to make the decision in the program at the run time. It decides which statement to execute and which statement to bypass, depending up certain conditions. The above syntax is used commonly in the program.	Here, in this example, instead of writing condition directly, we are passing function. Here, when the <code>if()</code> statement executes control is passed in the function <code>fun()</code>. The return value of function is used for condition.

➤ 10.32 Write a program to call user-defined function through the `if` statement.

```
void main()
{
    int a();
    clrscr();
    if(a()%2==0)
        printf("\n The number is even.");
    else
        printf("\n The Number is odd.");
}

int a()
{
    int a;
    printf("\n Enter value of a :");
    scanf("%d",&a);
    return(a);
}
```

OUTPUT:

```
Enter value of a : 5
The Number is odd.
```

Explanation:

In the above program, `a()` reads number from terminal and sends it to the `main()` function. The mod operation is done on the return value. If the result of mod operation is zero then the message printed is 'The number is even' otherwise 'The number is odd'. Thus, in this example the returned *value* of function is directly used for operation.

- (2) **switch...case** statement: The `switch case` statement also makes decision at run time in the program. It has multiple choices. The `switch()` requires one argument and

its body contains various `case` statements like branch. Depending upon the value of the `switch` argument matched case statement is executed. Syntax of switch with and without function is as per Table 10.7.

Syntax:

Table 10.7 *Function and switch() statements*

<i>Without function</i>	<i>With function</i>
<pre>switch(x) { case 1: ----- break; case 2: ----- break; default: ----- }</pre>	<pre>switch(b()) { case 1: ----- break; case 2: ----- break; default: ----- }</pre>
where 'x' is an argument of any data type.	where b() is a function.

- 10.33 Write a program to call user-defined function through the `switch()` statement.

```
# include <math.h>
# include <ctype.h>
void main()
{
    int a();
    int x=5;
    clrscr();
    switch(a())
    {
        case 's' :
            printf("\n Square of %d is %d",x,pow(x,2));
            break;
        case 'c' :
            printf("\n Cube of %d is %d",x,pow(x,3));
            break;
        case 'd' :
            printf("\n Double of %d is %d",x,x*2);
            break;
        default :
            printf("\n Unexpected Choice printed as it is : %d", x);
    }
}
int a()
{
    char c='';
```

```
printf("Enter Your Choice Square(s),Cube(c),Double(d) : ");
c=getche();
c=tolower(c);
return(c);
}
```

OUTPUT:

```
Enter Your Choice Square(s),Cube(c),Double(d) : D
Double of 5 is 10
```

Explanation:

In the above example, value ‘5’ is assigned to variable ‘x’. Here, the function is called from the `switch()` statement. It prompts the user to enter choice. The choice entered by the user is returned to the `switch()` statement. Depending upon this value, matching case statement is executed. In the shown output, user pressed ‘D’. The `switch()` case executes double operation.

10.11 FUNCTION AND LOOP STATEMENTS

Loop statements are used to repeat program code repetitively for a given number of times or based on certain conditions.

- (1) The `for` loop
- (2) The `while` loop
- (3) The `do..while` loop

1. Working with `for` loop: The syntax of `for` with & without function is as per Table 10.8.

Table 10.8 *Function and for loop*

Without Function	With Function
<code>for(starting value; stop value; step)</code> OR <code>for(initial value; condition, increment/decrement)</code> These are common syntax of the <code>for</code> loop statement.	<code>for(fun(); fun1(); fun2())</code> where, <code>fun()</code> , <code>fun1()</code> and <code>fun2()</code> are user-defined functions. In the above example, at every step of <code>for</code> loop we are using user-defined function, i.e. when such statement executes iterations are completed calling the functions.

I. Working with `for` loop :

➤ 10.34 Write a program to call function through the `for` loop.

```
# include <process.h>
void main()
{
```

```

int plus(int),m=1;
clrscr();
for(;plus(m);m++)
{
    printf("%3d",m);
}
int plus(int k)
{
    if(k==10)
    {
        exit(1);
        return 0;
    }
    else
        return(k);
}

```

OUTPUT:

1 2 3 4 5 6 7 8 9

Explanation:

In the above example, in each iteration of the **for** loop the function **plus()** gets called. The function **plus()** checks the value of the formal argument '*k*'. If variable '*k*' contains 10, the program is terminated, otherwise the value of '*k*' is returned as it is. Thus, the function **plus()** checks the value of the loop variable.

II. Working with the while loop: Its syntax is shown in Table 10.9.

Table 10.9 Function and while loop

Without Function	With Function
<pre>while(condition) { }</pre>	<pre>while(fun()) { }</pre>
This is a common syntax of the while loop statement.	Where, fun() is a user-defined function.
	In the above example, in place of condition we are passing a function. The while checks the returned value of function.

How it works with **while** statement? Below given is an example that illustrates the use of **while** statement.

- 10.35 Write a program to call user-defined function through **while** loop.

```

int main()
{
    int x,y();
}

```

```

    clrscr();
    while(y()!=0)
        printf("Value entered is non-zero");
        return 0;
}
int y()
{
    int x;
    printf("\n Enter a Number :");
    scanf("%d",&x);
    return(x);
}

```

OUTPUT:

```

Enter a Number : 5
Value entered is non-zero
Enter a Number : 0

```

Explanation:

In the above example, `y()` is a user-defined function. The function is called through the `while` loop. When the control is passed to `y()` function, it reads a number through the terminal and returns it to the `while` loop. The `while` loop checks this value. If it is non-zero, the `while` loop executes and a message is displayed 'value entered is non-zero' otherwise if '0' value is returned, the `while` loop terminates.

III. Working with do-while loop. Its syntax is given in Table 10.10.**Table 10.10** Function and do-while loop

Without Function	With Function
<pre>do { }while(condition);</pre> This is a common syntax of the <code>do-while</code> loop statement.	<pre>do { }while(fun());</pre> where, <code>fun()</code> is a user-defined function. In the above example, in place of condition we are passing a function. The <code>while</code> checks the returned value of the function.

- 10.36 Write a program to call a user-defined function through `do-while()` loop.

```

int main()
{
    int x,y();
    clrscr();
    do
        printf("Value enter is non-zero");
    while(y()!=0);
    return 0;
}

```

```

int y()
{
    int x;
    printf("\n Enter a Number :");
    scanf("%d",&x);
    return(x);
}

```

OUTPUT:

```

Value enter is non-zero
Enter a Number : 5
Value enter is non-zero
Enter a Number : 0

```

Explanation:

The above program is similar to the previous one. In the output, the message is displayed, though we have not entered any number. This is the only drawback of the **do-while** statement. It executes once, though the given condition is false.

10.12 FUNCTIONS WITH ARRAYS AND POINTERS

(i) Initialization of Array Using Function

User initializes the array using statement like `int d []={1,2,3,4,5};` instead of this, a function can also be directly called to initialize the array. The given below program illustrates this point.

➤ 10.37 Write a program to initialize an array using functions.

```

int main()
{
    int k,c(),d[]={c(),c(),c(),c(),c()};
    clrscr();
    printf("\n Array d[] elements are :");
    for(k=0;k<5;k++)
        printf("%2d",d[k]);
    return(0);
}
int c()
{
    static int m,n;
    m++;
    printf("\nEnter Number d[%d] : ",m);
    scanf("%d",&n);
    return(n);
}

```

OUTPUT:

```

Enter Number d[1] : 4
Enter Number d[2] : 5

```



```

Enter Number d[3] : 6
Enter Number d[4] : 7
Enter Number d[5] : 8
Array d[] elements are : 4 5 6 7 8

```

Explanation:

A function can be called in the declaration of an array. In the above program, `d[]` is an integer array and `C()` is a user-defined function. The function `C()` when called reads the value through the keyboard. The function `C()` is called from an array, i.e. the value returned by the function is assigned to the array.

(ii) Passing Array Elements to Function

Arrays are a collection of one or more elements of the same data type. Array elements can be passed to the function by value or reference. Below given programs explain both the ways.

- 10.38 Write a program to pass the array element to the function. Use the call by value method.

```

void main()
{
    int k, show(int,int);
    int num[]={12,13,14,15,16,17,18};
    clrscr();
    for(k=0;k<7;k++)
        show(k,num[k]);
}

show(int m,int u)
{
    printf("\nnum[%d] = %d",m+1,u);
}

```

OUTPUT:

```

num[1]=12
num[2]=13
num[3]=14
num[4]=15
num[5]=16
num[6]=17
num[7]=18

```

Explanation:

The `show()` is a user-defined function. The array `num[]` is initialized with seven elements. Using `for` loop, the `show()` function is called for seven times and one element is sent per call. The function `show()` prints the element.

- 10.39 Write a program to pass array element to the function. Use call by reference.

```
void main()
{
    void show(int*);
    int num[]={12,13,14,15,16,17,18};
    clrscr();
    show(num);
}
void show(int *u)
{
    int m=0;
    printf("\n num[7]={");
    while(m!=7)
    {
        printf("%2d,", *(u++));
        m++;
    }
    printf("\b }");
}
```

OUTPUT:

num[7]={12,13,14,15,16,17,18}

Explanation:

In the above program, base address of 0th element is passed to the function `show()`. The pointer `*u` contains base address of array `num[]`. The pointer notation prints the elements.

(iii) Passing Reverse Array to Function

- 10.40 Write a program to display array elements in the reverse order.

```
void main()
{
    int show(int*);
    int num[]={12,13,14,15,16,17,18};
    clrscr();
    show(&num[6]);
}
show(int *u)
{
    int m=6;
    while(m!=-1)
    {
        printf("\nnum[%d] = %d", m, *u);
        u--, m--;
    }
    return(0);
}
```

OUTPUT:

```

num[6]=18
num[5]=17
num[4]=16
num[3]=15
num[2]=14
num[1]=13
num[0]=12

```

Explanation:

Increment or decrement in any pointer points next or previous location of its type, respectively. In the above program, instead of the address of 0th element, the address of the last element is passed. The sixth element is the last element of the array. Decrement in pointer points to the address of the previous element of array.

(iv) Copying Array

We have already studied *call by reference* in which one can change the contents of any local variable of other function, provided its reference or address should be available. Using the call by reference method, copying contents of one array to another is possible.

- 10.41 Write a program to copy array elements using user-defined function.

```

void main()
{
    int cpy(int*, int*),h;
    int a1[]={1,2,3,4,5},a2[5];
    clrscr();
    cpy(&a1[0],&a2[0]);
    printf("Source Target");
    for(h=0;h<5;h++)
        printf("\n%5d\t\t%d",a1[h],a2[h]);
}

int cpy(int *p,int *m)
{
    int j=0;
    while(j!=5)
    {
        *m=*p;
        p++;
        m++;
        j++;
    }
    return(0);
}

```

OUTPUT:

```

Source Target
    1        1

```

2	2
3	3
4	4
5	5

Explanation:

The function `cpy()` collects the base addresses of both the arrays sent by the `main()` function. The pointer `*p` points to array `a1[]` and the pointer `*m` points to array `a2[]`. Assigning value of `*p` to `*m`, elements of array `a1[]` are copied to array `a2[]`. The for loop used in the `main()` function displays contents of both the arrays. Here, the function `cpy()` is called only once. Using the base addresses of the array the function `'cpy()'` performs this task independently.

(v) Reading Private Array

- 10.42 Write a program to read array of other function in `main()`.

```
void main()
{
    int k;
    int array();
    clrscr();
    for(k=0;k<5;k++)
    {
        printf("\t%d", array());
    }
}

int array()
{
    static int k;
    int b[5]={1,2,3,4,5};
    return(b[k++]);
}
```

OUTPUT:

```
1 2 3 4 5
```

Explanation:

In the above program, integer array `b[]` is declared and initialized in the function `array()`. The function `array()` is called five times. During each call, it returns successive elements of array `b[]` to function `main()`. Thus, all elements of array `b[]` are displayed.

(vi) Interchange of Array Elements

- 10.43 Write a program to interchange array elements of two array using functions.

```
void main()
{
    int read();
    void change(int*,int*);
    int x,a[5],b[5];
```

```

clrscr();
printf("Enter 10 Numbers :");
for(x=0;x<10;x++)
{
    if(x<5)
        a[x]=read();
    else
        b[x-5]=read();
}
printf("\n Array A & B");
for(x=0;x<5;x++)
{
    printf("\n%7d%8d", a[x],b[x]);
    change(&a[x],&b[x]);
}
printf("\nNow A & B");
for(x=0;x<5;x++)
{
    printf("\n%7d%8d", a[x],b[x]);
}
}
int read()
{
    int x;
    scanf("%d",&x);
    return(x);
}
void change(int *a, int *b)
{
    int *k;
    *a=*a*b;
    *b=*a-*b;
    *a=*a-*b;
}

```

OUTPUT:

```

Enter 10 Numbers:
1 2 3 4 5 6 7 8 9 0
Array A & B
1 6
2 7
3 8
4 9
5 0
Now A & B
6 1
7 2
8 3
9 4
0 5

```

Explanation:

Interchange of array elements can be done by call by reference. Here, we are passing address of array elements of arrays `a[]` and `b[]` to the function `change()`. The function `change()` interchanges the contents of the elements.

(vii) Global Pointer

- 10.44 Write a program to read array elements declared in different functions using global pointer declaration.

```
int *p,*q;
void main()
{
    int m=0,call(int),k[5]={3,8,5,2,5};
    p=k;
    clrscr();
    call(5);
    while(m!=5)
    {
        printf("%3d",*q);
        m++;
        q++;
    }
}
int call(int j)
{
    static int m=0,u[5]={5,1,6,0,6};
    q=u;
    while(m!=j)
    {
        printf("%3d",*p);
        m++;
        p++;
    }
    printf("\n");
    return(0);
}
```

OUTPUT:

```
3 8 5 2 5
5 1 6 0 6
```

Explanation:

In the above program, '`p`' and '`q`' are declared as global pointers, i.e. they are defined outside the `main()` function. The pointer '`p`' contains the base address of array `k[]` and pointer '`q`' contains base address of `u[]`. Hence, pointers '`p`' and '`q`' are global pointers. They can be accessed through any function.

10.13 PASSING ARRAY TO A FUNCTION

We can pass array as arguments to a function. Passing an array to a function with arguments means address of array is passed. Arrays for all the times are passed to function by address, i.e. array name is passed to the function. Example is illustrated as follows.

- 10.45 Program to pass an array to function and compute the average of five floating point numbers.

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

float avg( float [] );
void main()
{
    float array[5]={2.0,3.5,4.8,5.9,7.8};
    float average;
    clrscr();
    average= avg(array);
    printf("\nAverage of five numbers is %f\n", average );
    getch();
}

float avg( float a[5] )
{
    int i;
    float sum=0;
    for( i = 0; i < 5; ++i )
        sum=sum+a[i];
    return (float)(sum/5);
}
```

OUTPUT:

Average of five numbers is 4.800000

Explanation:

Array is passed to the function `avg()`. One by one elements of an array are passed to the function `avg()` and addition of array element with a variable `sum` is performed. Finally, the average of five elements is returned to the calling function and answer is displayed.

- 10.46 Program to pass an array to function and find the minimum value of the array using pointer.

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

int min( int *);
void main()
{
    int array[5]={10,9,4,5,7},n;
    clrscr();
    printf("Elements of array are:");
    for(int i=0;i<5;i++)
        printf(" %d",array[i]);
    n= min(array);
    printf("\nMinimum of five numbers is %d\n", n );
    getch();
}

int min( int *a )
{

```

```

int i;
int min;
min=*a;
for( i = 1; i < 5; i++ )
{
    if(min>*(a+i))
    {
        min=*(a+i);
    }
}
return min;
}

```

OUTPUT:

Elements of array are: 10 9 4 5 7
 Minimum of five numbers is 4

Explanation:

Here, in this program, min is a function called from the main program. Using call by reference the elements of an array are called in the function min(). Initially, it is assumed minimum value to the first element and compared with successive elements. In case, successive element is smaller than the element under comparison, swapping of elements is done. Eventually the element having minimum value is obtained and the same is displayed on the screen.

10.14 NESTED FUNCTIONS

A nested function refers to function within the function. The programmer can invoke function within a function. The following program gives an idea on how to define function within function.

➤ 10.47 Write a program to display some messages in sub() and nest() functions.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    void sub();
    clrscr();
    printf("\nWe are in main() function");
    sub();
    printf("\nWe are back in main() function");
}
void sub()
{
    void nest();
    printf("\nWe are in sub function");
    nest();
}

```



```
void nest()
{
    printf("\nWe are in nest function");
}
```

OUTPUT:

```
We are in main() function
We are in sub function
We are in nest function
We are back in main() function
```

Explanation:

The function sub() is invoked from the main() and nest() from the sub() and messages are displayed. Prototypes for functions sub() and nest() are initialized at the start of functions.

- 10.48 Write a program to invoke sub() and nest () functions and perform some arithmetic operations.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    void sqr();
    float a=5.5;
    clrscr();
    printf("\nIn main() %g",a);
    sqr();
    printf("\n Back in main() %g",a*a);
}
void  sqr()
{
    void cube();
    float b=2.1;
    printf("\n In sub() %g",b*b);
    cube();
}
void  cube()
{
    float b=1.1;
    printf("\n In nest%g",b*b*b);
}
```

OUTPUT:

```
In main() 5.5
In sub 4.41
In nest 1.331
Back in main() 30.25
```

Explanation:

Invocation of functions is the same as done in the previous program, and only some operations are performed and results are displayed.

10.15 RECURSION

So far, we have seen function calling one another. In programming, there might be a situation where a function needs to invoke itself. The C language supports recursive feature, i.e. a function is called repetitively by itself. The recursion can be used directly or indirectly. The direct recursion function calls to itself till the condition is true. In indirect recursion, a function calls to another function and then called function calls to the calling function.

When a function calls itself until the last call is invoked till that time the first call also remains open. At every time, a function invoked, the function returns the result of previous call. The sequence of return ensues all the way up the line until the first call returns the result to caller function.

➤ 10.49 Write a program to call the `main()` function recursively and perform the sum of one to five numbers.

```

int x,s;
void main(int);
void main(x)
{
    s=s+x;
    printf("\n x = %d s = %d",x,s);
    if(x==5)
        exit(0);
    main(++x);
}

```

OUTPUT:

```

x = 1 s = 1
x = 2 s = 3
x = 3 s = 6
x = 4 s = 10
x = 5 s = 15

```

Explanation:

In the above program, variables *x* and *y* are declared outside the `main()` function. Initially, their values are zeros. Followed by it, the prototype of function `main()` is defined. The variable *x* is passed through the `main()` function. The variable *x* is added to variable *s* till the value of *x* reaches 5. Every time the function `main()` is called repeatedly and *x* is incremented. The result of the program is displayed at the output. The value of *x* in `main()` is 1 because it is a command line argument.

The recursive function `main()` is called as in Table 10.11. The analysis of each step is given for understanding.

Table 10.11 Steps of the recursive function

Function Call	Value of x	Value of s (sum)
main(1)	x = 1	s=1 (0 + 1) = 1
main(2)	x = 2	s=3 (2 + 1 + 0) = 3
main(3)	x = 3	s=6 (3 + 2 + 1 + 0) = 6
main(4)	x = 4	s=10 (4 + 3 + 2 + 1 + 0) = 10
main(5)	x = 5	s=15 (5 + 4 + 3 + 2 + 1 + 0) = 15

- 10.50 Write a program to calculate triangular number of an entered number through the keyboard using recursion.

```
void main()
{
    int n,t,tri_num(int);
    clrscr();
    printf("\n Enter a Number :");
    scanf("%d",&n);
    t=tri_num(n);
    printf("\n Triangular number of %d is %d",n,t);
}

int tri_num(int m)
{
    int f=0;
    if(m==0)
        return(f);
    else
        f=f+m+tri_num(m-1);
    return(f);
}
```

OUTPUT:

```
Enter a Number : 5
Triangular Number of 5 is 15
```

Explanation:

In the above program, a number is entered whose triangular number is to be calculated. The number is passed to the function `tri_num()`. The value of variable *n* is copied to variable *m*. In the `tri_num()` function, the entered number is added to variable *f*. The entered number is decremented and the `tri_num` function is called repetitively (recursively) till the entered number becomes zero.

- 10.51 Write a program using recursion to display sum of digits of a given number.

```
# include <stdio.h>
# include <conio.h>
int sum(int);
void main ()
{
    int num,f;
    clrscr();
    printf ("\n Enter a number : ");
    scanf ("%d", &num);
    f=sum(num);
    printf ("\n Sum of the all digits of given number (%d) is (%d)",num,f);
    getch();
}
```

```

}
int sum(int f)
{
    if(f==0)
        return f;
    else
        return (f%10)+sum(f/10);
}

```

OUTPUT:

Enter a number: 654

Sum of all the digits of the given number (654) is (15)

10.16 TYPES OF RECURSION

Recursion process is a little bit difficult, but if one keeps track of the sequence in which statements are executed then it is easy to understand. Recursion is one of the most dominant tools used in the programming technique. There are various situations when we need to execute a block of statements for a number of times depending on the condition at the time recursion is useful. Recursion is used to solve a problem, which have iterations in the reverse order. Data structures also support recursion, for example tree supports recursion. Various programs are solved with recursion. The major application of recursion is game programming where a series of steps can be solved with recursion.

When a function calls itself it is called recursion. Recursions are of two types:

- (i) Direct recursion
- (ii) Indirect recursion.

When a function calls itself, this type of recursion is direct recursion. In this type, only one function is involved. In indirect recursions, two function calls each other. Figure 10.2 describes direct and indirect recursions.

```

int num()
{
    - - -
    num();
}

```

(a) Direct recursion

```

int num()
{
    - - -
    sum();
}

int sum()
{
    - - -
    num();
}

```

(b) In-direct recursion

Figure 10.2 *Types of recursion*

The recursion is one of the applications of stack. Stacks are also explained in this book. There are several problems without recursion; their solution is lengthy. The programming languages like c, c++ allow us to define the user-defined function. Functions in the programming languages are very useful because by using a function a separate block of statements can be defined. This block can be invoked a number of times anywhere in the program.

Two essential conditions should be satisfied by a recursive function. First every time a function calls itself directly or indirectly, the function should have a condition to stop the recursion. Otherwise, an infinite loop is generated that will halt the system. Some people think that recursion is a very needless luxury in the programming language. Using iteration, one can solve the problems. However, in programming at some situation, there is no substitute for recursion.

There are some kinds of problems associated with recursive functions that are not present in the non-recursive function. A function itself or any other function can invoke the recursive function. To ensure execution, it is very essential for the function to save the return address in order to return at a proper location. Also the function has to save the formal and local variables.

➤ 10.52 Write a C program to calculate factorial of a given number using recursion.

```
int fact(int);
void main()
{
    int num,f;
    clrscr();
    printf("\n Enter a number :");
    scanf("%d",&num);
    f=fact(num);
    printf("\n Factorial of (%d) is (%d)",num,f);
}
int fact(int f)
{
    if(f==1) return f;
    else return f*fact(f-1);
}
```

OUTPUT:

```
Enter a number: 4
Factorial of (4) is (24)
```

Explanation:

In the above program, `fact()` is a recursive function. The entered number is passed to function `fact()`. When function `fact()` is executed, it is repeatedly invoked by itself. Every time a function is invoked, the value of `f` is reduced by one and multiplication is carried out. The recursive function produces the numbers 4, 3, 2 and 1. The multiplication of these numbers is taken out and it return to `main()` function.

10.17 RULES FOR RECURSIVE FUNCTION

1. In recursion, it is essential to call a function itself, otherwise recursion would not take place.
2. Only the user-defined function can be involved in the recursion. Library function cannot be involved in recursion because their source code cannot be viewed.
3. A recursive function can be invoked by itself or by other function. It saves return address with the intention to return at proper location when return to a calling statement is made. The last-in-first-out nature of recursion indicates that stack data structure can be used to implement it.

4. Recursion is turning out to be increasingly important in non-numeric applications and symbolic manipulations.
5. To stop the recursive function, it is necessary to base the recursion on test condition, and proper terminating statement such as `exit()` or `return` must be written using the `if()` statement (see Figure 10.3).

```

int num()
{
    _ _ _
    if
    (condition)
        num();
}

```

Figure 10.3 *Terminating statement in recursion*

6. The user-defined function `main()` can be invoked recursively. To implement such recursion, it is necessary to mention prototype of function `main()`. An example in this regard is as follows.

➤ 10.53 Write a program to call `main()` recursively.

```

# include <process.h>
char str[]="Have a Good Day";
int x=0;
void main(void);
void main()
{
    switch(str[x])
    {
        case 'H' :
            clrscr();
            default:
                printf("%c",str[x]);
                break;
        case '\0':
            exit(0);
            break;
    }
    x++;
    main();
}

```

OUTPUT:

Have a Good Day

Explanation:

In this program, `main()` program is invoked recursively. A prototype of function `main()` is given before the function definition. In recursion, the function invoked should have return type and arguments. Working of recursion is briefed as follows.

1. When a recursive function is executed, the recursion calls are not implemented instantly. All the recursive calls are pushed on to the stack until the terminating condition is not detected. As soon as the terminating condition is detected, the recursive calls stored in the stack are popped and executed. The last call is executed first, then the second, third and so on.
2. During recursion, at each recursive call new memory is allocated to all the local variables of the recursive functions with the same name.
3. At each call, the new memory is allocated to all the local variables; their previous values are pushed onto the stack and with its call. All these values can be available to the corresponding function call when it is popped from the stack.

10.18 DIRECT RECURSION

In this type, only one function is involved which calls itself until the given condition is true. The reader can refer to program numbers 10.45 and 10.46. Following programming example is given on Direct recursion.

➤ 10.54 Write a program to calculate triangular number using recursion.

```
# include <process.h>
void main()
{
    int trinum(int);
    int t,x;
    clrscr();
    printf("\n Enter a Number :");
    scanf("%d",&x);
    t=trinum(x);
    printf("\n Triangular Number of %d is %d",x,t);
}
int trinum(int x)
{
    int f=0;
    if(x==0) return f;
    else f=f+x+trinum(x-1);
    return f;
}
```

OUTPUT:

```
Enter a Number: 4
Triangular Number of 4 is 10
```

Explanation:

In the above program, a function `trinum()` is defined which calls itself. An integer is entered through the keyboard and it is stored in the variable `x`. In function `trinum()`, the function calls itself and decreases the value of `x` passed by one. The return values are added to variable `f`. When the value of `x` becomes zero, the recursive process ends and the total sum is returned to variable `t` in function `main()`. The value of `t` is triangular of number means that the sum of all the numbers between 1 to entered number. The recursive function is called repetitively by itself without completing the execution of previous call. When program ends and the control about to return to caller function the return statement is executed for number of times equal to the function is called recursively. With F7 key, one can keep the track of number of times execution of function.

When a function returns, three actions are done. The return address is placed in the safe location. The data stored in local variables of function is freed. The previously saved address is retrieved. The return value of function is returned and put in the safe location and calling program receives it. Normally, the location is a hardware register, which is placed in CPU for the same purpose.

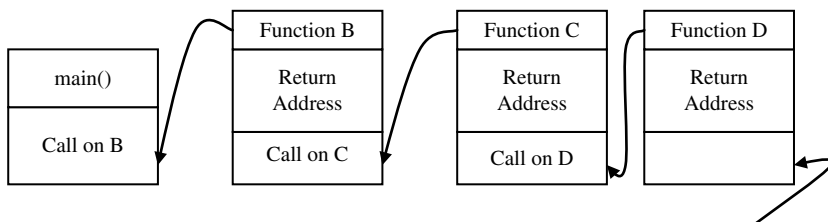


Figure 10.4 Functions calling one another

As shown in Figure 10.4, the main function invoke the function B. The function B invokes the function C and again C invokes D. The figure shows the control is present in function D. In every function, a location is reserved for storage of return address. The return address location of function D contains the address of statement in C immediately after the function invocation statement.

Every time a recursive function invokes itself, new data variables are created and memory is allocated. The data area contains all local variables and return addresses. In the recursion of function, the data area is not connected only with the function but closely associated with the particular function call. In every call new data area is allocated. While returning to the calling function the data area is de-allocated or freed and the former data area turns into current.

The recursion in C and C++ language is more expensive as compared to non-recursive function. It not only takes more space but is also time consuming. In some system programs such as compiler, operating system if a program contains recursive function, it will be executed for a lot of times. In such a case, an alternate non-recursive function may be defined.

10.19 INDIRECT RECURSION

In this type of recursion, two or more functions are involved in the recursion. The indirect recursion does not make any overhead as direct recursion. When control exits from one function and enter into another function, the local variables of former function are destroyed. Hence, memory is not engaged. The following program explains the indirect recursion.

- 10.55 Write a program to demonstrate recursion between two functions.

```
int s=0;
void show(void);
void main()
{
    if(s==5) exit(0);
    show ();
}
void show()
{
    printf(" %d",s);
    s++;
    main();
}
```

OUTPUT:

0 1 2 3 4

Explanation:

In the above program, two user-defined functions are defined as `main()` and `show()`. The `s` is a global variable. The `main()` function invokes the `show()` function and the `show()` function invokes the `main()` function. The value of `s` is increased and displayed. When the value of `s` reaches to 5, the program is terminated.

- 10.56 Write a program to display numbers in different rows.

```
int s;
void show(void);
void main()
{
    if(s==0) clrscr();
    if(s==10) exit(0);
    show();
}
void show()
{
    int j;
    for(j=0;j<=s;j++)
        printf(" %d",j);
    printf("\n");
    s++;
    main();
}
```

OUTPUT:

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8 9
```

Explanation:

This program is the same as the last one. Here, depending on the value of variable *s* the iteration of the **for** loop is performed.

10.20 RECURSION VERSUS ITERATIONS

Recursion and iteration are based on the control structure. Iteration uses repetition structure whereas recursion uses selection structure. Both recursion and iteration repeat the process but with a few differences. Let us discuss the differences.

In iteration repetition structure is explicitly used whereas in recursion the same function is invoked by itself. Both recursion and iteration go through the termination test. Iteration terminates when the loop continuation condition fails. The recursion also terminates when the test for termination is satisfied. Both iteration and recursion can be executed infinitely. When recursion and iteration are defined without termination condition, they turn to infinite loop. Recursion has several overheads. Each time a function is executed, a new copy of function is created. Memory is occupied by the functions. In iteration, only once the variable is created. Thus, iteration is very useful and efficient as compared to recursion. However, there are a few problems that cannot be solved with iteration, and recursion perfectly works.

We have studied both recursion and iteration. They can be applied depending upon the situation. Table 10.12 explains the differences between recursion and iteration. Iterative process is given in Figure 10.5.

Table 10.12 *Recursion versus iterations*

<i>Recursion</i>	<i>Iteration</i>
Recursion is the term given to the mechanism of defining a set or procedure in terms of itself.	The block of statement is executed repeatedly using loops.
A conditional statement is required in the body of the function for stopping the function execution.	The iteration control statements itself contain statements for stopping the iteration. At every execution, the condition is checked.
At some places, the use of recursion generates extra overhead. Hence, it is better to skip when easy solution is available with iteration.	All problems cannot be solved with iteration.
Recursion is expensive in terms of speed and memory.	Iteration does not create any overhead. All the programming languages support iteration.

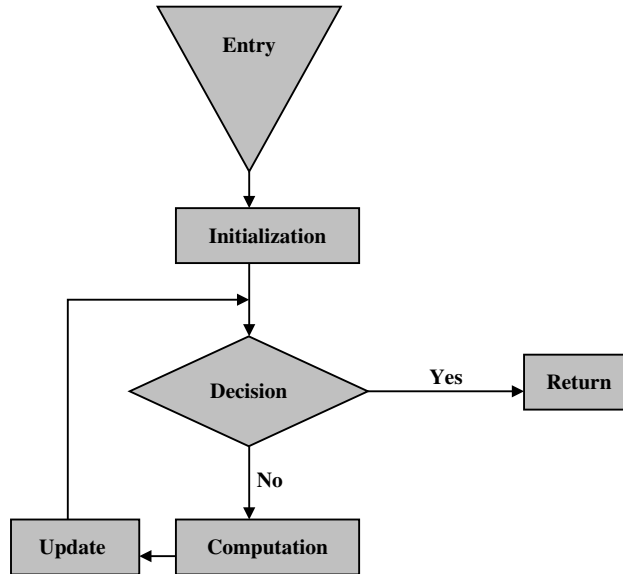


Figure 10.5 Iterative processes

Initialization: The variables involved in iteration process are initialized. These variables are used to decide when to end the loop.

Decision: The decision variable is used to decide if to continue or discontinue the loop. When the condition is satisfied, control goes to return, `else` goes to computation block.

Computation: The required processing or computation is carried out in this block.

Update: The decision argument is changed and shifted to the next iteration.

The common algorithm for any kind of recursive function is as per Figure 10.6.

Preliminary Part: The use of this block is to store the local variables, formal arguments and return address. The end-part will restore these data. Only recently saved arguments, local variables and return address are restored. The variables last saved are restored first.

Body: If the test condition is satisfied, it performs complete processing and control passes to end-block. If not partial processing is performed and a recursive call is made. The body also contains call to itself. There may be one or more calls. Every time a recursive call is made, the preliminary part of the function saves all the data. The body also contains two processing boxes, i.e. partial processing and complete processing. In some programs, the result can be calculated by only complete processing. For this, the recursive call may not be required. For example, we want to calculate factorial of one. The factorial of one is one. For this, it is needless to call function recursively. It can be solved by transferring control to complete processing box.

In other case, if five is given for factorial calculation, the factorial of five cannot be calculated in one step. Hence, the function will be called recursively. Everytime one step is solved, i.e. $5 \times 4 \times 3$ and so on. Hence, it is called partial processing.

Depth of Recursion: The recursion function calls itself infinite times. If we want to calculate factorial of five, then we can easily estimate the number of times the function would be called. In this case, we can determine the depth of the recursive function. In complex programs, it is difficult to determine the number of calls of recursive function.

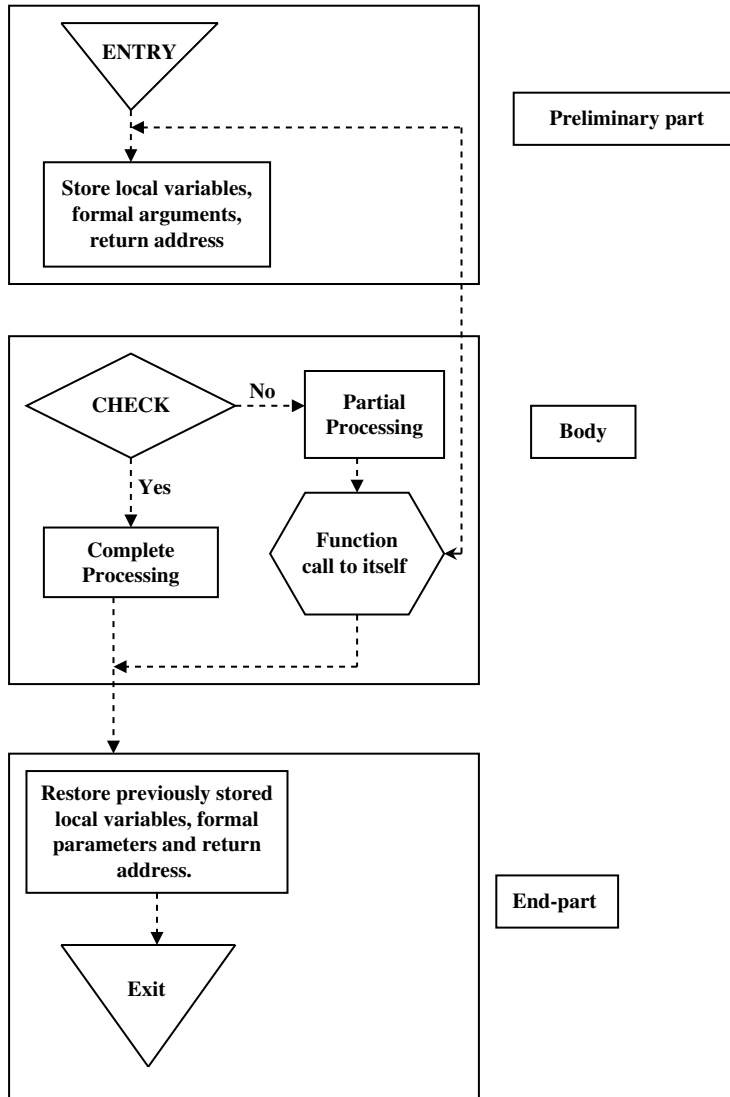


Figure 10.6 Recursive process

10.21 THE TOWERS OF HANOI

The Tower of Hanoi has historical roots in the ceremony of the ancient tower of Brahma. There are n disks of decreasing sizes mounted on one needle as shown in the Figure 10.7 (a). Two more needles are also required to stack the entire disk in the decreasing order. The use of third needle is for impermanent storage. While mounding the disk, following rules should be followed.

1. At a time only one disk may be moved.
2. The disk may be moved from any needle to another needle.
3. No larger disk should be placed on top of the smaller disk.

Our aim is to move the disks from A to C using the needle B as an intermediate by obeying the above three conditions. Only top-most disks can be moved to another needle. The following figures and explanation clear the process of Tower of Hanoi stepwise.

In Figure 10.7 (a), the three needles are displayed in the initial state. The needle X contains three disks and there are no disks on needle Y and Z.

In Figure 10.7 (b), the top-most disk is moved from needle X to Z. The arrow indicates the movement of disk from one needle to another needle.

In Figure 10.7 (c), the disk from the X needle moves to the Y needle.

In Figure 10.7 (d), the disk from Z needle moves to Y needle. Needle Y has two disks.

In Figure 10.7 (e), the disk from the X needle moves to the Z needle. Now there is no disk in the X needle.

In Figure 10.7 (f), the disk from the Y needle moves to the X needle.

In Figure 10.7 (g), the disk from the Y needle moves to the Z needle. The Y needle contains no disk.

In Figure 10.7 (h), the disk from the X needle moves to the Z needle. Thus, the Z needle contains all the three disks of the X needle shown in Figure 10.7 (a). Thus, the problem is solved.

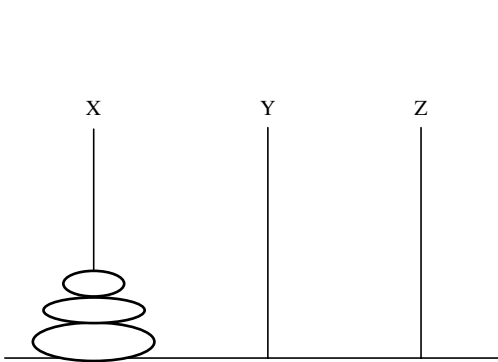


Figure 10.7 (a) *Towers of Hanoi*

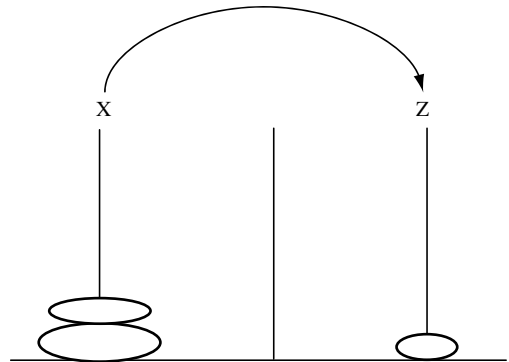


Figure 10.7 (b) *Towers of Hanoi*

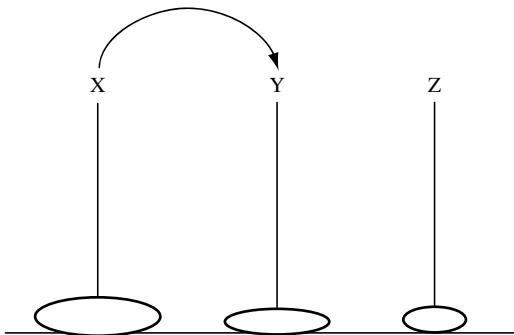


Figure 10.7 (c) *Towers of Hanoi*

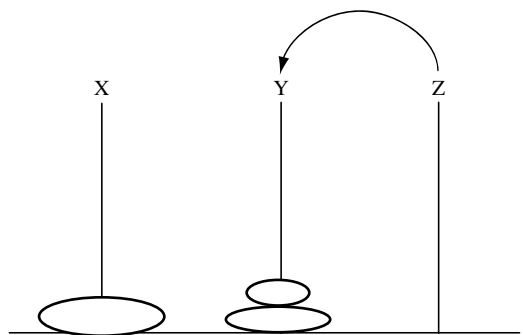
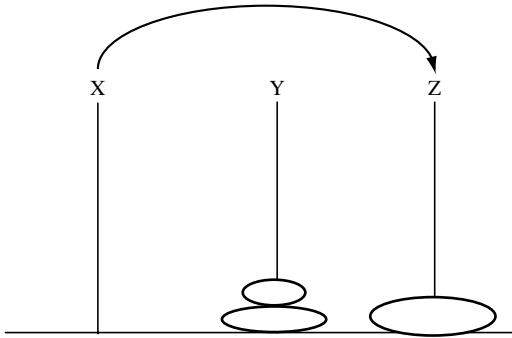
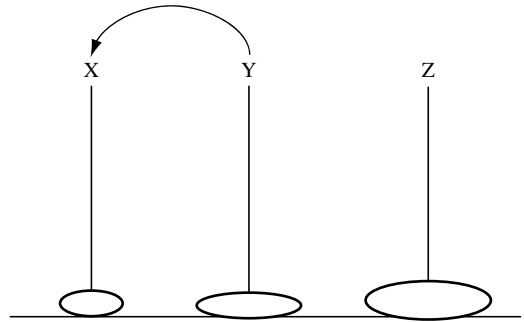
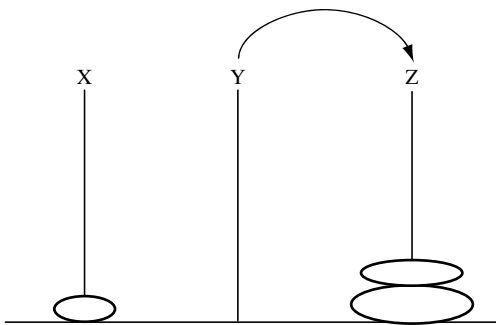
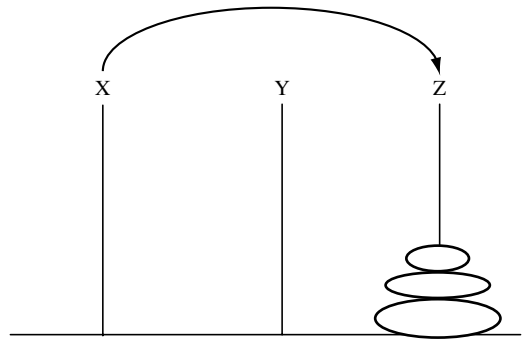


Figure 10.7 (d) *Towers of Hanoi*

Figure 10.7 (e) *Towers of Hanoi*Figure 10.7 (f) *Towers of Hanoi*Figure 10.7 (g) *Towers of Hanoi*Figure 10.7 (h) *Towers of Hanoi*

The above process is summarized as follows:

1. The three needles are displayed in the initial state.
2. The needle X contains three disks and there are no disks on needle Y and Z.
3. The top-most disk is moved from the needle X to Z
4. The disk from the X needle moves to the Y needle.
5. The disk from the Z needle moves to the Y needle.
6. The disk from the X needle moves to the Z needle.
7. The disk from the Y needle moves to the X needle.
8. The disk from the Y needle moves to the Z needle.
9. The disk from the X needle moves to the Z needle.

➤ 10.57 Write a program to illustrate the towers of Hanoi.

```
void hanoi(int, char, char, char);
void main()
{
    int num;
    printf("\n Enter a Number :");
```

```

    scanf("%d",&num);
    clrscr();
    hanoi(num, 'A', 'C', 'B');
}
void hanoi(int num, char f_peg, char t_peg, char a_peg)
{
    if(num==1)
    {
        printf("\nmove disk 1 from Needle %c to %c", f_peg, t_peg);
        return;
    }
    hanoi(num-1, f_peg, a_peg, t_peg);
    printf("\nmove disk %d from Needle %c to Needle %c", num, f_peg,
    t_peg);
    hanoi(num-1, a_peg, t_peg, f_peg);
}

```

OUTPUT:

```

Enter a Number: 3
move disk 1 from Needle A to C
move disk 2 from Needle A to Needle B
move disk 1 from Needle C to B
move disk 3 from Needle A to Needle C
move disk 1 from Needle B to A
move disk 2 from Needle B to Needle C
move disk 1 from Needle A to C

```

Explanation:

In the above program, numbers of disks are entered. The function `Hanoi()` is invoked from `main()`. The A, B and C are needles. If value of n is one, and the disk is transferred from A to C and program ends. If the value of n is greater than one, then the `Hanoi()` function invokes itself recursively. Every time the value of n is decreased by one. The output of the program is shown as above.

10.22 ADVANTAGES AND DISADVANTAGES OF RECURSION

Advantages

- (i) Although, at most of the times, a problem can be solved without recursion, but in some situations in programming, it is a must to use recursion. For example, a program to display a list of all files of the system cannot be solved without recursion.
- (ii) The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
- (iii) Using recursion, the length of the program can be reduced.

Disadvantages

- (i) It requires extra storage space. The recursive calls and automatic variables are stored on the stack. For every recursive calls, separate memory is allocated to automatic variables with the same name.

- (ii) If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory. In such a situation user has to press Ctrl+ break to pause and stop the function.
- (iii) The recursion function is not efficient in execution speed and time.
- (iv) If possible, try to solve a problem with iteration instead of recursion.

10.23 EFFICIENCY OF RECURSION

We have studied both advantages and disadvantages of recursion. We also studied iteration as an alternative to recursion. The major overhead of recursion is the memory it occupies and execution time. A non-recursive function will require minimum memory and less time for execution as compared to the recursive function. The recursive function uses stack to push and pop the local variables. In non-recursive function, the above push and pop operations with stack can be skipped. However, at some situation in programming, the use of recursion is a must. If the part of program is to be invoked frequently, in such a case it is better to develop a non-recursive function.

➤ 10.58 Write a program to display the given string 10 times using recursive call of function.

```
# include <process.h>
char str[]="Have a Good Day";
int x=1;
void main(void);
void main()
{
    printf("\n %.2d] %s",x,str);
    x++;
    switch(x)
    {
        case 1:
            clrscr();
        default :
            main();
        case 11:
            exit(1);
    }
}
```

OUTPUT:

```
01] Have a Good Day
02] Have a Good Day
03] Have a Good Day
04] Have a Good Day
05] Have a Good Day
06] Have a Good Day
07] Have a Good Day
08] Have a Good Day
09] Have a Good Day
10] Have a Good Day
```


Explanation:

The logic of the program is the same as in the previous one. In this program, the entire string is displayed at each call and x is incremented. The `main()` is called through the `switch case` structure. When x reaches 11, the program gets terminated.

10.24 LIBRARY FUNCTIONS

Table 10.13 Library function

S.No.	Function	Description
1	<code>sqrt(j)</code>	Calculates square root of j
2	<code>log(j)</code>	Natural logarithm of j base e
3	<code>ceil(j)</code>	Rounds j to the smallest integer $\leq j$
4	<code>pow(j, k)</code>	j raised to power k
5	<code>rand()</code>	Generates random number

- 10.59 Write a program to demonstrate the use of functions given in Table 10.13.

```
# include <math.h>
void main()
{
    clrscr();
    printf("\n Square root of 9 is %g",sqrt(9));
    printf("\n log of 5 is : %g",log(5));
    printf("\n 10.3 after ceil () : %g",ceil(10.3));
    printf("\n power 3 raised to 4 is : %g",pow(3,4));
}
```

OUTPUT:

```
Square root of 9 is 3
log of 5 is : 1.60944
10.3 after ceil() : 11
power 3 raised to 4 is : 81
```

Explanation:

In the above program, the `sqrt()` function returns the square root of 9. `Log()` gives result 1.60944. The `ceil()` promotes 10.3 to 11 and `pow()` calculates 3^4 . All the above functions are defined in `math.h` header file.

- 10.60 Write a program to demonstrate the use of `rand()` function

```
# include <stdlib.h>
void main ()
{
    int j;
    clrscr();
    for(j=0;j<=15;j++)
    {
```

```

        printf("%10d", 1+(rand()%6));
        if(j%5==0)
            printf("\n");
    }
}

```

OUTPUT:

```

5
5    3    5    5    2
4    2    5    5    5
3    2    2    1    5

```

Explanation:

In the above program, the `rand()` function produces random number which modular divided and added 1. The obtained number is displayed. The last `printf()` inserts a line.

SUMMARY

This chapter is focused on functions. You have studied how to initialize and use functions while writing programs in C. How the functions are interacted with one another is also explained with examples. It is also described how to use function as argument. The reader should know that the function always returns an integer value. Executing programs as illustrated in this chapter can be followed. Besides non-integer specifying data type in function prototype can also return a value. The recursive nature of function is also explained with suitable examples. Direct and indirect recursive functions have been explained with programming examples.

EXERCISES

I True or false:

1. Every C Program starts with function `main()`.
2. Library function can be defined by the user.
3. User-defined functions have serious limitations.
4. The `sum()` is a standard library function.
5. The `abs()` is a standard library function.
6. The `abs()` gives absolute value.
7. Mathematical library functions are defined in `math.h`.
8. Function not returning any value is known as a void.
9. The `return()` statement can return more than one value at a time.
10. The arguments of a function which are invoked are called actual arguments.
11. A user-defined function cannot call another user-defined function.
12. A user-defined function cannot call the `main()` function.
13. The scope of local variables is limited to the block in which they are defined.
14. Actual and formal arguments can have the same variable type.
15. In the call by value, values are passed to formal arguments.
16. In call by address, formal arguments are pointer to actual arguments.

17. A change made in formal arguments can change the value of variable permanently.
18. A function prototype gives information in advance to the compiler.
19. A function can pass arguments to another function.
20. The data type before the function name indicates the type of value the function will return.
21. By default function returns integer value.
22. When a function calls itself the process is called recursion.
23. Like variable, functions also have address in the memory.
24. Functions can be invoked using pointers to function.
25. Function prototype is not necessary to match with actual function definition.

II Match the following correct pairs given in Group A with Group B:

1.

<i>Group A</i>		<i>Group B</i>	
1	abs()	A	Clear the screen
2	sqrt()	B	Quit from the program
3	clrscr()	C	Gives square root of the value
4	exit()	D	Absolute value
		E	Gives square of the number

2.

<i>Group A</i>		<i>Group B</i>	
1	abs()	A	process.h
2	sqrt()	B	conio.h
3	clrscr()	C	math.h
4	exit()	D	stdio.h

3.

<i>Group A</i>		<i>Group B</i>	
1	delay()	A	alloc.h
2	ceil()	B	dos.h
3	circle()	C	math.h
4	malloc()	D	graphics.h

III Select the appropriate option from the multiple choices given below:

1. Arrays are passed as arguments to a function by
 - (a) value
 - (b) reference
 - (c) Both (a) and (b)
 - (d) None of the above
2. Following one keyword is used for function not returning any value
 - (a) void
 - (b) int
 - (c) auto
 - (d) None of the above
3. Recursion is a process in which a function calls
 - (a) itself
 - (b) another function
 - (c) main() function
 - (d) None of the above
4. By default the function returns
 - (a) integer value
 - (b) float value
 - (c) char value
 - (d) None of the above

5. The meaning of keyword void before function name means
- function should not return any value
 - function should return any value
 - no arguments are passed
 - None of the above
6. The function name itself is
- an address
 - value
 - definition
 - None of the above
7. A global pointer can access variable of
- all user-defined functions
 - only main() function
 - only library functions
 - None of the above
8. What will be the values of x and s on execution?
- ```
int x,s;
void main(int);
void main(x)
{
 printf("\n x = %d s = %d",x,s);
}
```
- x=1 s=0
  - x=0 s=0
  - x=1 s=1
  - None of the above
9. The **main()** is a
- user-defined function
  - library function
  - keyword
  - None of the above
10. What will be the value of x after execution
- ```
void main()
{
    float x=2.2,sqr(float),y;
    y=(int)sqr(x);
    printf("\n x=%g",y);
}
float sqr(float m)
{
    return (m*m);
}
```
- x=4
 - x=4.84
 - x=4.50
 - None of the above
11. What is the data type of variable m
- ```
void main()
{
 int sqr(int);
 int x=2;
 int sqr(x);
}
sqr(m)
{
 return (m*m);
}
```
- int
  - float
  - char
  - void

#### IV Attempt the following programming exercises:

- Write a program to display three different Metro names of India by using different functions.
- Write a program with two functions and call one function from other.
- Write a program which takes an int argument and calculates its cube.
- Write a program to display the table of given number. Write different functions for accepting the input, calculating the table and displaying the value.
- Write a program to calculate the sum of digits of a number. Use a function to return the sum.
- Write a program to swap the two variables present in one function to other function.
- Write a program to sort an array (in descending order) in different function and return it to the original function.
- Write a program to display 'Hello!' five times. Create a user-defined function `message()`.

9. Write a program to calculate average marks of five subjects by using pass by value.
10. Write a user-defined function for performing the following tasks.
  - (a) Cube of a number
  - (b) Perimeter of a circle
  - (c) Conversion of binary to decimal
  - (d) Addition of individual digits of a given number
11. Write a program to reverse the given number recursively.
12. Write a program to add 1 to 10 numbers recursively.

## V What will be the output/s of the following program/s?

1.

```
void main()
{
 clrscr();
 printf("\n India is");
 sub();
 getche();
}
sub()
{
 printf(" my Country");
}
```

2.

```
void main()
{
 clrscr();
 printf("\n India is");
 sub();
 secondsub();
 printf("\n I love my country");
 getche();
}
sub()
{
 printf("my Country");
}
secondsub()
{
 printf(" ");
}
```

3.

```
void main()
{
 clrscr();
 sum();
 getche();
}
sum()
{
 int a=7,f=0,d;
 while(a>=1)
 {
```

```
 f=f+a;
 a=a-1;
 }
 printf("\n%d",f);
}
```

4.

```
void main()
{
 clrscr();
 fact();
 getche();
}
fact()
{
 int a=5,f=1,d;
 while(a>=1)
 {
 f=f*a;
 a=a-1;
 }
 printf("\n%d",f);
}
```

5.

```
void main()
{
 int a=4,fac;
 clrscr();
 fac=fact(a);
 printf("\n%d",fac);
 getche();
}
int fact(int x)
{
 int f=1,d;
 while(x>=1)
 {
 f=f*x;
 x=x-1;
 }
 return(f);
}
```

6.

```

void main()
{
 int a=3,b=4,c=5,d;
 clrscr();
 mul();
 d=a+b+c;
 printf("\n%d",d);
 getch();
}
void mul()
{
 int a=2,b=3,c=4,d;
 d=a*b*c;
 printf("\n%d",d);
}

```

7.

```

void main()
{
 int result;
 clrscr();
 result=sq();
 printf("\n Result of function
is= %.2d",result);
}

```

```

 getch();
}
int sq()
{
 int x=2,y=3;
 return(x*x+y*y);
}

```

8.

```

void main()
{
 int num=5,result;
 clrscr();
 result=cb(num);
 printf("\n Result of function
is= %.2d",result);
 getch();
}
int cb(int x)
{
 float y;
 y=x*x*x;
 return(y);
}

```

## VI Find the bug/s in the following program/s?

1.

```

void main()
{
 sum(int f);
 int f=2;
 clrscr();
 void sum(f);
}
sum(int j)
{
 printf("%d",j);
}

```

2.

```

void main()
{
 int a=0, B();
 a=a+ +-B();
 printf("%d",a);
}
int B()
{
 int x;
 printf("Enter a Number");
}

```

```

scanf("%d",&x);
return x;
}

```

3.

```

void main()
{
 int sum(int j, k, l);
 clrscr();
 sum(1,2,3);
}
sum(int j,int k, int l)
{
 printf("%d %d %d",j,k,l);
}

```

4.

```

void main()
{
 void show (void);
 clrscr();
 show();
}
void show()

```

```

{
 puts("Hello");
}
5.
void main()
{
 float ave(int,int,int);
 float av;
 clrscr();
 av=ave (2,3,5);
 printf("%g",av);
}
float ave(int j,int k, int l)
{
 float x= j+k+l/3.0;
}
6.
void A()
{
 clrscr();
 printf("\n in A");
}
void main()
{
 A('x','x');
}

```

```

7.
void main()
{
 int B();
 B(5.5);
}
B(float a)
{
 clrscr();
 printf(" %f",a);
}
8.
void main()
{
 int B();
 B();
}
C()
{
 printf("In C");
}
B()
{
 printf("In B");
 C();
 B();
}

```

## VII Answer the following questions:

- Write the definition of a function. Mention the types of functions available in C.
- How do functions help to reduce the program size?
- Differentiate between library and user-defined functions.
- How does a function works? Explain how arguments are passed and results are returned?
- List any five library functions and illustrate them with suitable examples.
- What are actual and formal arguments?
- What are the uses of the `return()` statements?
- What does it mean if there is no return statement in the function?
- What are the void functions?
- Why is it possible to use the same variable names for actual and formal arguments?
- What is the `main()` function in C? Why is it necessary in each program?
- Explain the different formats of `return()` statements. How many values return statement returns at each call?
- What is a global pointer? Illustrate with a suitable example.
- Why the return statement is not necessary when function is called by reference?
- Distinguish between function prototype and function definition.
- Does the function prototype match with the function definition?
- Can we define a user-defined function with the same library function name?
- What is recursion? Explain its advantages.
- Explain the types of recursions.
- Is it possible to call library function recursively?

## ANSWERS

### I True or false:

| Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. |
|-----|------|-----|------|-----|------|-----|------|-----|------|
| 1.  | T    | 2.  | F    | 3.  | F    | 4.  | F    | 5.  | T    |
| 6.  | T    | 7.  | T    | 8.  | T    | 9.  | F    | 10. | F    |
| 11. | F    | 12. | F    | 13. | T    | 14. | T    | 15. | T    |
| 16. | T    | 17. | F    | 18. | T    | 19. | T    | 20. | T    |
| 21. | T    | 22. | T    | 23. | T    | 24. | T    | 25. | F    |

### II Match the following correct pairs given in Group A with Group B:

1.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|
| 1. | D    | 2. | C    | 3. | A    | 4. | B    |

2.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|
| 1. | C    | 2. | C    | 3. | B    | 4. | A    |

3.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|
| 1. | B    | 2. | C    | 3. | D    | 4. | A    |

### III Select the appropriate option from the multiple choices given below:

| Q.  | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q.  | Ans. |
|-----|------|----|------|----|------|----|------|-----|------|
| 1.  | c    | 2. | a    | 3. | a    | 4. | a    | 5.  | a    |
| 6.  | c    | 7. | a    | 8. | a    | 9. | a    | 10. | a    |
| 11. | a    |    |      |    |      |    |      |     |      |

### V What will be the output/s of the following program/s?

| Q. | Ans.                                     | Q. | Ans.                       |
|----|------------------------------------------|----|----------------------------|
| 1. | India is my Country                      | 5. | 24                         |
| 2. | India is my Country<br>I love my country | 6. | 24 12                      |
| 3. | 28                                       | 7. | Result of function is= 13  |
| 4. | 120                                      | 8. | Result of function is= 125 |



**VI Find the bug/s in the following program/s?**

| Q. | Ans.                                                         |
|----|--------------------------------------------------------------|
| 1. | int sum (int) is needed instead of sum(int f).               |
| 2. | Prototype of function sum( ) is incorrect, but program runs. |
| 3. | Function prototype error.                                    |
| 4. | In Function definition void is not mentioned.                |
| 5. | Return statement is not mentioned.                           |
| 6. | No Bug.                                                      |
| 7. | Function prototype error.                                    |
| 8. | infinite recursion.                                          |

## CHAPTER

# 11

# Storage Classes

### Chapter Outline

- 11.1** Introduction
- 11.2** Automatic Variables
- 11.3** External Variables
- 11.4** static Variables
- 11.5** static external Variables
- 11.6** register Variables

## 11.1 INTRODUCTION

The variables declared in C programs are totally different from other languages. We can use the same variable names in the C program in separate blocks. When we declare a variable, it is available only to a specific part or block of the program. Remaining block or other functions cannot access the variable. Variables declared within function are called internal variables and those declared outside are called external or global variables.

The area or block of the C program from where the variable can be accessed is known as the scope of variables. The area or scope of the variable depends on its storage class, i.e. where and how it is declared. There are four scope variables.

- (i) Function
- (ii) File
- (iii) Block
- (iv) Function prototype

The storage class of a variable tells the compiler:

- (i) the storage area of a variable
- (ii) the initial value of a variable if not initialized
- (iii) the scope of a variable
- (iv) the life of a variable, i.e. how long the variable would be active in a program.

Any variable declared in C can have any one of the above four storage classes:

- (i) Automatic variables
- (ii) External variables
- (iii) Static variables
- (iv) Static external variables
- (v) Register variables

The storage class determines storage duration, scope and linkage. The variable's storage duration is the period during which the variable exists in the memory. Some kind of variables are repeatedly created in memory and some exist throughout the program execution.

Another important aspect is scope, i.e. the area in which the variable can be accessed. Some variables are global and can be accessed anywhere in the program. On the other hand, some variables are of limited scope in the block where they are defined. The above declared four kinds of storage classes can be further divided into two types, depending on the storage duration. They are automatic storage duration and static storage duration. Generally, local variables have automatic storage where the use of keyword `auto` is optional. Static variables belong to static storage class defined as `static` and `external` keywords. From the above discussion, we can state that there are two types of storage classes, i.e. (i) local and (ii) global; and storage class specifiers are `auto`, `register`, `static` and `extern`.

### 11.1.1 | Lifetime of a Variable

Every variable has its lifetime, i.e. its time duration during which its status is active in the program. We can also say that lifetime of a variable is the time gap between its declaration and cleanup. The lifetime depends upon the storage class. For example, `auto` variable gets destroyed immediately when the function execution is over, whereas `static` variable remains in the memory.

### 11.1.2 | Visibility of a Variable

Visibility of a variable is another property. It defines its scope. The scope is of two types, i.e. local and global. Global is recognized throughout the program, whereas the local variable scope is limited to the declaration block.

## 11.2 AUTOMATIC VARIABLES

The `auto` variables are defined inside a function. A variable declared inside the function without storage class name is, by default, an `auto` variable. These functions are declared on the stack. The stack provides temporary storage. The scope of the variable is local to the block in which they are defined. These variables are available only to the current block or program. Once the executions of the function take place and return turns off the function, the contents and existence of the `auto` variables or local variables vanish. Whenever a function is executed, all `auto` variables are allocated memory and de-allocated when the execution of function ends. `auto` variables are safe, i.e. they cannot be accessed directly by other functions.

The keyword '`auto`' is used for declaration of automatic variables. By default, every variable pertains to `auto` storage class.

Some programs are given on `auto` storage class.

- 11.1 Write a program to show the working of the `auto` variable.

```
void main()
{
 auto int v=10;
 clrscr();
 call2();
 printf("\nV=%d", v);
}
void call1()
{
 auto int v=20;
 printf("\nV=%d", v);
}
 call2()
{
 auto int v=30;
 call1();
 printf("\nV=%d", v);
}
```

#### OUTPUT:

```
V=20
V=30
V=10
```

#### *Explanation:*

In the above program, the variable '`v`' is declared and initialized in three different functions. Each function, when called, prints the local values of variables '`v`', which are shown at the output. When a function is called, it declares its own variable '`v`', and when a function

terminates, the local variable 'v' is destroyed. Here, every function has its own variable 'v' with different values. The three variables 'v' are having their own unique memory locations. Even after the removal of `auto` word from the program, there will not be any error thrown by the compiler. The reader can verify this fact.

- 11.2 Write a program to show the working of `auto` variables in different blocks.

```
void main()
{
 int x=10;
 clrscr();
 printf("\n X=%d",x);
 {
 int x=20;
 printf("\n X=%d",x);
 }
 printf("\n X=%d",x);
}
```

**OUTPUT:**

```
X=10
X=20
X=10
```

**Explanation:**

In the above program, declaration of 'x' is made two times with different values. Before the second block, the value of 'x' was 10. The control passes to the second block and again 'x' is declared and assigned a new value 20. When the control exits from the second block, the value of second 'x' disappears and the value of current block, i.e. 10, is printed. In the second variable, declaration `int x=20`, i.e. we declared another variable 'x' different from the first variable 'x'. If we declare only `x=20`, i.e. changing the value of the first 'x', no second variable 'x' is created. Hence, `int` tells the compiler to create a new variable and allocates two bytes for it in the memory.

- 11.3 Write a program to use the same variable in different blocks with different data types.

```
void main()
{
 int x=10;
 clrscr();
 printf("\n X=%d",x);
 {
 float x=2.22;
 printf("\n X=%g",x);
 }
 {
 char *x="Auto Storage Class";
```

```

 printf("\n X=%s", x);
 }
 printf("\n X=%d", x);
}

```

**OUTPUT:**

```

X=10
X=2.22
X=Auto Storage Class
X=10

```

**Explanation:**

The above program is the same as the previous one. Here, in each block, the variables are defined with different data types.

### 11.3 EXTERNAL VARIABLES

The external storage class indicates that the variable has been defined at other place other than the place where it is declared. It can be declared in another source file. The variable declaration generally appears before `main()` and use of keyword `extern` is optional. Initialization cannot be done because its value is defined in another source file. Memory is not allotted in the source program, and it is allotted, where it is defined.

The variables that are available to all the functions, i.e. from entire program, can be accessed. The variables are called external or global variables. External variables are declared outside the function body. In case, in a program both `external` and `auto` variables are declared with the same name, the first priority is given to the `auto` variable. In this case, external variable is hidden. If we declare external variables in the program, there is no need to pass these variables to other function. The compiler does not allocate memory for these variables. It is already allocated for it in another module where it is declared as a global variable.

➤ 11.4 Write a program to demonstrate the use of external variables.

```

int j=4;
void main()
{
 extern int j;
 clrscr();
 j=j*3;
 printf("j=%d", j);
 fun();
 printf("\nj=%d", j);
}
fun ()
{
 j=j*j;
}

```

**OUTPUT:**

```

j=12
j=144

```

**Explanation:**

In the above program, variable *j* is declared and initialized with 4. The second declaration inside the `main()` with keyword `external` indicates that it is declared already. The expression `j = j*3` gives the result 12, i.e. the value of *j* considered is 4. In the function `fun()`, the value of *j* is considered 12, i.e. the last result and hence, the result 144. We can also declare the statement `int j=4` in another file. The file must be included in the main program. Consider the program below:

```
FILE v.c
int j=4;
```

- 11.5 Write a program to demonstrate the use of the external variables.

```
include "v.c"
void main()
{
 extern int j;
 clrscr();
 j=j*3;
 printf("j=%d", j);
 fun();
 printf("\nj=%d", j);
}
fun()
{
 j=j*j;
}
```

**OUTPUT:**

```
j=12
j=144
```

**Explanation:**

This program is the same as the last one. Only the statement `int j=4` is stored in the file `v.c`. The file is included in the main program. The output is the same.

- 11.6 Write a program to show the working of the external variables.

```
int v=10;
void main()
{
 clrscr();
 call1();
 call2();
 printf("\n In main() V=%d", v);
}
```

```

call1()
{
 printf("\n In Call1() V=%d",v);
}
call2()
{
 printf("\n In call2() V=%d",v);
}

```

**OUTPUT:**

```

In Call1() V= 10
In call2() V= 10
In main() V= 10

```

**Explanation:**

In the above program, variable 'v' is declared outside the function body and initialized to the value 10. Every function can access the variable 'v', so no re-declaration or local variable is created. Every function in turn prints the value of 'v'. The same value is printed by all the functions.

- 11.7 Write a program to show the working of `auto` and global variables with the same name.

```

int v=10;
void main()
{
 clrscr();
 call1();
 call2();
 printf("\n In main() V=%d",v);
}
call1()
{
 int v=20;
 printf("\n In Call1() V=%d",v);
}
call2()
{
 printf("\n In call2() V=%d",v);
}

```

**OUTPUT:**

```

In Call1() V= 20
In call2() V= 10
In main() V= 10

```

**Explanation:**

In the above program, an external variable and global variables are declared with the same name. In such a case, when `call1()` function is called, the local variable 'v' of `call1()`



hides the global variable 'v'. As soon as the control returns back from the `call1()` function, the local variable 'v' will be destroyed and the global variables 'v' appear.

- 11.8 Write a program to declare external variables using the `extern` keyword.

```
int m=10;
void main()
{
 extern int m;
 clrscr();
 printf("\n m=%d",m);
}
```

**OUTPUT:**

m = 10

**Explanation:**

In the above program, variable 'm' is declared and initialized with 10 before the `main()` function. The variable 'm' is again re-defined with `extern` keyword. The `extern` keyword is optional.

## 11.4 STATIC VARIABLES

The `static` variable may be of an internal or external type, depending where it is declared. If declared outside the function body, it will be `static` global. In case, declared in the body or block, it will be an `auto` variable. When a variable is declared as `static`, its garbage value is removed and initialized to `NULL` value. The contents stored in these variables remain constant throughout the program execution. A `static` variable is initialized only once; it is never reinitialized. The value of the `static` variable persists at each call and the last change made in the value of the `static` variable remains throughout the program execution. Using this property of the `static` variable, we can count how many times a function was called.

The following programs illustrate the working of `static` variables.

- 11.9 Write a program to show the use of the `static` variable.

```
void main()
{
 clrscr();
 for(;;)
 print();
}
print()
{
 int static m;
 m++;
 printf("\nm=%d",m);
}
```

```

 if(m==3)
 exit(1);
}

```

**OUTPUT:**

```

m=1
m=2
m=3

```

**Explanation:**

In the above program, `print()` function is called. The variable '`m`' of `print()` function is the static variable. It is increased and printed in each call. Its contents persist at every call. In the first call, its value is changed from 0 to 1, in the second call from 1 to 2, and in the third call from 2 to 3.

- 11.10 Write a program to show the difference between variables of `auto` and `static` class, when they are not initialized.

```

void main()
{
 int x;
 static int y;
 clrscr();
 printf("\nx=%d & y=%d", x, y);
}

```

**OUTPUT:**

```

x=1026 & y =0

```

**Explanation:**

In the above program, variables '`x`' and '`y`' are declared as integer variables. The variable '`y`' is a static variable. Both variables are printed. The value of '`x`' printed is some garbage, i.e. 1026 and '`y`' is 0. When we declare a variable as static, it is automatically initialized to zero, otherwise some garbage is assigned to its vacant locations.

## 11.5 STATIC EXTERNAL VARIABLES

The static external variable can be accessed as external variables only in the file in which they are defined. No other file can access the static external variables that are defined in another file.

- 11.11 Program to demonstrate the use of `static` external Variable.

```

#include <stdio.h>
#include <conio.h>
static int i;
int main()
{

```

```

 i=20;
 clrscr();
 printf("\n The value of I is : %d",i);
 getch();
 return 0;
}

```

**OUTPUT:**

The value of i is : 20

**Explanation:**

In the above program, the external variable 'i' is accessed within the scope of this program only. Other files cannot access a static external variable that is defined in another file.

**Note:** The difference between an ordinary external variable and a static external variable is of scope. An ordinary external variable is visible to all functions in the file and can be used by functions in other files. A static external variable is visible only to functions in its own file and below the point of definition.

## 11.6 REGISTER VARIABLES

We can also keep some variables in the CPU registers instead of memory. The keyword `register` tells the compiler that the variable list followed by it is kept on the CPU registers, since register access is faster than the memory access. If the CPU fails to keep the variables in CPU registers, in that case, the variables are assumed as `auto` and stored in memory. CPU registers are limited in number. Hence, we cannot declare more variables with register variables. However, compiler automatically converts register variables to non-register variables, once the limit is reached. User cannot determine the success or failure of register variables. We cannot use register class for all types of variables. The CPU registers 8086 in microcomputer are 16 bit registers. The data types `float` and `double` need more than 16 bits space. If we define variables of these data type with register class, no errors will be shown. The compiler treats them as a variable of `auto` class.

**Syntax:**

```
register int k;
```

- 11.12 Write a program to declare and use variable of `register` class.

```

void main()
{
 register int m=1;
 clrscr();
 for(;m<=5;m++)
 printf("\t%d",m);
}

```

**OUTPUT:**

1      2      3      4      5

**Explanation:**

In the above program, variable 'm' is declared and initialized to 1. The **for** loop displays values from 1 to 5. The **register** class variable is used as a loop variable.

## SUMMARY

After studying this chapter, your understanding in regard to the variables is now perfect. You have studied the variables and their motives. The variables are declared using storage class in different blocks and functions. You have also gained the knowledge of **auto storage class** used to define local variables, **static storage class** used to initialize variables with null, **register storage class** to use CPU registers for storage of data and **extern storage class** to declare global variables. Now, programmer must have gained knowledge on **static** external variables and their access, in the file in which they are declared. Varied easy examples have been illustrated on the above storage classes. The reader is advised to execute the programs provided in this chapter, in order to understand the concepts and applications of the storage class.

## EXERCISES

**I Match the following correct pairs given in Group A with Group B:**

1.

| Serial No. | Group A          | Serial No. | Group B        |
|------------|------------------|------------|----------------|
| 1.         | Default storage  | A          | Auto           |
| 2.         | Static storage   | B          | Global scope   |
| 3.         | External storage | C          | Local scope    |
| 4.         | Auto storage     | D          | CPU register   |
| 5.         | Register         | E          | Value persists |

2.

| Serial No. | Group A   | Serial No. | Group B            |
|------------|-----------|------------|--------------------|
| 1.         | Automatic | A          | External data type |
| 2.         | Static    | B          | auto               |
| 3.         | Register  | C          | register           |
| 4.         | External  | D          | static             |

**II Select the appropriate option from the multiple choices given below:**

- A static variable is one that
  - retains its value throughout the life of the program
  - cannot be initialized
  - is initialized once at the commencement of the execution and cannot be changed at the run time
  - is the same as an automatic variable but is placed at the head of the program
- An external variable is one
  - which is globally accessible by all functions
  - which is declared outside the body of any function
  - which resides in the memory till the end of the program
  - all of the above
- If a storage class is not mentioned in the declaration then default storage class is
  - automatic
  - static
  - external
  - register
- If the CPU fails to keep the variables in CPU registers, in that case the variables are assumed
  - automatic
  - static
  - external
  - None of the above
- What will be the value of variable 'x' on the execution of the following program?
 

```
int x;
void main()
{
 clrscr();
 x++;
 printf("\n %d", x);
}
```

  - x=1
  - x=0
  - garbage value
  - None of the above

**III Attempt the following programs:**

- Write a program to calculate the sum of digits of the entered number. Create user-defined function `sumd()`. Use the static variable and calculate the sum of digits.
- Write a program to call function `main()` recursively. The program should be terminated when the `main()` function is called during 5th time. Take help of the static variable.
- Write a program to calculate the triangular number. Use the static variable with the user-defined function.
- Write a program to create the `for` loop from 1 to 10000. Declare the loop variable of class `register` and `auto`. Observe the time required for completing the loop for both types of variables. Find out in which class the execution time is minimum.

**IV Answer the following questions:**

- List and explain the four scope variables in brief.
- What are the automatic variables?
- Distinguish between `static` and `external` variables.
- Briefly explain `register` variables.
- List the limitations of `register` variables.
- Why register storage class does not support all data types?
- Can we declare a variable in different scopes with different data types? Answer in detail.
- Explain lifetime and visibility of a variable.

**V What will be the output/s of the following program/s?**

1.

```

void main()
{
 int v=5;
 clrscr();
 {
 int v=4;
 printf(" %d ",v);
 }
 printf(" %d ",v);
 {
 int v=9;
 printf(" %d ",v);
 }
 printf(" %d ",v);
}

```

2.

```

void main()
{
 int x=11;
 clrscr();
 printf("\n x=%d",x);
 {
 float x=2.5;
 printf(" x=%g",x);
 }
 {
 char *x="Auto Storage
 Class";
 printf(" x=%s",x);
 }
}

```

3.

```

int v=10;
void main()
{
 clrscr();
 call1();
 call2();
 printf("In main()");
}
call1()
{
 printf(" In call1()");
}
call2()
{
 printf(" In call2()");
}

```

4.

```

int v=10;
void main()
{
 clrscr();
 call1();
 call2();
 printf(" In main()");
}
call1()
{
 printf(" In call1() v=%d",v);
}
call2()
{
 printf(" In call2()
 v=%d",v);
}

```

5.

```

int v=10;
void main()
{
 clrscr();
 for(;;)
 print();
}
print()
{
 int static m;
 m++;
 printf(" %d",m);
 if (m==3) exit(0);
}

```

6.

```

void main()
{
 auto int j=2;
 clrscr();
 {
 {
 printf("%d",j);
 }
 printf(" %d",j);
 }
 printf(" %d",j);
}

```

7.

```

void main()
{
 auto int j=1;
 clrscr();
 {
 auto int j=2;
 {
 auto int j=3;
 printf ("%d",j);
 }
 printf ("%d",j);
 }
 printf ("%d",j);
}

```

8.

```

void main()
{
 clrscr();
 increment();
 increment();
 increment();
}
increment()
{
 static int j=1;
 printf ("\n%d",j);
 j++;
}

```

## VI Find the bug/s in the following program/s:

1.

```

Auto int x=20;
void main()
{
 auto int x=10;
 clrscr();
 printf ("\nx=%d",x);
}

```

```

printf ("%d\n",x);
show();
}

```

```

show()
{

```

```

 printf ("%d",x);
}

```

5.

```

void main()
{
 int count =5;
 printf ("\n%d",count--);
 if (count!=0) main();
}

```

2.

```

void main()
{
 register x=10;
 printf ("\nx=%d",x);
}

```

3.

```

void main()
{
 clrscr();
 value();
}
static value()
{
 printf ("Hi");
}

```

6.

```

float x=5.5;
void main()
{
 float y,f();
 clrscr();
 x*=4.0;
 y=f(x);
 printf ("%g %g",x,y);
}
float f(float a)
{
 a+=1.3;
 x-=4.5; return a+x;
 return 0;
}

```

4.

```

void main()
{
 int x=25;
 clrscr();
}

```

```

7.
register int i;
void main()
{
 clrscr();
 for(i=3;i>0;i--)
 printf ("\ % d",i);
 getch();
}

```

## ANSWERS

### I Match the following correct pairs given in Group A with Group B:

1.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | A    | 2. | E    | 3. | B    | 4. | C    | 5. | D    |

2.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|
| 1. | B    | 2. | D    | 3. | C    | 4. | A    |

### II Select the appropriate option from the multiple choices given below:

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | a    | 2. | d    | 3. | a    | 4. | a    | 5. | a    |

### V What will be the output/s of the following program/s?

| Q. | Ans.                                      |
|----|-------------------------------------------|
| 1. | 4 5 9 5                                   |
| 2. | x=11 x=2.5 x=Auto Storage Class           |
| 3. | In call1() In call2()In main()            |
| 4. | In call1() v=10 In call2() v=10 In main() |
| 5. | 1 2 3                                     |
| 6. | 2 2 2                                     |
| 7. | 321                                       |
| 8. | 1<br>2<br>3                               |



**VI Find the bug/s in the following program/s:**

| Q. | Ans.                                                                                  |
|----|---------------------------------------------------------------------------------------|
| 1. | Automatic variables cannot be defined outside functions.                              |
| 2. | No Bug.                                                                               |
| 3. | No Bug.                                                                               |
| 4. | variable x should be defined.                                                         |
| 5. | count should be static.                                                               |
| 6. | return 0 should be removed and function declaration should be f(float) in third line. |
| 7. | Incompatible storage class.                                                           |

## CHAPTER

# 12 Preprocessor Directives

### Chapter Outline

- 12.1** Introduction
- 12.2** The `#define` Directive
- 12.3** Undefining a Macro
- 12.4** Token Pasting and Stringizing Operators
- 12.5** The `#include` Directive
- 12.6** Conditional Compilation
- 12.7** The `#ifndef` Directive
- 12.8** The `#error` Directive
- 12.9** The `#line` Directive
- 12.10** The `#pragma inline` Directive
- 12.11** The `#pragma saveregs`
- 12.12** The `#pragma` Directive
- 12.13** The Predefined Macros in ANSI and TURBO-C
- 12.14** Standard I/O Predefined Streams in `stdio.h`
- 12.15** The Predefined Macros in `ctype.h`
- 12.16** Assertions

## 12.1 INTRODUCTION

You are now aware about the execution of C programs. For refreshing the knowledge of the reader, the program execution needs certain steps. They are as follows: (i) the C program is written in editor, and then (ii) compilation, (iii) linking and (iv) the executable code generation are to be done. In between these stages, there also involves one more stage i.e. preprocessor. The preprocessor is a program that processes the source program, before it is passed to the compiler.

The program typed in the editor is the source code to the preprocessor. The preprocessor then passes the source code to the compiler. It is not necessary to write the program with the preprocessor facility. But it is good practice to use it preferably at the beginning. Preprocessor can reduce the execution time of a program, because it takes place of a function call.

One of the most important features of the C language is to offer preprocessor directives. The preprocessor directives are always preferably initialized at the beginning of the program before the `main()`. It begins with a symbol `#` (hash). It can be placed anywhere but quite often, it is declared at the beginning before the `main()` function or any particular function. In traditional C language `#` (hash) must begin at the first column.

## 12.2 THE `#define` DIRECTIVE

The syntax of the `#define` directive is as follows:

```
define identifier substitute
```

OR

```
#define identifier (argument 1... argument N) substitute
```

**Example:**

```
define PI 3.14
```

This statement defines `PI` as macro templates and `3.14` as macro substitute. During preprocessing, the preprocessor replaces every occurrence of `PI` (identifier) with `3.14` (substitute value). Here, `PI` is a macro template and `3.14` is its macro expansion. The macro templates are generally declared with capital letters for quick identification. One can also define macros with small letters. The macro templates and its expansions must be separated with at least one blank space. It is not necessary to provide space between `#` and `define`. It is optional to the programmer. To increase readability, the programmer should provide space.

The macro definition should not be terminated with a semi-colon. The words followed by `#` are not keywords. The programmer can use these words for variable names.

A few examples are illustrated below for understanding macros.

➤ 12.1 Use the identifier for `3.14` as `PI` and write a program to find the area of circle using it.

```
define PI 3.14
void main()
{
 float r, area;
 clrscr();
```

```

printf("\n Enter radius of the circle in cm :-");
scanf("%f",&r);
area=PI*r*r;
printf("Area of the Circle = %.2f cm^2",area);
getche();
}

```

**OUTPUT:**

```

Enter radius of the circle in cm = 7
Area of the Circle = 153.86 cm2

```

**Explanation:**

In the above program, PI replaces 3.14 during the program execution. In the program, instead of writing the value of PI as 3.14. We directly define the value of PI as 3.14. The term PI is replaced by 3.14 and used for calculating the area of a circle.

➤ 12.2 Write a program to define and create identifier for C statements and variables.

- (1) Read N as 10
- (2) Replace clrscr() with cls
- (3) Replace getche() with wait()
- (4) Replace printf with display

```

define N 10
define cls clrscr()
define wait() getche()
define display printf
void main()
{
 int k;
 cls;
 for(k=1;k<=N;k++)
 display(" %d",k);
 wait();
}

```

**OUTPUT:**

```

1 2 3 4 5 6 7 8 9 10

```

**Explanation:**

In the above program pre-processor directives are defined as follows:

(i) 10 is replaced by N. (ii) `getche()` is replaced by `wait()`. (iii) `printf` is replaced by `display`. The preprocessor directives are executed first and then program.

- 12.3 Write a program to define macros for logical operators.

```
define and &&
define equal ==
define larger >
void main()
{
 int a,b,c;
 clrscr();
 printf("Enter Three Numbers. :");
 scanf("%d %d %d",&a,&b,&c);
 if(a larger b and a larger c)
 printf("%d is larger than other two numbers.",a);
 else
 if(b larger a and b larger c)
 printf("%d is larger than other two numbers.",b);
 else
 if(c larger a and c larger b)
 printf("%d is the larger than other two numbers.",c);
 else
 if(a equal b and b equal c)
 printf("\n Numbers are same.");
}
```

**OUTPUT:**

```
Enter Three Numbers:7 8 4
8 is larger than other two numbers.
```

**Explanation:**

In the above program, three macros are defined. They are and (&&), equal (=) and larger (>). Instead of using operators in expressions, macros are used and result is obtained.

- 12.4 Write a program to create identifier for displaying double and triple of a number.

```
define DOUBLE(a) a*2
define TRIPLE(a) a*3
void main()
{
 int a=1;
 clrscr();
 printf("\nSINGLE\tDOUBLE\tTRIPLE");
 for(;a<=5;a++)
```

```
printf("\n%d\t%d\t%d", a, DOUBLE(a), TRIPLE(a));
getche();
}
```

**OUTPUT:**

| SINGLE | DOUBLE | TRIPLE |
|--------|--------|--------|
| 1      | 2      | 3      |
| 2      | 4      | 6      |
| 3      | 6      | 9      |
| 4      | 8      | 12     |
| 5      | 10     | 15     |

**Explanation:**

In this program, we are using two identifiers `double()` and `triple()`. These are substitute for `double()` and `triple()` with  $a*2$  and  $a*3$ , respectively. When a value is passed through these macros their corresponding expansions are solved.

## 12.3 UNDEFINING A MACRO

A macro defined with `#define` directives can be undefined with `#undef` directive.

**Syntax:**

```
#undef macro_template substitute
```

It is useful, when we do not want to allow the use of macros in any portion of the program.

➤ 12.5 Write a program to undefine a macro.

```
define wait getche()
void main()
{
 int k;
 # undef wait() getche();
 clrscr();
 for(k=1;k<=5;k++)
 printf("%d\t", k);
 wait;
}
```

**Explanation:**

In the above program, `wait()` is defined in place of `getche()`. In the program `#undef` directive undefines the same macro. Hence, the compiler flags an error message 'undefined symbol 'wait' in function main()'. In case, one more statement '`# define wait getche()`' is written after '`#undef wait getche()`' then the compiler does not show any error and the output displayed on the screen would be '1 2 3 4 5'.

## 12.4 TOKEN PASTING AND STRINGIZING OPERATORS

**Stringizing operation:** In this operation macro argument is converted to a string. The sign # carries the operation. It is placed before the argument.

➤ 12.6 Write a program to carry out stringizing operation.

```
define say(m) printf(#m)
void main()
{
 clrscr();
 say>Hello);
 getch();
}
```

**OUTPUT:**

Hello

**Explanation:**

In the above program, after conversion the statement `say>Hello)` is treated as `printf("Hello")`. It is not essential to enclose the text with quotation marks in the stringizing operator. If # is removed from the macro definition, the user has to enclose the text in double quotes.

➤ 12.7 Write a program to carry stringizing operation and macro arguments.

```
define DOUBLE(x) printf("Double of "#x" = %d\n",x*2)
define TRIPLE(x) printf("Triple of "#x" = %d\n",x*3)
void main()
{
 int m;
 clrscr();
 printf("Enter a number :");
 scanf("%d",&m);
 DOUBLE(m);
 TRIPLE(m);
 getch();
}
```

**OUTPUT:**

```
Enter a number : 5
Double of m = 10
Triple of m = 15
```

**Explanation:**

In the above program the value of 'm' is passed to `double()` and `triple()` macros which is assigned to `x`. `#X` prints the name of the variable passed through the macros.

A macro can have arguments. A program on macro with arguments is illustrated below.

- 12.8 Write a program to find the largest out of two numbers using macro with arguments.

```
define MAX(x,y) if (x>y) c=x; else c=y;
void main()
{
 int x=5,y=8,c;
 clrscr();
 MAX(x,y);
 printf("\n Largest out of two numbers = %d",c);
 getch();
}
```

**OUTPUT:**

Largest out of two numbers = 8

**Explanation:**

In the above program, macro `MAX( )` is defined with two arguments `x` and `y`. When a macro is called, its corresponding expression is executed and the result is displayed. The expression contains the `if` statement that determines the largest number and assigns it to variable `c`.

## 12.5 THE #include DIRECTIVE

The `#include` directive loads specified file in the current program. The macros and functions of loaded file can be called in the current program. The included file also gets compiled with the current program. The syntax is as given below:

- (a) `# include "filename"`
- (b) `# include <filename>`

where `#` is a symbol used with directives.

- (a) The file name is included in the double quotations marks, which indicates that the search for the file is made in the current directory and in the standard directories.

**Example:**

```
include "stdio.h"
```

- (b) When the file name is included within the angle brackets, the search for file is made only in the standard directories.

**Example:**

```
include <stdio.h>
include <udf.h>
```



- 12.9 Write a program to call the function defined in 'udf.c' file.

```
include "udf.c"
void main()
{
 clrscr();
 display();
}
```

**OUTPUT:**

Function Called

Contents of udf.c file.

```
int display(); display () {printf("\n Function
Called"); return 0;}
```

**Explanation:**

In the first program the 'udf.c' is included. It is a user-defined function file. It is compiled before the main program is compiled. The complete programs along with the included one are executed. The output of the program 'Function Called' is displayed.

## 12.6 CONDITIONAL COMPILATION

Quite often, one can use conditional compilation directives in the programs. The most frequently used conditional compilation directives are `#ifdef`, `#else`, `#endif`. These directives allow the programmer to include the portions of the codes based on the conditions. The compiler compiles selected portion of the source codes based on the conditions. The syntax of the `#ifdef` directives is given below.

**Syntax:**

```
#ifdef identifier
{
 statement1;
 statement2;
}
#else
{
 statement3;
 statement4;
}
#endif
```

The `#ifdef` preprocessor tests whether the identifier has defined substitute text or not. If the identifier is defined, then `#if` block is compiled and executed. The compiler ignores `#else` block even if errors are intentionally made. Error messages will not be displayed. If identifier is not defined then the `#else` block is compiled and executed.

- 12.10 Write a program to use conditional compilation statement as to whether the identifier is defined or not.

```
define LINE 1
void main()
{
 clrscr();
 #ifdef LINE
 printf("This is line number one. ");
 #else
 printf("This is line number two.");
 #endif
 getch();
}
```

**OUTPUT:**

This is line number one.

**Explanation:**

In the above program, `#ifdef` checks whether the `LINE` identifier is defined or not. If defined the `#if` block is executed. On execution, the output of the program is 'This is line number one'. In case the identifier is undefined, the `#else` block is executed and output is 'This is line number two'.

- 12.11 Write a program similar to the one given above with conditional compilation directives as to whether the identifier is defined or not.

```
define E =
void main()
{
 int a,b,c,d;
 clrscr();
 #ifdef E
 {
 a E 2;
 b E 3;
 printf("A=%d & B=%d", a, b);
 }
 #else
 {
 c=2;
 d=3;
 printf("C=%d & D=%d", c, d);
 }
}
```

```
#endif
getche();
}
```

**OUTPUT:**

A=2 & B=3

**Explanation:**

The execution of the above program is the same as the previous one. The only difference is the name of the identifier. Here, in this program the identifier is *E*. Compiler searches for the identifier *E*. If it is found, then the execution of `#if` block takes place otherwise the execution of `#else` block takes place. The `#endif` statement indicates the end of `#if` `#endif` block.

## 12.7 THE `#ifndef` DIRECTIVE

The syntax of the `#ifndef` directive is given below.

**Syntax:**

```
#ifndef <identifier>
{
 statement1;
 statement2;
}
#else
{
 statement3;
 statement4;
}
#endif
```

The `#ifndef` works exactly opposite to that of `#ifdef`. The `#ifndef` preprocessor tests whether the identifier has defined substitute text or not. If the identifier is defined then `#else` block is compiled and executed and the compiler ignores the `#if` block even if errors are intentionally made. Error messages will not be displayed. If identifier is not defined then `#if` block is compiled and executed.

- 12.12 Write a program to check conditional compilation directives `#ifndef`. If it is observed, display one message, otherwise another message.

```
define T 8
void main()
{
 clrscr();
 #ifndef T
```

```

printf("\n Macro is not defined.");
#else
printf("\n Macro is defined.");
#endif
getche();
}

```

**OUTPUT:**

Macro is defined.

**Explanation:**

In the above program, `#ifndef` checks for the identifier *T*. If it is defined the `#else` block is executed. On the execution of the block, the output of the program is 'Macro is defined'. In case the identifier is undefined, the `#else` block is executed and output is 'Macro is not defined'.

## 12.8 THE `#error` DIRECTIVE

The `#error` is used to display the user-defined message during the compilation of the program. The syntax is as follows:

```

if !defined (identifier)
error <ERROR MESSAGE>
#endif

```

➤ 12.13 Write a program to display the user-defined error message using `#error` directive.

```

define B 1
void main()
{
 clrscr();
 #if !defined(A)
 #error MACRO A IS NOT DEFINED.
 #else
 printf("Macro found.");
 #endif
}

```

**Explanation:**

In the above program identifier '*B*' is defined. In the absence of an identifier, an error is generated and the `#error` directive displays the error message. The error message is user-defined and displayed in the message box at the bottom of the editor.

Note: The `#defined` directive will work exactly opposite to `#! defined` directive. The syntax is as given below:

```

if defined (identifier)
{
}

```

```
#else
error <ERROR MESSAGE>
#endif
```

## 12.9 THE #line DIRECTIVE

In order to renumbering the source text this directive is used. The syntax of line directive is as follows:

```
#line <constant> [<identifier>]
```

This causes the compiler to renumber the line number of the next source line as given by <constant> and <identifier> gives the current input file. If <identifier> is absent, then the current file name remains unchanged.

**Example:**

```
#line 15 pragma.c
```

## 12.10 THE #pragma inline DIRECTIVE

This reports the compiler that the source code has in-line `asm` statements. It is important to know previously that, the source code contains assembly code.

## 12.11 THE #pragma saveregs

This assure that a huge function will not modify the value of any of the registers when it is entered. Place this directive immediately before the function definition.

## 12.12 THE #pragma DIRECTIVE

The ANSI-C and TURBO-C provide `pragma` directives. These `#pragma` directives are defined with `#` (hash) and these are the preprocessor directives. These directives deal with formatting source listing and placing components in the object file generated by the compiler. It sets /resets certain warning and errors during the compilation of C program. When a program is compiled, the compiler throws errors and warnings. The programmer should see the errors rather than the warnings. After the removal of errors, the programmer can turn attention on warnings and take further steps for sorting warnings. Tables 12.1, 12.2 and 12.3 and 12.4 describe the different `pragma` names with error messages.

**Table 12.1** *ANSI violations and #pragma*

| <i>#pragma Name</i>                     | <i>Warning On</i> | <i>Warning Off</i> |
|-----------------------------------------|-------------------|--------------------|
| Hexadecimal or octal constant too large | +big              | -big               |
| Redefinition not identical              | +dup              | -dup               |
| Both return and return of a value used  | +ret              | -ret               |
| Not part of structure                   | +str              | -str               |
| Undefined structure                     | +stu              | -stu               |

(Continued)

**Table 12.1** *ANSI violations and #pragma (Continued)*

| <i>#pragma Name</i>                  | <i>Warning On</i> | <i>Warning Off</i> |
|--------------------------------------|-------------------|--------------------|
| Suspicious pointer conversion        | +sus              | -sus               |
| Void functions cannot return a value | +voi              | -voi               |
| Zero length structure                | +zst              | -zst               |

**Table 12.2** *Common errors and #pragma*

| <i>#pragma Name</i>                  | <i>Warning On</i> | <i>Warning Off</i> |
|--------------------------------------|-------------------|--------------------|
| Assigned a value but never used      | +aus              | -aus               |
| Possible use before definition       | +def              | -def               |
| Code has no effect                   | +eff              | -eff               |
| Parameter never used                 | +par              | -par               |
| Possibly incorrect assignment        | +pia              | -pia               |
| Unreachable code                     | +rch              | -rch               |
| Function should return a value       | +rvl              | -rvl               |
| Ambiguous operators need parentheses | +amb              | -amb               |

**Table 12.3** *Less common errors and #pragma*

| <i>#pragma Name</i>                  | <i>Warning On</i> | <i>Warning Off</i> |
|--------------------------------------|-------------------|--------------------|
| Superfluous & with function or array | +amp              | -amp               |
| No declaration for function          | +nod              | -nod               |
| Call to function with no prototype   | +pro              | -pro               |
| Structure passed by value            | +stv              | -stv               |
| Declared but never used              | +use              | -use               |

**Table 12.4** *Portability warnings and #pragma*

| <i>#pragma Name</i>                         | <i>Warning On</i> | <i>Warning Off</i> |
|---------------------------------------------|-------------------|--------------------|
| Non-portable pointer assignment             | +apt              | -apt               |
| Constant is long                            | +cln              | -cln               |
| Non-portable pointer comparison             | +cpt              | -cpt               |
| Constant out of range in comparison         | +rng              | -rng               |
| Non-portable pointer conversion             | +rpt              | -rpt               |
| Conversion can lose significant digits      | +sig              | -sig               |
| Mixing pointers to signed and unsigned char | +ucp              | -ucp               |

**Syntax:**

```
pragma warn + xxx
```

```
pragma warn -xxx
```

Where the first statement turns on the warning message and the second statement sets off the warning message.

- 12.14 Write a program to set off certain errors shown by the program using `#pragma` directives.

```
#pragma warn -aus
#pragma warn -def
#pragma warn -rvl
#pragma warn -use
int main()
{
 int x=2,y,z;
 printf("\n y= %d",y);
}
```

**Explanation:**

The above program contains the following warnings:

- (i) Possible use of 'y' before definition in function.
- (ii) 'z' declared but never used in function main.
- (iii) 'x' is assigned a value which is never used in function.
- (iv) Function should return a value in function main.

The display of these warning messages can be made on or off by setting the `pragma` options. In the above program, the four `pragma` options are set to off. Hence, after compilation the above listed lines will not be displayed. If the four `pragma` options are set to on the compiler will display these warning messages.

## 12.13 THE PREDEFINED MACROS IN ANSI AND TURBO-C

- (i) **ANSI C Predefined Macros:** The list of predefined macros according to ANSI standard is given in Table 12.5. There are five predefined macros and are always available to the programmer for use. They cannot be undefined. Every macro name is defined with two underscores as prefix and suffix. These macros are useful for finding system information such as date, time and file name and line number. A program is illustrated for testing these macros.

**Table 12.5** *Predefined macros in ANSI –C*

| <i>Predefined Macros</i> | <i>Function</i>                                |
|--------------------------|------------------------------------------------|
| <code>__DATE__</code>    | Displays system date in string format.         |
| <code>__TIME__</code>    | Displays system time in string format.         |
| <code>__LINE__</code>    | Displays line number as an integer.            |
| <code>__FILE__</code>    | Displays current file name in string format.   |
| <code>__STDC__</code>    | In ANSI C the value returned will be non-zero. |

- 12.15 Write a program to use predefined macros of ANSI C.

```
include <stddef.h>
void main()
{
 clrscr();
}
```

```
printf("\nDATE : %s", __DATE__);
printf("\nTIME : %s", __TIME__);
printf("\nFILE NAME : %s", __FILE__);
printf("\nLINE NO. : %d", __LINE__);
}
```

**OUTPUT:**

```
DATE : Oct 05 2010
TIME : 21:07:39
FILE NAME : PRE_MA~1.C
LINE NO. : 8
```

**Explanation:**

In the above program, five macros are used in the `printf()` statements. On its execution the output of the program is displayed as shown above. The program displays system date, time, program file name and total lines in the program. The STDC indicates whether the version of C compiler follows ANSI C standard. Here, the result is one. Hence, the compiler follows ANSI C standards. The programmer should set ANSI keyword on option by selecting **option menu** of the editor – **compiler-source- ANSI keywords** only on.

- (ii) **TURBO-C Predefined Macros:** The predefined macros in TURBO-C are listed in Table 12.6. The programmer can use the macro `__TURBOC__` to find the version of Turbo. Setting option of code generation in option menu one can test next two macros. For more information, the user is advised to view options in option menu.

- 12.16 Write a program to use a few predefined macros indicated in Table 12.6.

**Table 12.6** *Predefined macros in TURBO-C*

| <i>Predefined Macros</i> | <i>Function</i>                                                            |
|--------------------------|----------------------------------------------------------------------------|
| <code>__TURBOC__</code>  | Displays current TURBO-C version                                           |
| <code>__PASCAL__</code>  | 1 if 'Calling convention...' Pascal option is chosen, otherwise undefined. |
| <code>__CDECL__</code>   | 1 if "Calling convention...C" option is chosen, otherwise undefined.       |
| <code>__MSDOS__</code>   | The integer constant 1.                                                    |

```
include <stddef.h>
void main()
{
 clrscr();
 printf("\nMSDOS : %d", __MSDOS__);
 printf("\nCALLING CONVENTION : %d", __CDECL__);
 getch();
}
```

**OUTPUT:**

```
MSDOS : 1
CALLING CONVENTION : 1
```



- (iii) **Memory Model Macros:** The following six macros are defined on the basis of memory models chosen by the user. There are six memory models and only one is defined which is currently in use, and remaining are undefined. The program is illustrated below to find the memory model. To change the memory model the user is advised to select **option menu->compiler -> model**.

➤ 12.17 Write a program to display the name of memory model that is currently in use.

**Table 12.7** Memory model predefined macros

|                          |                        |                         |
|--------------------------|------------------------|-------------------------|
| <code>__TINY__</code>    | <code>__SMALL__</code> | <code>__MEDIUM__</code> |
| <code>__COMPACT__</code> | <code>__LARGE__</code> | <code>__HUGE__</code>   |

```
void main()
{
 clrscr();
 #ifdef __TINY__
 printf("\nTINY %d", __TINY__);
 #else
 #ifdef __SMALL__
 printf("\nSMALL %d", __SMALL__);
 #else
 #ifdef __MEDIUM__
 printf("\nMEDIUM %d", __MEDIUM__);
 #else
 #ifdef __COMPACT__
 printf("\nCOMPACT %d", __COMPACT__);
 #else
 #ifdef __LARGE__
 printf("\nLARGE %d", __LARGE__);
 #else
 printf("\nHUGE %d", __HUGE__);
 #endif
 #endif
 #endif
 #endif
}

```

**OUTPUT:**

LARGE 1

**Explanation:**

In the above program, the preprocessor directives activate only one macro based on the memory model currently selected on the system. Remaining five macros are undefined. The ladder of `#if`, `#else`, `#endif` checks the definition of all the six macros. When it finds the defined macro it is displayed.

# 12.14 STANDARD I/O PREDEFINED STREAMS IN `stdio.h`

The predefined streams automatically open when the program is started. Table 12.8 describes their macro expansion defined in `stdio.h` header file.

**Table 12.8** *Standard I/O predefined macros in `stdio.h`*

| Macros              | Function                      | Definition in <code>stdio.h</code>             |
|---------------------|-------------------------------|------------------------------------------------|
| <code>stdin</code>  | Standard input device.        | <code>#define stdin (&amp;_streams[0])</code>  |
| <code>stdout</code> | Standard output device.       | <code>#define stdout (&amp;_streams[1])</code> |
| <code>stderr</code> | Standard error output device. | <code>#define stderr (&amp;_streams[2])</code> |
| <code>stdaux</code> | Standard auxiliary device.    | <code>#define stdaux (&amp;_streams[3])</code> |
| <code>stdprn</code> | Standard printer.             | <code>#define stdprn (&amp;_streams[4])</code> |

➤ 12.18 Write a program to enter text and display it using macro expansions.

```

void main()
{
 char ch[12];
 int i;
 clrscr();
 printf("Input a Text : ");
 for(i=0;i<11;i++)
 ch[i]=getc(&_streams[0]);
 printf("Text Inputted : ");
 for (i=0;i<11;i++)
 putc(ch[i],&_streams[1]);
}

```

**OUTPUT:**

```

Input a Text : Programming
Text Inputted : Programming

```

**Explanation:**

In the above program instead of using macros their corresponding macro expansion is used in the program. The first macro expansion `&_streams [0]` reads string through the keyboard and the second `&_streams [1]` displays it.

# 12.15 THE PREDEFINED MARCOS IN `ctype.h`

The header file '`ctype.h`' contains a set of macros that check characters. Table 12.9 describes all the macros. These macros take an argument of integer type and return an integer.

**Table 12.9** *Predefined macros in ctype.h.*

| <i>Sr. No.</i> | <i>Macro</i> | <i>Returns True (!0) Value If</i>   |
|----------------|--------------|-------------------------------------|
| 01             | isalpha(d)   | d is a letter.                      |
| 02             | isupper(d)   | d is a capital letter.              |
| 03             | islower(d)   | d is a small letter.                |
| 04             | isdigit(d)   | d is a digit.                       |
| 05             | isalnum(d)   | d is a letter or digit.             |
| 06             | isxdigit(d)  | d is a hexadecimal digit.           |
| 07             | isspace(d)   | d is a space.                       |
| 08             | ispunct(d)   | d is a punctuation symbol.          |
| 09             | isprint(d)   | d is a printable character.         |
| 10             | isgraph(d)   | d is printable, but not be a space. |
| 11             | isctrl(d)    | d is a control character.           |
| 12             | isascii(d)   | d is an ASCII code.                 |

- 12.19 Write a program to identify whether the entered character is a letter or digit and capital or small using predefined macros.

```
include<ctype.h>
void main()
{
 char d;
 int f;
 clrscr();
 printf("\n Enter any character : ");
 d=getche();
 f=isalpha(d);
 if(f!=0)
 {
 printf("\n%c is a letter in",d);
 f=isupper(d);
 if (f!=0)
 printf(" Capital case");
 else
 printf(" Small Case");
 }
 else
 {
 f=isdigit(d);
 if(f!=0)
 printf("\n %c is a digit",d);
 else
 {
 f=ispunct(d);
 if(f!=0)
```

```

 printf("\n %c is a punctuation symbol",d);
 }
}
getche();
}

```

**OUTPUT:**

```

Enter any character : C
C is a letter in Capital case

```

**Explanation:**

In the above program a character is entered through the keyboard. The macro `isalpha()` checks whether it is digit or letter. It returns true or false value to variable `d`. The `if` condition checks the value of variable `d`. If the entered character is a letter the `if` block is executed and again the macro `isupper()` checks whether the character is capital or small. Thus appropriate messages are displayed.

If the entered character is not a letter `else` block of the first `if` statement is executed. The macro `isdigit()` checks whether the character is a digit or other any symbol. If it is a digit the `if` block is executed, otherwise `else` block is executed. In the `else` block, macro `ispunct()` checks whether the entered character is a punctuation symbol. If so the message is displayed.

## 12.16 ASSERTIONS

The `assert()` macro is defined in the `assert.h` header file. This macro tests the value of an expression. If the expression contains a false value, `assert()` displays an error message and executes function `abort()` to abort the program execution. The following program demonstrates the use of the `assert()` function.

➤ 12.20 Write a program to demonstrate the use of the `assert()` macro.

```

include <assert.h>
void main()
{
 int x=4;
 clrscr();
 assert(x!=4);
}

```

**OUTPUT:**

```

Assertion failed: x!=4, file P1.C, line 9
Abnormal program termination

```

**Explanation:**

In this program the value of `x` is 4. The `assert()` macro checks the value of `x`. If the condition is false, the macro executes `abort()` and program is terminated.

## SUMMARY

In this chapter you have studied one of the most useful features of the C language, i.e. the preprocessor directive. It supports the programmer to write portable programs, which can be executed on different types of systems. After having gone through this chapter and on execution of programs, you have the knowledge of the uses of `#define`, `#undef`, `#include`, `#line`, token pasting and stringizing operator, conditional compilation through illustrated examples. You have also learnt how to display programmer's own error messages using `#error` directive and making various warnings on/off displayed by compiler using `#pragma` directive. You have been exposed to predefined macros in `ctype.h`. You are now aware of the predefined macros and their uses.

## EXERCISES

### I Fill in the blanks:

- The program typed in the editor is the \_\_\_\_\_ to the preprocessor.
  - source code
  - object code
  - ASCII code
  - `#define`
  - `#include`
  - `#ifdef`
- The preprocessor passes the source code to the C \_\_\_\_\_.
  - compiler
  - assembler
  - interpreter
- The preprocessor directives are always initialized at the \_\_\_\_\_.
  - beginning of the program
  - run time
  - compile time
- The preprocessor begins with a symbol \_\_\_\_\_.
  - //
  - #
  - \$
- A macro defined with \_\_\_\_\_ directives can be undefined with `#undef` directive.
  - bsm code
  - asm code
  - c++ code
- In \_\_\_\_\_ operation macro argument is converted to string.
  - string concatenation
  - stringizing operation
  - string coping
- The \_\_\_\_\_ loads specified file in the current program.
  - `#include`
  - `#define`
  - `#ifndef`
- The \_\_\_\_\_ is used to display a user-defined message during the compilation of the program.
  - `#error`
  - `#pragma`
  - `#stderr`
- `inline` DIRECTIVE reports the compiler that the source code has in line \_\_\_\_\_ statements.
  - bsm code
  - asm code
  - c++ code

### II True or false:

- The preprocessor is a program that processes the source code before compilation.
- The program typed in the editor is the source code for the preprocessor.
- The `#define` defines the macro templates.
- The macro definition must be terminated by a semi-colon.
- The `define` is a keyword.

6. The `#undef` undefines the macro.
7. The `#include` loads the a specified file.
8. With `#include "stdio.h"` the compiler searches the file in the entire system.
9. With `#include <stdio.h>` the compiler searches the file in the standard directory.
10. Conditional compilation means a few statements can be skipped from a compiler.
11. The `#ifdef` and `#ifndef` work exactly in the same manner.
12. The `#error` flags are user-defined messages.
13. The `#pragma` sets off/on warning and error messages.
14. The `#ninclude` closes the file loaded by `#include`.

### III Match the functions/words given in Group A with meanings in Column B:

1.

| <i>Sr. No</i> | <i>Predefined Macros</i> | <i>Sr. No</i> | <i>Function</i>                                 |
|---------------|--------------------------|---------------|-------------------------------------------------|
| 1             | <code>__DATE__</code>    | A             | Displays current file name in string format     |
| 2             | <code>__TIME__</code>    | B             | Displays line number as an integer              |
| 3             | <code>__LINE__</code>    | C             | In ANSI 'C' the value returned will be non-zero |
| 4             | <code>__FILE__</code>    | D             | Displays system date in string format           |
| 5             | <code>__STDC__</code>    | E             | Displays system time in string format           |

2.

| <i>Sr. No</i> | <i>Directive</i>      | <i>Sr. No</i> | <i>Function</i>                                       |
|---------------|-----------------------|---------------|-------------------------------------------------------|
| 1             | <code>#define</code>  | A             | Specifies the alternative when <code>#if</code> fails |
| 2             | <code>#else</code>    | B             | Tests a compile time condition                        |
| 3             | <code>#include</code> | C             | Tests whether the a macro is not defined              |
| 4             | <code>#ifdef</code>   | D             | Specifies the end of <code>#if</code>                 |
| 5             | <code>#endif</code>   | E             | Tests for a macro definition                          |
| 6             | <code>#ifndef</code>  | F             | Specifies the file to be included                     |
| 7             | <code>#if</code>      | G             | Undefines a macro                                     |
| 8             | <code>#undef</code>   | H             | Defines a macro substitute value                      |

3.

| <i>Sr. No</i> | <i>Macros</i>       | <i>Sr. No</i> | <i>Function</i>              |
|---------------|---------------------|---------------|------------------------------|
| 1             | <code>stdin</code>  | A             | Standard output device       |
| 2             | <code>stdout</code> | B             | Standard printer             |
| 3             | <code>stderr</code> | C             | Standard input device        |
| 4             | <code>stdaux</code> | D             | Standard error output device |
| 5             | <code>stdprn</code> | E             | Standard auxiliary device    |

4.

| <i>Sr. No</i> | <i>Macros</i>             | <i>Sr. No</i> | <i>Function</i>             |
|---------------|---------------------------|---------------|-----------------------------|
| 1             | <code>COMPILER</code>     | A             | Translates assembly program |
| 2             | <code>PREPROCESSOR</code> | B             | A program can be typed      |

(continued)

| Sr. No | Macros    | Sr. No | Function                         |
|--------|-----------|--------|----------------------------------|
| 3      | LINKER    | C      | Compiles the source code         |
| 4      | Assembler | D      | Supplies source code to compiler |
| 5      | EDITOR    | E      | Relocatable object code          |

#### IV Select the appropriate option from the multiple choices given in the brackets:

- What will be the value of  $y$  after the execution of the following program?
 

```
define plus(x) x;
define minus(x) --x+ plus(x);
void main()
{
 int x=8,y;
 clrscr();
 y=minus(x)
 printf("\n y = %d",y);
}
```

(a)  $y = 14$   
 (b)  $y = 15$   
 (c)  $y = 13$   
 (d) None of the above
- What will be the values of variables  $x$  and  $y$  after execution of the following program?
 

```
define P x++;
define plus(x) P
void main()
{
 int x=2,y;
 clrscr();
 y=plus(x)
 printf("\nx = %d y = %d",x,y);
}
```

(a)  $x=3$   $y=2$   
 (b)  $x=3$   $y=3$   
 (c)  $x=2$   $y=2$   
 (d) None of the above
- In the following example whether macro is treated as
 

```
define S "This Book Teaches C"
void main()
{
 printf("\n %s",S);
}
```

(a) macro as well as array  
 (b) only macro  
 (c) only array  
 (d) None of the above
- What will be the output after the execution of the following program?
 

```
define S "This Book Teaches C"
void main()
{
 clrscr();
 printf("\n %c",*(S+3));
}
```

(a) s  
 (b) h  
 (c) i  
 (d) T
- What will be the value of variable  $z$  after the execution of the following program?
 

```
define ROW 2
define COL 3
int a[ROW][COL]={8,6,4,2,0,-2};
void main()
{
 int x,y,z=0;
 clrscr();
 for(x=0;x<ROW;x++)
 for(y=0;y<COL;y++)
 if(a[x][y]>z)
 z=a[x][y];
 printf("\nz = %d",z);
}
```

(a)  $z=8$   
 (b)  $z=-2$   
 (c)  $z=0$   
 (d)  $z=2$
- What will be the value of variable  $k$  and  $m$  after the execution of the following program?
 

```
define product(k) k*k
void main()
{
 int k=3,m;
 m=product(k++);
 clrscr();
 printf("\t k=%d m=%d",k,m);
}
```

(a)  $k=5$   $m=9$   
 (b)  $k=4$   $m=16$

- (c) k=5 m=25
- (d) k=4 m=9

7. The following program will display

```
define P &x
void main()
{
 int x=2;
 clrscr();
 printf("\t %u",P);
}
```

- (a) address
- (b) value
- (c) error message
- (d) None of the above

8. The following program will display the output

```
void main()
{
 int x=2, *p=5;
 p=&x;
 # define P &p
 clrscr();
 printf("\t %d", **P);
}
```

- (a) 2
- (b) 5
- (c) 65500
- (d) None of the above

9. Consider the statement `# deifne PI 3.14`, it means

- (a) every occurrence of PI (identifier) replaced with 3.14 (substitute value).
- (b) occurrence of PI only in expressions replaced with 3.14
- (c) None of the above

10. A macro defined with `#define` directives can be undefined with

- (a) `# undef directive`
- (b) `#ifndef`
- (c) `#!def`

11. The file name is included in the double quotations marks indicates that

- (a) the search for the file is to be made in the current directory and in the standard directories
- (b) the search for the file is to be made in the current directory
- (c) the search for the file is to be made in the entire system

12. When the file name is included without double quotation marks and when the program is executed, the message that appears on the screen will be

- (a) the search for file is made only in the standard directories
- (b) the search for file is made only in the current directory
- (c) the search for the file is to be made in the entire system
- (d) bad file name format in include directory

13. Generally the standard directories for header file and library files are

- (a) `include` and `lib`
- (b) `turboc2` and `include`
- (c) `tc2` and `Lib`

14. The standard directory called `include` contains

- (a) header files
- (b) library file
- (c) program file

15. The standard directory called `lib` contains

- (a) header files
- (b) library file
- (c) program file

16. The conditional compilation directives allow the programmer to

- (a) compile a part of program
- (b) compile entire program
- (c) compile program excluding header files

17. In the statement `#ifdef`, `#else`, `#endif`, the compiler ignores `#else` block

- (a) when macro is not defined
- (b) when macro is defined
- (c) None of the above

18. What is the output of the following program?

```
#define SQUARE X*X*X
void main()
{
 int X=10;
 printf("%d",SQUARE);
 getch();
}
```

- (a) 100
- (b) 1000
- (c) 10



19. What is the output of the following program?

```
#define PI 3.14
void main()
{
 float x;
 clrscr();
 for(x=PI*2;x<=7.28;x++)
```

```
printf("%g",x);
getche();
}
```

- (a) 7.28
- (b) 6.28
- (c) 6.28 7.28

## V What will be the output/s of the following program/s?

1.

```
define PI 3.14
void main()
{
 float r=2.2,area;
 area=PI*r*r;
 clrscr();
 printf("Area of a Circle =
 %.2f cm2",area);
}
```

2.

```
define N 5
define say printf
void main()
{
 int k;
 clrscr();
 for(k=1;k<=N;k++)
 say(" %d ",k);
}
```

3.

```
define DOUBLE(a) a*2
void main()
{
 int a=1;
 clrscr();
 for(;a<=5;a++)
 printf(" %d",DOUBLE(a));
}
```

4.

```
define say(m) printf(#m)
void main()
{
 clrscr();
 say(C is portable);
}
```

5.

```
define MAX(x,y) if (x>y) c=x;
else c=y;
```

```
void main()
```

```
{
 int x=3,y=5,c;
 clrscr();
 MAX(x,y);
 printf("\n Largest of two
 numbers = %d",c);
}
```

6.

```
define else
void main()
{
 clrscr();
 #ifdef LINE
 printf("This is line number
 one. ");
 #else
 printf("This is line number
 two.");
 #endif
}
```

7.

```
define K 1
void main()
{
 clrscr();
 #ifndef K
 printf("\n Macro is not
 defined.");
 #else
 printf("\n Macro is defined.");
 #endif
}
```

8.

```
include <stddef.h>
void main()
{
 clrscr();
 printf("\nDATE : %s",__ __
 DATE__);
}
```

```

printf("\nTIME : %s",_ _
TIME_ _);
printf("\nFILE NAME : %s",_ _
FILE_ _);
printf("\nLINE NO. : %d",_ _
LINE_ _);
}

```

9.

```

define RANGE (j> 30 && j<51)
void main()
{
 int j=31;
 clrscr();
 if (RANGE)
 printf("Within range");
 else
 printf("Out of Range");
}

```

10.

```

define FUN(k) k+3.14
void main()
{
 {
 int x=2;
 clrscr();
 printf(" %g", x*FUN(FUN
(x)));
 }
}

```

11.

```

define PRINT(a) printf ("%c",a)
void main()
{
 int a=2;
 clrscr();
 for(a=48;a<55;a++)
 PRINT(a);
}

```

12.

```

void main()
{
 clrscr();
 #ifdef _ _TINY_ _
 printf("\nTINY %d",_ _
TINY_ _);
 #else
 #ifdef _ _SMALL_ _
 printf("\nSMALL %d",_ _
SMALL_ _);
 #else
 #ifdef _ _MEDIUM_ _
 printf("\nMEDIUM %d",_ _
MEDIUM_ _);
 #endif
 #endif
 #endif
}

```

13.

```

include<math.h>
define P a*2
define X P/4
void main()
{
 int a=20,z;
 clrscr();
 printf("\n %d ",P);
 printf(" %d ",X);
 getch();
}

```

14.

```

define con(x,y) x##y
void main()
{
 int xy=20;
 clrscr();
 printf("\n %d ",con(x,y));
 getch();
}

```

## VI Find the bug/s in the following programs:

1.

```

define 3.14 PI
void main()
{
 printf("%f",PI);
}

```

2.

```

define DOUBLE(a) a*2
define TRIPLE(a) a*3

```

```

void main()
{
 int a=1;
 clrscr();
 for (;a<=5;a++)
 printf("\n %d \t%d\t
%d",a,double(a),
TRIPLE(a));
}

```

```

3. # define wait getche()
 void main()
 {
 int k;
 # undef wait()
 getche();
 clrscr();
 for(k=1;k<5;k++)
 printf(" %d ",k);wait;
 }

4. # define say(m) printf(m)
 # define show(m) printf(#m)
 void main()
 {
 clrscr();
 say(Hello);
 show(Hello);
 }

5. # define T 8
 void main()
 {
 clrscr();
 #ifndef T
 printf("\n Macro is not
 defined");
 #else
 printf("\n Macro is defined");
 }

6. define T 8
 void main()
 {
 clrscr();
 printf("\n %d",T);
 }

7. void main()
 {
 clrscr();
 printf("\n TIME : %s",
 _ _TIME_ _);
 }

8. void main()
 {
 char d='1';
 int f;

```

```

 f=isalpha(1);
 clrscr();
 if (f!=0)
 printf("%c is a letter",d);
 else
 printf("\n %c is not a
 letter",d);
 }

9. # define B 1
 void main()
 {
 clrscr();
 #if !defined(A)
 #error MACRO A IS NOT
 DEFINED.
 #else
 printf("Macro found.");
 #endif
 }

10. define FUN(k) k+3.14
 void main()
 {
 {
 int x=2;
 clrscr();
 printf("%d",x*FUN(x));
 }
 }

11. # define PRINT (a) printf("%c",a)
 void main()
 {
 int a=2;
 clrscr();
 for (a=48;a<55;a++)
 PRINT(a);
 }

12. #pragma warn +aus
 #pragma warn +def
 #pragma warn +rvl
 #pragma warn +use
 void main()
 {
 int x=2,y,z;
 clrscr();
 printf("\n y= %d",y);
 }

```

## ANSWERS

### I Fill in the blanks:

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | a    | 2. | a    | 3. | a    | 4. | b    | 5. | a    |
| 6. | b    | 7. | a    | 8. | a    | 9. | b    |    |      |

### II True or false:

| Q.  | Ans. | Q.  | Ans. | Q. | Ans. | Q. | Ans. | Q.  | Ans. |
|-----|------|-----|------|----|------|----|------|-----|------|
| 1.  | T    | 2.  | T    | 3. | T    | 4. | F    | 5.  | F    |
| 6.  | T    | 7.  | T    | 8. | F    | 9. | T    | 10. | T    |
| 11. | F    | 12. | T    |    |      |    |      |     |      |

### III Match the following correct pairs given in Group A with Group B:

1.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | D    | 2. | E    | 3. | B    | 4. | A    | 5. | C    |

2.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | H    | 2. | A    | 3. | F    | 4. | E    | 5. | D    |
| 6. | C    | 7. | B    | 8. | G    |    |      |    |      |

3.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | C    | 2. | A    | 3. | D    | 4. | E    | 5. | B    |

4.

| Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. | Q. | Ans. |
|----|------|----|------|----|------|----|------|----|------|
| 1. | C    | 2. | D    | 3. | E    | 4. | A    | 5. | B    |

### IV Select the appropriate option from the multiple choices given below:

| Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. | Q.  | Ans. |
|-----|------|-----|------|-----|------|-----|------|-----|------|
| 1.  | a    | 2.  | a    | 3.  | a    | 4.  | a    | 5.  | a    |
| 6.  | a    | 7.  | a    | 8.  | a    | 9.  | a    | 10. | a    |
| 11. | a    | 12. | d    | 13. | a    | 14. | a    | 15. | b    |
| 16. | a    | 17. | b    | 18. | b    | 19. | b    |     |      |

**V What will be the output/s of the following program/s?**

| Q.  | Ans.                                                                                                                                              |
|-----|---------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.  | Area of Circle = 15.20 cm <sup>2</sup> .                                                                                                          |
| 2.  | 1 2 3 4 5                                                                                                                                         |
| 3.  | 2 4 6 8 10                                                                                                                                        |
| 4.  | C is portable.                                                                                                                                    |
| 5.  | Largest of two numbers = 5.                                                                                                                       |
| 6.  | This is line number two.                                                                                                                          |
| 7.  | Macro is defined.                                                                                                                                 |
| 8.  | DATE : Oct 5 2010<br>TIME : 14:04:54<br>FILE NAME : ANK.C<br>LINE NO. : 8                                                                         |
| 9.  | Within Range.                                                                                                                                     |
| 10. | 10.28                                                                                                                                             |
| 11. | 0123456                                                                                                                                           |
| 12. | SMALL 1.<br>Based on the memory model selection the answer will be different.<br>Here a small memory model is selected. Hence the output small 1. |
| 13. | 40 10                                                                                                                                             |
| 14. | 20                                                                                                                                                |

**VI Find the bug/s in the following programs:**

| Q.  | Ans.                                                               |
|-----|--------------------------------------------------------------------|
| 1.  | The statement defining macro would be # define PI 3.14.            |
| 2.  | The macro DOUBLE is typed as double.                               |
| 3.  | The macro wait is undefined.                                       |
| 4.  | The text 'Hello' should be enclosed in double quotes in say macro. |
| 5.  | #endif is missing.                                                 |
| 6.  | # is missing.                                                      |
| 7.  | No Bug Time = 02:20:45.                                            |
| 8.  | Header file <ctype.h> to be included.                              |
| 9.  | Macro A not defined.                                               |
| 10. | format string should be %g or %f.                                  |
| 11. | The gap between PRINT (a) and (a) should be removed.               |
| 12. | There is no bug. Warning errors are set.                           |

## CHAPTER

# 13 Structure and Union

### Chapter Outline

- 13.1** Introduction
- 13.2** Features of Structures
- 13.3** Declaration and Initialization of Structures
- 13.4** Structure within Structure
- 13.5** Array of Structures
- 13.6** Pointer to Structure
- 13.7** Structure and Functions
- 13.8** typedef
- 13.9** Bit Fields
- 13.10** Enumerated Data Type
- 13.11** Union
- 13.12** Calling BIOS and DOS Services
- 13.13** Union of Structures

## 13.1 INTRODUCTION

You know that a variable stores a single value of a data type. Arrays can store many values of a similar data type. Data in the array is of the same composition in nature as far as the type is concerned. In real life, we need to have different data types; for example, to maintain employees information we should have information such as name, age, qualification, salary and so on. Here, to maintain the information of employees dissimilar data types are required. Name and qualification of the employee are `char` data type, age is an `int` and salary is `float`. All these data types cannot be expressed in a single array. One may think to declare different arrays for each data type. But there will be huge increase in source codes of the program. Hence, arrays cannot be useful here. For tackling such a mixed data type problems, a special feature is provided by C. It is known as a structure.

A *structure* is a collection of one or more variables of different data types, grouped together under a single name. It is a derived data type to be arranged in a group of related data items of different data types. It is a user-defined data type because the user can decide the data types to be included in the body of a structure. By using structures, we can make a group of variables, arrays, pointers.

## 13.2 FEATURES OF STRUCTURES

In order to copy elements of an array to another array, elements of the same data type are copied one by one. It is not possible to copy all the elements at a time. However, in a structure it is possible to copy the contents of all structure elements of different data types to another structure variable of its type using assignment (=) operator. It is possible because the structure elements are stored in successive memory locations.

Comparison between array and structure is shown in Table 13.1.

**Table 13.1** Comparison between an array and a structure

| S. No. | Points of Comparison                     | Array                    | Structure                        |
|--------|------------------------------------------|--------------------------|----------------------------------|
| 01     | Collection of data                       | Same data type           | Different data types             |
| 02     | Keyword                                  | It is not a keyword      | <code>struct</code> is a keyword |
| 03     | Declaration and definition of data types | Only declaration         | Both declaration and definition  |
| 04     | Bit fields                               | Does not have bit fields | May contain bit fields           |

Nesting of structures is possible, i.e. one can create structure within the structure. Using this feature, one can handle complex data types. It is also possible to pass structure elements to a function. This is similar to passing an ordinary variable to a function. One can pass individual structure elements or entire structure by value or address.

It is also possible to create structure pointers. In the pointer, we have studied pointing a pointer to an `int`, pointing to a `float` and pointing to a `char`. In a similar way, we can create a pointer pointing to structure elements. For this it requires `->` operator.

### 13.3 DECLARATION AND INITIALIZATION OF STRUCTURES

Structures can be declared as follows:

```
struct struct_type
{
 type variable1;
 type variable2;

};
```

Structure declaration always starts with **struct** keyword. Here, **struct\_type** is known as **tag**. The **struct** declaration is enclosed within a pair of curly braces. Closing brace is terminated with a semi-colon. Using **struct** and **tag**, a user can declare structure variables like **variable1**, **variable2** and so on. These are the members of the structure. After defining structure template, we can create variables as given below:

```
struct struct_type v1, v2, v3;
```

Here *v1*, *v2* and *v3* are variables or objects of structure **struct\_type**. This is similar to declaring variables of any data type.

The declaration defines the structure but this process does not allocate memory. The memory allocation takes place only when variables are declared.

```
struct book1
{
 char book[30];
 int pages;
 float price;
};
struct book1 bk1;
```

In the above example, a structure of type *book1* is created. It consists of three members: **book** [30] of **char** data type, **pages** of **int** type and **price** of **float** data type. Figure 13.1 explains various members of a structure.

```
struct book1 bk1;
```

The above line creates variable **bk1** of type **book1**, and it reserves total 36 bytes (30 bytes for **book[30]**, 2 bytes for **int** and 4 bytes for **float**). Through **bk1** all the three members of structure can be accessed. Program 13.1 can be referred to for understanding the memory size requirement for structure elements. In order to initialize structure elements with certain values following statement is used.

```
struct book1 bk1 = {"shrinivas", 500, 385.00};
```

All the members of structure are related to variable **bk1**.

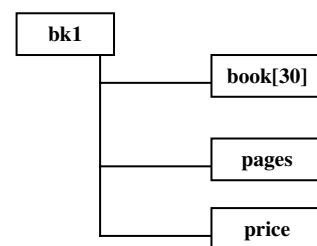


**Figure 13.1** Block diagram of a structure



structure\_variable.member or bk1.book

The period (.) sign is used to access the structure members.

We can directly assign values to members as given below:

```
bk1.book = "shrinivas";
bk1.pages=500;
bk1.price=385.00;
```

- 13.1 Write a program to display the size of structure elements. Use `sizeof()` of operator.

```
void main()
{
 struct book1
 {
 char book[30];
 int pages;
 float price;
 };
 struct book1 bk1;
 clrscr();
 printf("\n Size of Structure Elements");
 printf("\n Book : %d", sizeof(bk1.book));
 printf("\n Pages : %d", sizeof(bk1.pages));
 printf("\n Price : %d", sizeof(bk1.price));
 printf("\n Total Bytes : %d", sizeof(bk1));
}
```

**OUTPUT:**

```
Size of Structure Elements
Book : 30
Pages : 2
Price : 4
Total Bytes : 36
```

**Explanation:**

In the above program, structure `book1` is defined with three member variables `char book[30]`, `int pages` and `float price`, respectively. The `bk1` is an object of the structure `book1`. Using the `sizeof()` operator, their sizes are displayed. The memory sizes in bytes displayed are 30, 2 and 4, respectively. The total size of one record is 36, i.e. size of all member variables of the above structure.

A few examples are illustrated below for understanding the working of a structure.

- 13.2 Write a program to define a structure and initialize its member variables.

```
void main()
{
 struct book1
 {
```

```

 char book[30];
 int pages;
 float price;
 };
 struct book1 bk1={"Programming in C ",600,185};
 clrscr();
 printf("\n Book Name : %s",bk1.book);
 printf("\n No. of Pages : %d",bk1.pages);
 printf("\n Book Price : %.2f",bk1.price);
 getch();
}

```

**OUTPUT:**

```

Book Name : Programming in C
No. of Pages : 600
Book Price : 185.00

```

**Explanation:**

In the above program, the structure `book1` is defined with its member variables `char book[30]`, `int pages` and `float price`. The `bk1` is an object of the structure `book1`. The statement `struct book1 bk1={"Programming in C " 300,285}` defines object `bk1` and initializes the variables with the values enclosed in the curly braces, respectively. Using the `printf()` statement, contents of the individual fields are displayed.

It is possible to copy the structure variable to another structure variable one by one or whole at once. An example is illustrated below on this concept.

➤ 13.3 Write a program to copy structure elements from one object to another object.

```

include <string.h>
void main()
{
 struct disk
 {
 char co[15];
 float type;
 int price;
 };
 struct disk d1={"SONY",1.44,20};
 struct disk d2,d3;
 strcpy(d2.co,d1.co);
 d2.type=d1.type;
 d2.price=d1.price;
 d3=d2=d1;
 clrscr();
 printf("\n %s %g %d",d1.co,d1.type,d1.price);
 printf("\n %s %g %d",d2.co,d2.type,d2.price);
}

```

```
printf("\n %s %g %d", d3.co, d3.type, d3.price);
getche();
}
```

**OUTPUT:**

```
SONY 1.44 20
SONY 1.44 20
SONY 1.44 20
```

**Explanation:**

In the above program, `d1`, `d2` and `d3` are objects defined based on structure `disk`. The object `d1` is initialized. The contents of `d1` are copied to `d2` and `d3` objects. In the first method, individual elements of `d1` object are copied using the assignment statement. The `strcpy()` function is used because the first element of the structure is a string. In the second method, all the contents of `d1` are copied to `d2` and `d3`. Here, the statement `d3=d2=d1` performs this task. Thus, in a structure, elements are possible to be copied (or elements can be copied to ...) to another object of the same type at one stroke.

- 13.4 Write a program to read the values using `scanf()` and assign them to structure variables.

```
void main()
{
 struct book1
 {
 char book[30];
 int pages;
 float price;
 };
 struct book1 bk1;
 clrscr();
 printf("Enter Book name, pages, price :");
 scanf("%s", bk1.book);
 scanf("%d", &bk1.pages);
 scanf("%f", &bk1.price);

 printf("\n Book Name : %s", bk1.book);
 printf("\n No. of Pages : %d", bk1.pages);
 printf("\n Book Price : %.2f", bk1.price);
 getche();
}
```

**OUTPUT:**

```
Enter Book name, pages, price :C 500 450
Book Name : C
No. of Pages : 500
Book Price : 450.00
```

**Explanation:**

This program is the same as the previous one. Instead of initializing values, the values are read using `scanf()` statements and structure variables. The `printf()` statement displays the contents of structure variables.

### 13.4 STRUCTURE WITHIN STRUCTURE

We can take any data type for declaring structure members like `int`, `float`, `char`. In the same way, we can also take objects of one structure as a member in another structure. Thus, a structure within a structure can be used to create complex data applications (see Figure 13.2). The syntax of the structure within the structure is as follows:

```
struct time
{
 int second;
 int minute;
 int hour;
};
struct t
{
 int carno;
 struct time st;
 struct time et;
};
struct t player;
```

- 13.5 Write a program to read and display the car number, starting time and reaching time. Use structure within structure.

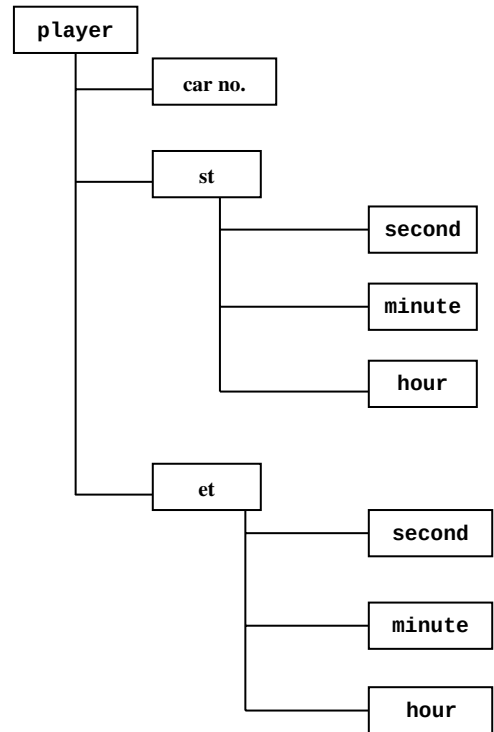


**Figure 13.2** Structure within structure

```
void main()
{
 struct time
 {
 int second;
 int minute;
 int hour;
 };
 struct t
 {
 int carno;
 struct time st;
 struct time rt;
 };
 struct t r1;
 clrscr();
 printf("\n Car No. Starting Time Reaching Time :");
 scanf("%d", &r1.carno);
 scanf("%d %d %d", &r1.st.hour, &r1.st.minute, &r1.st.second);
 scanf("%d %d %d", &r1.rt.hour, &r1.rt.minute, &r1.rt.second);
 printf("\n\tCar No. \tStarting Time \tReaching Time\n");
 printf("\t%d\t", r1.carno);
```

```

printf("\t%d:%d:%d\t\t", r1.st.hour, r1.st.minute, r1.st.second);
printf("\t%d:%d:%d", r1.rt.hour, r1.rt.minute, r1.rt.second);
getche();
}

```

**OUTPUT:**

```

Car No. Starting Time Reaching Time :125 2 50 30 3 50 25
Car No. Starting Time Reaching Time
125 2:50:30 3:50:25

```

**Explanation:**

In the above program, two structures are defined. The first structure is time that contains member fields `int second`, `int minute` and `int hour`. The second structure is `t` whose member fields are `carno`, `st` and `rt`. The variables `st` and `rt` are the objects of the first structure. Using these two variables, it is possible to access the member variables of the first structure. The variable `r1` is an object of the structure `t`. The statement `r1.carno` accesses the variable `carno` of the structure `t` and the statement `r1.st.hour` accesses the variable `hour` of the structure `time`. Here, the dot operator is used twice because we are accessing time structure through the object of `t` structure.

- 13.6 Write a program to enter full name and date of birth of a person and display the same. Use the nested structure.

```

void main()
{
 struct name
 {
 char first[10];
 char second[10];
 char last [10];
 };
 struct b_date
 {
 int day;
 int month;
 int year;
 };
 struct data
 {
 struct name nm;
 struct b_date bt;
 };
 struct data r1;
 clrscr();
 printf("\n Enter Name (First / Second / Last)\n");
 scanf("%s %s %s", r1.nm.first, r1.nm.second, r1.nm.last);
 printf("\n Enter Birth Date Day / Month / Year\n");
 scanf("%d %d %d", &r1.bt.day, &r1.bt.month, &r1.bt.year);
 printf("Name : %s %s %s\n", r1.nm.first, r1.nm.second, r1.nm.last);
}

```

```

 printf("Birth Date : %d.%d.%d", r1.bt.day, r1.bt.month, r1.bt.year);
 getch();
}

```

**OUTPUT:**

```

Enter Name(First / Second / Last)
Ram Sham Pande
 Enter Birth Date Day / Month / Year
12 12 1980
Name : Ram Sham Pande
Birth Date : 12.12.1980

```

**Explanation:**

In the above example, structure name, `b_date` and data are defined. The structure data has member variables of type `name` and `b_date` structures, respectively. Therefore, this type of structure is called a nested structure. The variable `r1` is a variable of type data structure. Using `scanf()` statement, the program reads data from the keyboard. In the same way, using the `printf()` statement entered data is displayed on the screen. Here, the dot (.) operator is used twice as we are accessing variables of structure which are inside the another structure.

## 13.5 ARRAY OF STRUCTURES

As we know an array is a collection of similar data types. In the same way, we can also define an array of structure. In such type of array, every element is of structure type. Array of structure can be declared as follow:

```

struct time
{
 int second;
 int minute;
 int hour;
} t[3];

```

In the above example, `t[3]` is an array of three elements containing three objects of time structure. Each element of `t[3]` has structure of time with three members that are second, minute and hour. A program is explained as given below.

➤ 13.7 Write a program to create an array of structure objects.

```

void main()
{
 int k;
 struct time
 {
 int second;
 int minute;

```

```

 int hour;
};
struct t
{
 int carno;
 struct time st;
 struct time rt;
};
struct t r1[3];
clrscr();
printf("\nCar No. Starting Time Reaching Time :\n\n");
printf("\t hh:mm:ss\t hh:mm:ss \n");
for (k=0;k<3;k++)
{
 scanf("%d", &r1[k].carno);
 scanf("%d %d %d", &r1[k].st.hour, &r1[k].st.minute, &r1[k].
 st.second);
 scanf("%d %d %d", &r1[k].rt.hour, &r1[k].rt.minute, &r1[k].
 rt.second);
}
printf("\n\tCar No. \tStarting Time \tReaching Time\n");
for (k=0;k<3;k++)
{
 printf("\n\t%d\t", r1[k].carno);
 printf("\t%d:%d:%d\t\t", r1[k].st.hour, r1[k].st.minute, r1[k].
 st.second);
 printf("\t%d:%d:%d", r1[k].rt.hour, r1[k].rt.minute, r1[k].rt.second);
}
getche();
}

```

**OUTPUT:**

| Car No. | Starting Time<br>hh:mm:ss | Reaching Time :<br>hh:mm:ss |
|---------|---------------------------|-----------------------------|
| 120     | 2 20 25                   | 3 25 58                     |
| 121     | 3 25 40                   | 4 40 25                     |
| 122     | 4 30 52                   | 5 40 10                     |

| Car No. | Starting Time | Reaching Time : |
|---------|---------------|-----------------|
| 120     | 2 :20: 25     | 3 :25:58        |
| 121     | 3 :25: 40     | 4 :40: 25       |
| 122     | 4 :30: 52     | 5 :40:10        |

**Explanation:**

In the above program, two structures `time` and `t` are declared. An array of three-elements `r1[3]` is defined. The first `for` loop executes three times and the `scanf()` statement reads data through the keyboard for each element of the object. The second `for` loop and the `printf()` statements within it display the contents of the array of object with their elements.

- 13.8 Write a program to display names, roll numbers and grades of three students who have appeared in an examination. Declare the structure of name, roll nos and grade. Create an array of structure objects. Read and display the contents of the array.

```

void main()
{
 int k=0;
 struct stud
 {
 char name[12];
 int rollno;
 char grade[2];
 };
 struct stud st[3];
 while(k<3)
 { clrscr();
 gotoxy(2,4);
 printf("Name : ");
 gotoxy(17,4);
 scanf("%s",st[k].name);

 gotoxy(2,5);
 printf("Roll No. : ");
 gotoxy(17,5);
 scanf("%d",&st[k].rollno);

 gotoxy(2,6);
 printf("Grade :");
 gotoxy(17,6);
 scanf("%s",st[k].grade);
 st[k].grade[1]='\0';
 puts(" press any key..");
 getch();
 k++;
 }
 k=0;
 clrscr();
 printf("\nName\t Rollno\ Grade\n");
 while(k<3)
 {
 printf("\n%s\t %d\t %s",st[k].name,st[k].rollno,st[k].grade);
 k++;
 }
}

```

**OUTPUT:**

| Name     | Rollno. | Grade |
|----------|---------|-------|
| Sanjay   | 125     | A     |
| Rajesh   | 126     | A+    |
| Srinivas | 127     | A     |

Note: This output is displayed after entering the data of three students.

**Explanation:**

In the above program, structure `stud` is declared with its members `char name [12]`, `int rollno` and `char grade[2]`. The array `st[3]` of structure `stud` is declared. The first



while loop and `scanf()` statements within the loop are used for repetitive data reading. The second while loop and `printf()` statements within it display the contents of the array.

## 13.6 POINTER TO STRUCTURE

We know that the pointer is a variable that holds the address of another data variable. The variable may be of any data type, i.e. `int`, `float` or `double`. In the same way, we can also define pointer to structure. Here, starting address of the member variables can be accessed. Thus, such pointers are called structure pointers.

### *Example:*

```
struct book
{
 char name[25];
 char author[25];
 int pages;
};
struct book *ptr;
```

In the above example, `*ptr` is pointer to structure `book`. The syntax for using pointer with member is as given below:

(1) `ptr->name` (2) `ptr->author` (3) `ptr->pages`.

By executing these three statements, starting address of each member can be estimated.

➤ 13.9 Write a program to declare pointer to structure and display the contents of the structure.

```
void main()
{
 struct book
 {
 char name[25];
 char author[25];
 int pages;
 };
 struct book b1={"JAVA COMPLETE REFERENCE", "P.NAUGHTON", 886};
 struct book *ptr;
 ptr=&b1;
 clrscr();
 printf("\n %s by %s of %d pages", b1.name, b1.author, b1.pages);
 printf("\n %s by %s of %d pages", ptr->name, ptr->author, ptr->pages);
}
```

### **OUTPUT:**

```
JAVA COMPLETE REFERENCE by P.NAUGHTON of 886 pages
JAVA COMPLETE REFERENCE by P.NAUGHTON of 886 pages
```

### *Explanation:*

In the above program, the function `printf()` statement prints structure elements by calling them as usual. In the second `printf()` statement to print the structure elements using pointer an arrow operator  $\rightarrow$  (– and > together) is used instead of dot (.) operator. The reason is that the `ptr` is not a structure variable but pointer to a structure.

- 13.10 Write a program to declare pointer as members of structure and display the contents of the structure.

```
void main()
{
 struct boy
 {
 char *name;
 int *age;
 float *height;
 };
 static struct boy *sp;
 char nm[10]="Mahesh";
 int ag=20;
 float ht=5.40;
 sp->name=nm;
 sp->age=&ag;
 sp->height=&ht;
 clrscr();
 printf("\n Name = %s", sp->name);
 printf("\n Age = %d", *sp->age);
 printf("\n Height = %.2f", *sp->height);
}
```

**OUTPUT:**

```
Name = Mahesh
Age = 20
Height = 5.40
```

**Explanation:**

In the above program, structure boy is declared. The members of structure boy are pointers. The pointer \*sp is a pointer to structure boy. Another three ordinary variables char nm[10]="Mahesh", int ag=20 and float ht=5.40 are declared and initialized. The addresses of these ordinary variables are assigned to structure variables using arrow operator. Using the printf() statement, the contents of structure variables are displayed.

- 13.11 Write a program to declare a pointer as members of structure and display the contents of the structure without using the arrow (→) operator.

```
include <string.h>
void main()
{
 struct boy
 {
 char *name;
 int *age;
```

```

 float *height;
 };
 struct boy b;
 char nm[10]="Somesh";
 int ag=20;
 float ht=5.40;
 strcpy(b.name, nm);
 b.age=&ag;
 b.height=&ht;
 clrscr();
 printf("\n Name = %s", b.name);
 printf("\n Age = %d", *b.age);
 printf("\n Height = %g", *b.height);
}

```

**OUTPUT:**

```

Name = Somesh
Age = 20
Height = 5.4

```

**Explanation:**

This program is the same as the previous one. Here, no pointer is declared in structure. Hence, using **dot operator**, we can display the contents of the structure.

- 13.12 Write a program to display the contents of the structure using the ordinary pointer.

```

void main()
{
 int *p;
 struct num
 {
 int a;
 int b;
 int c;
 };
 struct num d;
 d.a=2;
 d.b=3;
 d.c=4;
 p=&d.a;
 clrscr();
 printf("\n a=%d", *p);
 printf("\n b=%d", *(++p));
 printf("\n c=%d", *(++p));
}

```

**OUTPUT:**

```

a=2
b=3
c=4

```

**Explanation:**

In the above program, `*p` and structure `num` are declared. The structure `num` has three members `a`, `b` and `c` of integer data type and initialized with the values 2, 3 and 4, respectively. We know that structure variables are stored in successive memory locations. If we get starting address of one variable then we can display next elements. The address of variable `a` is assigned to pointer `p`. By applying unary `*` and `++` operators a pointer is incremented and values are displayed.

## 13.7 STRUCTURE AND FUNCTIONS

Like variables of standard data type structure variables can be passed to the function by value or address. The syntax of the same is as follows:

```
struct book
{
 char name[35];
 char author[35];
 int pages;
} b1;

void main()
{

 show(&b1);

}
show (struct book *b2)
{

}
```

Whenever a structure element requires to pass to any other function, it is essential to declare the structure outside the `main()` function, i.e. global.

In the above example, structure `book` is declared before `main()`. It is a global structure. Its member elements are `char name[35]`, `char author[35]` and `int pages`. They can be accessed by all other functions.

- 13.13 Write a program to pass address of a structure variable to a user-defined function and display the contents.

```
/* passing address of structure variable */
struct book
{
 char name[35];
 char author[35];
 int pages;
};
```

```

void main()
{
 struct book b1= {"JAVA COMPLETE REFERENCE", "P.NAUGHTON", 886};
 show(&b1);
}
show(struct book *b2)
{
 clrscr();
 printf("\n %s by %s of %d pages", b2->name, b2->author, b2->pages);
}

```

**OUTPUT:**

JAVA COMPLETE REFERENCE by P.NAUGHTON of 886 pages

**Explanation:**

In the above program, structure book is defined before `main()`. In the `main()` function, `b1` is a structure object declared and initialized. The address of object `b1` is passed to function `show()`. In the function `show()`, the address is assigned to pointer `*b2` that is a pointer to the structure book. Thus, using the `→` operator contents of structure elements are displayed.

- 13.14 Write a program to pass structure elements to function `print()` and print the elements.

```

void main()
{
 struct boy
 {
 char name[25];
 int age;
 int wt;
 };
 struct boy b1={"Amit", 20, 25};
 print(b1.name, b1.age, b1.wt);
}
print(char *s, int t, int n)
{
 clrscr();
 printf("\n %s %d %d", s, t, n);
}

```

**OUTPUT:**

Amit 20 25

**Explanation:**

In the above example, the structure name has member variables like a character array `name[25]`, `age` and `wt` of integer type. We have passed the base address of `name` to the function & `age` and `wt` by call by value. Thus, here values are passed using call by reference and call by value methods. Instead of passing each element, one can also pass the entire structure into the function. This is shown in the below-given program.

- 13.15 Write a program to pass the entire structure to the user-defined function.

```
/* passing entire structure to function */
struct boy
{
 char name[25];
 int age;
 int wt;
};
void main()
{
 struct boy b1={"Amit",20,25};
 print(b1);
}
print(struct boy b)
{
 clrscr();
 printf("\n %s %d %d",b.name,b.age,b.wt);
 return 0;
}
```

**OUTPUT:**

Amit 20 25

**Explanation:**

In the above program, structure `boy` is defined outside the `main()`. So it is global and every function can access it. The object defined on structure `boy` `b1` is passed to function `print()`. The formal argument (object) of function `print()` receives the contents of object `b1`. Thus, using dot operator contents of individual elements are displayed.

## 13.8 typedef

We can create new data type by using `typedef`. The statement `typedef` is to be used while defining the new data type. The syntax is as follows:

```
typedef type dataname;
```

Here, `type` is the datatype and `dataname` is the user-defined name for that type.

```
typedef int hours;
```

Here, `hours` is another name for `int` and now we can use `hours` instead of `int` in the program as follows:

```
hours hrs;
```

- 13.16 Write a program to create user-defined data type `hours` on `int` data type and use it in the program.

```
#define H 60
void main()
{
 typedef int hours;
 hours hrs;
 clrscr();
 printf("Enter Hours: ");
 scanf("%d", &hrs);
 printf("\nMinutes = %d", hrs*H);
 printf("\nSeconds = %d", hrs*H*H);
}
```

**OUTPUT:**

```
Enter Hours: 2
Minutes = 120
Seconds = 7200
```

**Explanation:**

In the above example, with `typedef` we have declared `hours` as an integer data type. Immediately after the `typedef` statement `hrs` is a variable of `hours` data type which is similar to `int`. Further program calculates minutes & seconds using `hrs` variable.

- 13.17 Write a program to create string data type.

```
void main()
{
 typedef char string[20];
 string a=" Hello ",b;
 clrscr();
 puts("Enter Your Name :");
 gets(b);
 printf("%s %s",a,b);
}
```

**OUTPUT:**

```
Enter Your Name : KAMAL
Hello KAMAL
```

**Explanation:**

In the above program, `string[20]` is a user-defined character data type. It defines two variables `a` and `b` having 20 character space for each. Similarly, we can also use `typedef` for defining the structure. One of its example is as given below.

- 13.18 Create a user-defined data type from structure. The structure should contain the variables such as `char`, `int`. By using these variables, display name, sex and `acno` of an employee.

```
void main()
{
 typedef struct
```

```

{
 char name[20];
 char sex[2];
 int acno;
}info;
info employee={"Sanjay","M",125};
clrscr();
printf("\nName\t Sex\t A/c No.\n");
printf("%s\t",employee.name);
printf(" %s\t",employee.sex);
printf(" %d\n",employee.acno);
}

```

**OUTPUT:**

|        |     |         |
|--------|-----|---------|
| Name   | Sex | A/c No. |
| Sanjay | M   | 125     |

**Explanation:**

In the above program, `info` is another user-defined name for defining the structure. Here, `info` is used for defining the structure variables such as employee's name, sex and age. The user can understand the rest of the program.

- 13.19 Create a user-defined data type from structure. The structure should contain the variables such as `char`, `int`. By using these variables display `name`, `sex` and `acno` of two employees. Use array of structures.

```

void main()
{
 typedef struct
 {
 char name[20];
 char sex[2];
 int acno;
 }info;
 info employee[2];
 int k;
 clrscr();
 for (k=0;k<2;k++)
 {
 printf(" Name of the Employee :");
 scanf("%s",employee[k].name);
 printf(" Sex :");
 scanf("%s",employee[k].sex);
 printf("A/c No. :");
 scanf("%d", &employee[k].acno);
 }
 printf("\nName\t Sex\t A/c No.\n");
 for (k=0;k<2;k++)
 {
 printf("%s\t",employee[k].name);

```



```

 printf(" %s\t", employee[k].sex);
 printf(" %d\n", employee[k].acno);
 }
}

```

**OUTPUT:**

```

Name of the Employee : AJAY
Sex : M
A/c No. : 122
Name of the Employee : ANITA
Sex : F
A/c No. : 124
NAME SEX A/C NO.
AJAY M 122
ANITA F 124

```

**Explanation:**

In the above program, using `typedef` statement the user-defined data type `info` is created. The `info` data type contains two characters and one integer field. An array `employee [2]` is declared based on the `info` data type. The first `for` loop and `scanf()` statements within it read data. The second `for` loop and the `printf()` statements within it display the contents of the array on the screen.

- 13.20 Write a program to define the structure containing the details of the employee. The structure may contain first, middle, last name, place, city and pin code. Use `typedef` to create data type. Display the records of two employees.

```

void main()
{
 int j;
 typedef struct
 {
 char first[20];
 char middle[20];
 char last [15];
 char city[15];
 int pincode;
 }name;
 name person[2];
 clrscr();
 for (j=0;j<2;j++)
 {
 printf("\nRecord No. : %d",j+1);
 printf("\nFirst Name :");
 scanf("%s",person[j].first);
 printf("Middle Name :");
 scanf("%s",person[j].middle);
 printf("Last Name :");
 scanf("%s",person[j].last);
 printf("City & Pincode");
 }
}

```

```

 scanf("%s %d", person[j].city, &person[j].pincode);
 }
 for (j=0; j<2; j++)
 {
 printf("\n\nFirst Name : %s", person[j].first);
 printf("\nMiddle Name : %s", person[j].middle);
 printf("\nLast Name : %s", person[j].last);
 printf("\nCity & Pincode : %s - %d", person[j].city, person[j].
 pincode);
 }
}

```

**OUTPUT:**

```

Record No. : 1
First Name : Jay
Middle Name : Mohan
Last Name : Deshmukh
City & Pincode : Nanded 431 602
Record No. : 2
First Name : Vijay
Middle Name : Kamal
Last Name : Nandedkar
City & Pincode : Nanded 431 602
Record No. : 1
First Name : Jay
Middle Name : Mohan
Last Name : Deshmukh
City & Pincode : Nanded 431 602
Record No. : 2
First Name : Vijay
Middle Name : Kamal
Last Name : Nandedkar
City & Pincode : Nanded 431 602

```

***Explanation:***

In the above program, the `typedef` structure contains employee details like first, middle, last name, city and pincode. Based on this structure, the user defines data type `name`, which is further used in the program to accept and print the relevant data fed by the user.

## 13.9 BIT FIELDS

Bit field provides exact amount of bits required for storage of values. If a variable value is 1 or 0 then we need a single bit to store it. In the same way if the variable is expressed between 0 and 3 then the two bits are sufficient for storing these values. Similarly if a variable assumes values between 0 and 7 then three bits will be sufficient to hold the variable and so on. The number of bits required for a variable is specified by non-negative integer followed by a colon.

To hold the information, we use the variables. The variables occupy minimum one byte for `char` and two bytes for `int`. Instead of using complete integer if bits are used, and space of memory can be saved. For example, to know the information about the vehicles, the following information has to be stored in the memory:

- (i) PETROL VEHICLE
- (ii) DIESEL VEHICLE
- (iii) TWO\_WHEELER VEHICLE
- (iv) FOUR\_WHEELER VEHICLE
- (v) OLD MODEL
- (vi) NEW MODEL

In order to store the status of the above information, we may need two bits for the type of fuel as to whether vehicle is of petrol or diesel type. Three bits for its type as to whether the vehicle is a two- or a four-wheeler. Similarly, three bits for model of the vehicle. Total bits required for storing the information would be eight bits i.e. 1 byte. It means that the total information can be packed into a single byte. Eventually, bit fields are used for conserving the memory. The amount of memory saved by using bit fields will be substantial which is proved from the above example.

However, there are restrictions on bit fields when arrays are used. Arrays of bit fields are not permitted. Also, the pointer cannot be used for addressing the bit field directly, although the use of the member access operator (->) is acceptable.

The unnamed bit fields could be used for padding as well as for alignment purposes. The structure for the above problem would be as follows:

```
struct vehicle
{
 unsigned type: 3;
 unsigned fuel: 2;
 unsigned model: 3;
};
```

The colon (:) in the above declaration tells to the compiler that bit fields are used in the structure and the number after it indicates how many bits are required to allot for the field.

- 13.21 Write a program to store the information of vehicles. Use bit fields to store the status of information.

```
define PETROL 1
define DIESEL 2
define TWO_WH 3
define FOUR_WH 4
define OLD 5
define NEW 6
void main()
{
 struct vehicle
 {
 unsigned type : 3;
 unsigned fuel : 2;
 unsigned model :3;
 };
 struct vehicle v;
 v.type=FOUR_WH;
```

```

 v.fuel=DISEL;
 v.model=OLD;
 clrscr();
 printf("\n Type of Vehicle : %d",v.type);
 printf("\n Fuel : %d",v.fuel);
 printf("\n Model : %d",v.model);
 getch();
}

```

**OUTPUT:**

```

Type of Vehicle : 4
Fuel : 2
Model : 5

```

***Explanation:***

In the above program, using **#define** macros are declared. The information about the vehicle is indicated between integers 1 to 6. The structure vehicle is declared with bit fields. The number of bits required for each member is initialized. As per the program, for type of vehicle requires 3 bits, fuel requires 2 bits and model requires 3 bits. An object v is declared. Using the object bits fields are initialized with data. The output of the program displays integer value stored in the bit fields, which can be verified with macro definitions initialized at the beginning of the program.

- 13.22 Write a program to display the examination result of students using bit fields.

```

define PASS 1
define FAIL 0
define A 0
define B 1
define C 2
void main()
{
 struct student
 {
 char *name;
 unsigned result : 1;
 unsigned grade : 2;
 };
 struct student v;
 v.name="Sachin";
 v.result=PASS;
 v.grade =C;
 clrscr();
 printf("\n Name : %s",v.name);
 printf("\n Result : %d",v.result);
 printf("\n Grade : %d",v.grade);
 getch();
}

```

**OUTPUT:**

```
Name : Sachin
Result : 1
Grade : 2
```

***Explanation:***

The above program is the same as the previous one. Only the member variables and bits assigned are different.

**13.10 ENUMERATED DATA TYPE**

The `enum` is a keyword. It is used for declaring enumeration types. The programmer can create his/her own data type and define what values the variables of these data types can hold. This enumeration data type helps in reading the program.

Consider the example of 12 months of a year.

```
enum month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

This statement creates a user-defined data type. The keyword `enum` is followed by the tag name `month`. The enumerators are the identifiers *Jan*, *Feb*, *Mar*, *Apr*, *May* and so on. Their values are constant unsigned integers and starts from 0. The identifier `Jan` refers to 0, `Feb` to 1 and so on. The identifiers are not to be enclosed with quotation marks. Please also note that integer constants are also not permitted.

- 13.23 Write a program to create enumerated data type for 12 months. Display their values in integer constants.

```
void main()
{
 enum month {Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct,
 Nov, Dec};
 clrscr();
 printf("\nJan = %d", Jan);
 printf("\nFeb = %d", Feb);
 printf("\nJune = %d", June);
 printf("\nDec = %d", Dec);
}
```

**OUTPUT:**

```
Jan = 0
Feb = 1
June = 5
Dec = 11
```

**Explanation:**

In the above program, enumerated data type month is declared with 12-month names within two curly braces. The compiler assigns 0 value to the first identifier and 11 to the last identifier. Using `printf()` statement, the constants are displayed for different identifiers. By default, the compiler assigning values from 0 onwards. Instead of 0 the programmer can initialize his/her own constant to each identifier. The below given program illustrates this concept.

- 13.24 Write a program to create enumerated data type for 12 months. Initialize the first identifier with 1. Display their values in integer constants.

```
void main()
{
 enum month
 {Jan=1, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec};
 clrscr();
 printf("\nJan = %d", Jan);
 printf("\nFeb = %d", Feb);
 printf("\nJune = %d", June);
 printf("\nDec = %d", Dec);
}
```

**OUTPUT :**

```
Jan = 1
Feb = 2
June = 6
Dec = 12
```

**Explanation:**

In the above program, enumerated data type month is declared with 12 months names within two curly braces. The compiler starts assigning values from 1 to first identifier because the first identifier is initialized with constant 1 and 12 to last identifier. Using `printf()` statement, the constants are displayed for different identifiers.

- 13.25 Write a program to display the name of month using enumerated data type. Initialize the enumerated data with user-defined constant.

```
void main()
{
 int f;
 enum month {Jan=1, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec};
 clrscr();
 for (f=Jan; f<=Dec; f++)
 switch(f)
 {
 case Jan :
 printf("\n January");
```

```
 break;
 case Feb :
 printf("\n February");
 break;
 case Mar :
 printf("\n March");
 break;
 case Apr :
 printf("\n April");
 break;
 case May :
 printf("\n May");
 break;
 case June :
 printf("\n June");
 break;
 case July :
 printf("\n July");
 break;
 case Aug :
 printf("\n August");
 break;
 case Sep :
 printf("\n September");
 break;
 case Oct :
 printf("\n October");
 break;
 case Nov :
 printf("\n November");
 break;
 case Dec :
 printf("\n December");
 break;
 }
}
```

**OUTPUT :**

```
January
February
March
April
May
June
July
August
September
October
November
December
```

**Explanation:**

In the above program, enumerated data type `month` is defined and initialized with month names. In the declaration, the enumerated month is initialized with constant 1, so that the counting of enum constants starts from 1. The constants are used in the `for` loop. The `switch()` case executes appropriate case and displays month names.

- 13.26 Write a program to use enumerated data type.

```
include <string.h>
void main()
{
 enum capital
 {
 Mumbai, Hyderabad, Bangalore
 };
 struct state
 {
 char name[15];
 enum capital c;
 };
 struct state s;
 strcpy(s.name, "Andhra Pradesh");
 s.c=Hyderabad;
 clrscr();
 printf("\n State : %s", s.name);
 printf("\n Capital : %d", s.c);
 if (s.c==Hyderabad)
 printf("\n Hyderabad is the Capital city of %s", s.name);
}
```

**OUTPUT:**

```
State : Andhra Pradesh
Capital : 1
Hyderabad is the Capital city of Andhra Pradesh
```

**Explanation:**

In the above program, enum data type `capital` is defined and initialized with three identifiers. They are `Mumbai`, `Hyderabad` and `Bangalore`. The Structure `state` is declared with two members such as `name` and `capital`. The structure variables are assigned with values `Andhara Pradesh` and `Hyderabad`. The `printf()` statement displays the contents of structure variables. If the structure variable `s.c` contains value `Hyderabad` then a message is displayed otherwise not.

- 13.27 Write a program to identify the type of entered character whether it is a letter, digit or other symbol. Use enumerated data type.

```
include <ctype.h>
void main()
```



```

{
 char ch;
 int f;
 enum ctype
 {
 Letter, Digit, Other
 };
 clrscr();
 printf("\n Enter any character ");
 ch=getch();
 f=isalpha(ch);
 if(f!=0)
 printf("\n %c is type %d symbol ",ch,Letter);
 else
 {
 f=isdigit(ch);
 if (f!=0)
 printf("\n %c is type %d symbol ",ch,Digit);
 else
 printf("\n %c is type %d symbol ",ch,Other);
 }
}

```

**OUTPUT:**

```

Enter any character
= is type 2 symbol

```

**Explanation:**

In the above program, enum data type `ctype` is declared with identifiers letter, digit and anything else. A character is entered through the keyboard. The macros `isalpha()` and `isdigit()` check the character whether it is a letter, digit or anything else. Depending upon the type of character entered appropriate identifiers are displayed. In the output, = is entered. It is neither a letter nor a digit. Hence, it comes under other type whose enumerated value is 2. Thus, we get the type entered character with its value.

## 13.11 union

Union is a variable, which is similar to the structure. It contains the number of members like structure but it holds only one object at a time. In the structure, each member has its own memory location whereas the members of unions have the same memory locations. It can accommodate one member at a time in a single area of storage. union also contains members of types `int`, `float`, `long`, arrays, pointers. It allocates fixed specific bytes of memory for access of data types irrespective of any data type.

The union requires bytes that are equal to the number of bytes required for the largest members. For example, if the union contains `char`, `integer` and `long integer` then the number of bytes reserved in the memory for the union is 4 bytes. An example is illustrated below for proper understanding.

- 13.28 Write a program to find the size of `union` and the number of bytes reserved for it.

```
void main()
{
 union result
 {
 int marks;
 char grade;
 };
 struct res
 {
 char name[15];
 int age;
 union result perf;
 }
 data;
 clrscr();
 printf("Size of union : %d\n", sizeof(data.perf));
 printf("Size of Structure : %d\n", sizeof(data));
}
```

**OUTPUT:**

```
Size of union : 2
Size of Structure : 19
```

**Explanation:**

Union contains two variables of data types `int` and `char`, respectively. The `int` and `char` require 2 and 1 bytes, respectively, for storage. According to the theory, the `union` reserves two bytes because `int` takes more space than the `char` data type.

The structure `res` is defined immediately after `union`, which contains data types `char`, `int` and `union` variable. The size of the structure is printed which is nothing but the sum of 15 bytes of `char` array, two bytes of `int` and two bytes of `union` variable `perf`. Thus, the total size of structure is 19.

## 13.12 CALLING BIOS AND DOS SERVICES

We can access BIOS and DOS services through C language. A few simple examples are illustrated by making the use of BIOS and DOS services. The programmer does not have direct access to the CPU registers. However, the C language provides an interface that allows the programmer to access any BIOS or DOS service. The C language defines a `union` type called 'REGS' and structure type called 'SREGS' in the file 'DOS.H' which escort the C compiler. These data types are used to pass parameters to the CPU registers. In order to have access to these definitions, the programmer has to include 'dos.h' header file.

The `union` type 'REGS' contains the CPU registers. These registers are used for holding the data temporarily. The microprocessors 8086, 8088, 80186, 80286 and 80386 use a variety of registers for arithmetic and logical operations. They are also used to receive instructions and pass data to and from memory.

The CPU registers are 16 bits in size in 8086. They are AX, BX, CX, DX, SI, DI and CFLAG, which is carry flag. The size of CFLAG is also 16 bits. The structure that defines these 16 bit registers is called 'x'. The 'x' stands for a register pair. Size of each lower or upper register is 8 bits. Thus, the size of pair register 'x' is 16 bits.

The **union** also allows us to refer 8 bit registers of above type processors. These 8 bits registers are AL, AH, BL, BH, CL, CH, DL and DH. The structure that defines these 8 bit registers is called 'h', where 'h' stands for a high register, i.e. 8 bit register.

There are also segment registers in the above processors. The size of segment registers is 16 bits. These segment registers DS, ES, CS and SS. The structure type 'SREGS' defines these segment registers.

The C language provides the functions such as `intdos()`, `intdosx()` and `segread()` to access the DOS services. The `intdos()` is used for calling DOS interrupt. The `intdos()` invokes a DOS function by issuing software interrupt INT 21H.

We can also invoke BIOS services by using C functions such as `int86()` and `int86x()`. The `int86()` stands for 8086 interrupt. The `int86()` functions invoke a BIOS functions by issuing a software interrupt. The `int86()` function can be invoked with the following three arguments.

Interrupt type number corresponding to ROM-BIOS service.

Two arguments of union type REGS. They are 'x' and 'h'.

The `int86()` function takes the values of input registers from one 'REGS' union and returns the output registers in another. The `int86x()` function is similar to `int86()`. The difference is `int86x()` requires an input argument in DS or ES register. Below given are a few examples of `int86()` functions.

➤ 13.29 Write a program to find the memory size of the computer.

```
include <dos.h>
/* Finding Memory Size */
void main()
{
 union REGS in,out;
 int86(18, &in, &out);
 clrscr();
 printf("\n Total Memory = %d KB",out.x.ax);
}
```

#### OUTPUT:

Total Memory = 640 KB

#### Explanation:

In the above program, 'in' and 'out' are objects of type union REGS. In this example, we need not send any values to the ROM-BIOS function. Thus, nothing is to be passed to any of the registers before invoking `int86()`. Here, the first argument 18 is the interrupt number followed by two arguments of union REGS. The result obtained here is the memory size displayed by calling ax register. Here, extended memory of PC is displayed.

➤ 13.30 Write a program to display the system time at a specified cursor position.

```
include <dos.h>
/* Positioning Cursor on the console */
void main()
{
```

```

union REGS in,out;
in.h.ah=2;
in.h.dh=20;
in.h.dl=15;
int86(16,&in,&out);
printf("%s",_TIME_);
}

```

**OUTPUT:**

19:42:49

**Explanation:**

In the above program, the register **ah** contains service number, **dh** contains the row number and **dl** contains the column number. Here, interrupt number is 16 which is placed in the **int86()** function with two **union** variables. Thus, the system time is displayed at the specified position.

- 13.31 Write a program to change the cursor in different sizes.

```

include <dos.h>
/* Cursor sizes */
void main()
{
 union REGS in,out;
 in.h.ah=0x01;
 in.h.ch=0;
 in.h.cl=10;
 int86(0x10,&in,&out);
 getch()
}

```

**Explanation:**

In the above program, 0x01 is the service number under interrupt number 0x10. The **CH** register contains starting scan line and **CL** contains ending scan line. After execution of the function, the cursor is modified. Programmer can verify after running the program.

- 13.32 Write a program to create a directory using DOS interrupt.

```

include <dos.h>
void main()
{
 union REGS in,out;
 char dir[11];
 clrscr();
 puts("\n Enter Directory Name : ");
}

```

```
scanf("%s",dir);
in.x.dx=(int) &dir;
in.h.ah=0x39;
intdos(&in,&out);
if(out.x.cflag!=0)
printf("Directory %s not created",dir);
else
printf("\n Directory %s created",dir);
}
```

**OUTPUT:**

```
Enter Directory Name : XYZ
Directory XYZ created
```

**Explanation:**

The program prompts to enter the name of the directory to be created. The name entered is stored in the variable `dir`. The address of `dir` is assigned to register `DX`. Now, the `DX` register points to directory name. The DOS service number `0x39` is called to create the directory. If the carry flag is zero then it means the operation is successfully carried out otherwise an error occurs. Appropriate messages are displayed on the screen.

- 13.33 Write a program to display the given character on the screen.

```
include <dos.h>
/* displays specified character */
void main()
{
 union REGS in,out;
 in.h.ah=02;
 in.h.dl=67;
 intdos(&in,&out);
}
```

**OUTPUT:**

```
c
```

**Explanation:**

In the above program, the register variable `ah` is initialized with service number `02` which controls standard output devices. The register variable `dl` is initialized with ASCII code of character that is to be displayed on the screen. After the execution of `intdos()` function, the given character is displayed on the screen.

- 13.34 Write a program to display the attributes of a file using the DOS interrupt.

```
include <dos.h>
/* Getting attribute of a file */
void main()
```

```

{
 union REGS in,out;
 char file[15];
 int mask,j;
 clrscr();
 puts("\n Enter a file name : ");
 scanf ("%s",file);
 in.x.dx=(int) &file;
 in.h.al=0;
 in.h.ah=0x43;
 intdos(&in,&out);
 if (out.x.cflag!=0)
 printf("\n File not found");
 else
 {
 printf("\n File attributes of %s are\n",file);
 if (out.h.cl==0)
 puts("\n Normal file");
 else
 {
 mask=1;
 for (j=1;j<=6;++j)
 {
 switch(out.h.cl & mask)
 {
 case 1 : puts ("[*] Read Only");
 break;
 case 2 : puts ("[*] Hidden ");
 break;
 case 4 : puts ("[*] System");
 break;
 case 8 : puts ("[*] Volume Label");
 break;
 case 16 : puts ("[*] Subdirectory");
 break;
 case 32 : puts ("[*] Archive");
 break;
 }
 mask=mask*2;
 }
 }
 }
}

```

**OUTPUT:**

```

Enter a file name :
C:\IO.SYS
File attributes of C:\IO.SYS are
[*] Read Only
[*] Hidden
[*] System

```

**Explanation:**

In the above program, a file name is entered whose attributes are to be displayed. The DX register contains the address of the entered file. The register AL is initialized with 0 to get the file attributes. The register AH is always initialized with service number. Here, it is initialized with 0x43. The result obtained is stored in the AL register. Checking the status of CL register by AND (&) operation, we can find the attributes of a file. Table 13.2 describes Bit status of CL and file attributes.

**Table 13.2** Bit status and attributes

| Bit Status of CL | Attribute      |
|------------------|----------------|
| 0                | Read only file |
| 1                | Hidden file    |
| 2                | System file    |
| 3                | Volume label   |
| 4                | Sub-directory  |
| 5                | Archive        |
| 6 & 7            | Unused         |

- 13.35 Write a program to delete a file using the DOS interrupt.

```
include <dos.h>
void main()
{
 union REGS in,out;
 char file[11];
 puts("\n Enter a file name : ");
 scanf("%s",file);
 in.h.ah=0x41;
 in.x.dx=(int)file;
 intdos(&in, &out);
 if(out.x.cflag==0)
 puts("\n File successfully deleted");
 else
 puts("\n File could not be deleted");
}
```

**OUTPUT:**

```
Enter a file name : ABC.TXT
File successfully deleted
```

**Explanation:**

In the above program, a file name is entered which is to be deleted. The ah register contains service number 0x41 which performs this task. The register DX contains addresses of the file. If the function is successful then the flag register contains 0 otherwise it contains non-zero values. The statuses of cflag registers are checked using the if statement and appropriate messages are displayed.

This program works correctly only with small memory model. To delete a file the segment address of the file is stored in the DS register whereas the offset address is stored in the DX register. In this program, without separating the offset and segment address, address is assigned to register DX. It works correctly in small memory model because there is only one data segment in small memory model and DS register always contains the segment address. If you try this program with other memory models then it would not run successfully. To make this program suitable for all the memory models, we need

to make the address of file far. This address is then separated into offset and segment parts. For that C provides two macros `FP_SEG` and `FP_OFF`. The description of far pointer and `FP_SEG` and `FP_OFF` macros is beyond the scope of this book.

### 13.13 UNION OF STRUCTURES

We know that one structure can be nested within another structure. In the same way, a `union` can be nested within another `union`. We can also create a structure in a `union` or vice versa.

➤ 13.36 Write a program to use structure within union. Display the contents of structure elements.

```
void main()
{
 struct x
 {
 float f;
 char p[2];
 };
 union z
 {
 struct x set;
 };
 union z st;
 st.set.f=5.5;
 st.set.p[0]=65;
 st.set.p[1]=66;

 clrscr();
 printf("\n %g", st.set.f);
 printf("\n %c", st.set.p[0]);
 printf("\n %c", st.set.p[1]);
}
```

#### OUTPUT:

```
5.5
A
B
```

#### *Explanation:*

In the above program, structure `x` is defined. The `union z` contains structure member as its member variable. The variable `st` is an object of `union z`. The member variables of structure `x` are assigned with certain values. Using objects of structure and `union` with a dot operator, the member contents of member variables are displayed.

## SUMMARY

This chapter explains the concept of structure and programs on it. One of the powerful features of C language is that it supports the creation of a structure. The various features of a structure are described at the beginning of this chapter. For the beginners the concepts and examples on structures are



given in an easy way and a step-by-step process is adopted. The various titles under structure such as how the structures are declared, initialized, structure within the structure, array of structure, pointer to structure are elaborated. Moreover, the point that how functions are defined in structure is also illustrated with good examples. The `typedef` facility can be used for creating user-defined data types and illustrated with many examples. Enumerated data type, `union` are the important subtitles of this chapter. Enumerated data type provides user-defined data types. `union` is a principal method by which a programmer can derive dissimilar data types. The last but not the least the DOS and ROM-BIOS functions and their applications are explained. The user is advised to go through Appendix C whereby using these functions a number of programs can be developed. The readers will benefit a lot if they execute the programs given in this chapter.

## EXERCISES

### I True or false:

1. A structure is a set of different data type.
2. A `struct` is a keyword.
3. Structure elements can be accessed directly.
4. The dot operator can be used to access the `struct` variable.
5. The structure definition must be terminated by a semi-colon.
6. The declaration and initialization of structure variable can be done at once.
7. Structure can be defined inside another structure (nested).
8. It is possible to pass structure elements to function.
9. The array structure elements cannot be declared.
10. The `ptr` is a pointer to structure and it can access elements using `->` operator.
11. Bit fields provide the exact amount of bits required for storage.
12. Bit fields are always defined of signed type.
13. The `enum` keyword is used to define enumerated data type.
14. The value of enumerated data type starts from 1.
15. The structure elements are stored in the separate memory locations.
16. The `union` elements are stored at random memory locations.
17. The `union` has common storage space for all its variable.
18. The `union` requires more space as compared to a structure.

### II Select the appropriate option from the multiple choices given below:

1. Identify the most appropriate sentence to describe unions
  - (a) unions contain members of different data types which share the same storage area in memory
  - (b) unions are like structures
  - (c) unions are less frequently used in the program
  - (d) unions are used for set operations
2. The member variable of a structure is accessed by using
  - (a) dot (`.`) operator
  - (b) arrow (`->`) operator
  - (c) asterisk (`*`) operator
  - (d) ampersand (`&`) operator
3. The structure combines variables of
  - (a) dissimilar data types
  - (b) similar data types
  - (c) unsigned data types
  - (d) None of the above
4. The `typedef` statement is used for
  - (a) declaring user-defined data types
  - (b) declaring variant variables
  - (c) for typecasting of variables
  - (d) None of the above

5. The number of bytes required for enumerated data type in memory is
- 2 bytes
  - 4 bytes
  - 1 byte
  - 3 bytes
6. The service number is always initialized in the register
- AH
  - AL
  - BH
  - AX
7. The `intdos()` function invokes interrupt number
- 0x21
  - 0x17
  - 0x18
  - None of the above
8. The `int86()` function invokes
- ROM-BIOS services
  - DOS services
  - Both (a) and (b)
  - None of the above
9. Interrupt 0x21 is a
- software interrupt
  - hardware interrupt
  - Both (a) and (b)
  - None of the above
10. The union holds
- one object at a time
  - multiple objects
  - Both (a) and (b)
  - None of the above
11. Bit fields are used only with
- unsigned `int` data type
  - float data type
  - char data type
  - `int` data type
12. Observe the following program neatly and choose the appropriate `printf()` statement from the options.
- ```
struct month
{
    char *month;
};
void main()
{
    struct month m={"March"};
    ----
}
```
- `printf ("\n Month : %s", m.month);`
 - `printf ("\n Month : %s", m->month);`
 - `printf ("\n Month : %s", m.*month);`
 - `printf ("\n Month : %s", *m.month);`
13. What will be the value of *m* displayed on execution of the following program?
- ```
struct bit
{
 unsigned int m :3;
};
void main()
{
 struct bit b={8};
 clrscr();
 printf ("\n m = %d", b.m);
}
```
- m*=0
  - m*=8
  - m*=3
  - None of the above
14. The size of the structure in bytes occupied in the following program will be
- ```
struct bit
{
    unsigned int m :4;
    unsigned int x :4;
    int k;
    float f;
};
struct bit b;
```

```

void main()
{
    clrscr();
    printf ("\n Size of structure
in Bytes = %d", sizeof(b));
}

```

- (a) 8
- (b) 10
- (c) 7
- (d) 4

III Attempt the following programming exercises:

- Write a program to define a structure with tag `state` with fields `state name`, `number of districts` and `total population`. Read and display the data.
- Write the program (1)-using pointer to structure.
- Define a structure with tag `population` with fields `Men` and `Women`. Create a structure within a structure using `state` and `population` structure. Read and display the data.
- Modify the program developed using exercise (3). Create array of structure variables. Read and display the 5 records.
- Create user-defined data type equivalent to `int`. Declare three variables of its type. Perform arithmetic operations using these variables.
- Create enumerated data type `logical` with `TRUE` and `FALSE` values. Write a program to check whether the entered number is positive or negative. If the number is positive display 0 otherwise display 1. Use enumerated data type `logical` to display 0 and 1.
- Write a program to accept records of the different states using array of structures. The structure should contain `char state` and number of `int engineering collages`, `int medical collages`, `int management collages` and `int universities`. Calculate the total collages and display the state, which is having the highest number of collages.
- Write a program to check the status of the printer using the `int86()` function. The details are as given below:
 Interrupt - 0x17
 Inputs - AH = 0x02
 - DX = printer port number (0=LPT1, 1=LPT2, 2=LPT3)
 Returns - AH = Completion / nonsuccess code
 Completion values are
 Bit 0=1 : time out
 Bit 3=1 : I/O mistake
 Bit 4=1 : Printer selected
 Bit 5=1 : Out of paper
 Bit 6=1 : Printer acknowledge
 Bit 7=1 : Printer not engaged
- Write a program to reboot the system. Use the following data with `int86()` function.
 (a) Interrupt - 0x19
 (b) Input - Void (Nothing)

IV Answer the following questions:

- What is a structure in C? How is a structure declared?
- What is the use of the keyword `struct`? Explain the use of the dot operator.
- How are structure elements stored in memory?
- Explain nested structure. Draw a diagram to explain a nested structure.
- How are arrays of structure variables defined? How are they beneficial to the programmers?
- How are structure elements accessed using pointer? Which operator is used?
- Is it possible to pass structure variable to function? Explain in detail the possible ways.
- How are user-defined data types defined?

9. Explain the importance of bit fields. How do bit fields save memory space?
10. Explain the enumerated data type.
11. What is a union in C? How is data stored using union?
12. What are the differences between union and structure?
13. Explain REGS and SREGS unions. List any five CPU registers of each union type.
14. Explain `int86()` and `intdos()` functions. How they used to interact with hardware?

V What is/are the output/s of the following programs?

1.

```
void main()
{
    struct emp
    {
        char name[30];
        int age;
        int sal;
    };
    struct emp
    e={"Satish",28,20000};
    clrscr();
    printf("\n Name : %s",e.
    name);
    printf("\n Age : %d",e.age);
    printf("\n Salary : %d",e.sal);
    getche();
}
```

OUTPUT:

```
Name : Satish
Age : 28
Salary : 20000
```

2.

```
void main()
{
    struct emp
    {
        char name[30];
        int age;
        int sal;
    };
    struct emp e={"Santosh",28,
    7000};
    clrscr();
    printf ("\n SIZE OF e :
    %d",sizeof(e));
}
```

OUTPUT:

```
SIZE OF e : 34
```

3.

```
void main()
{
    struct emp
```

```
{
    int k;
    int a;
    int s;
};
struct emp e={1,2,4};
e.k++;
clrscr();
printf("%d %d %d",e.k,e.a,e.s);
}
```

OUTPUT:

```
2 2 4
```

4.

```
# include <string.h>
void main()
{
    struct boy
    {
        char *name;
        int *age;
        float *height;
    };
    struct boy b;
    char nm[10]="Somesesh";
    int ag=20;
    float ht=5.40;
    strcpy(b.name,nm);
    b.age=&ag;
    b.height=&ht;
    clrscr();
    printf("\n Name = %s",b.name);
    printf("\n Age = %d",*b.age);
    printf("\n Height = %g",*b.
    height);
}
```

OUTPUT:

```
Name = Somesh
Age = 20
Height = 5.4
```

5.

```
void main()
{
```

```

int *p;
struct num
{
    int a;
    int b;
    int c;
};
struct num d;
d.a=2;
d.b=3;
d.c=4;
p=&d.a;

```

```

clrscr();
printf("\n a=%d", *p);
printf("\n b=%d", *(++p));
printf("\n c=%d", *(++p));
}

```

OUTPUT:

```

a  =  2
b  =  3
c  =  4

```

VI Find the bug/s in the following programs:

1.

```

void main()
{
    struct book
    {
        int pages;
        float price;
    }
    *b1;
    b1.pages=500;
    b1.price=255;
    clrscr();
    printf("\n Pages : %d", b1.
pages);
    printf("\n Price : %g", b1.
price);
}

```

1. *b1 is a pointer and hence operator -> is to be used in place of dot (.).

2.

```

void main()
{
    struct book
    {
        int pages;
        float price;
    }
    b1
    b1.pages=450;
    b1.price=450.55;
    clrscr();
}

```

2. structure should be terminated by semicolon.

3.

```

void main()
{

```

```

struct book
{
    int pages;
    float price;
}
b2, *b3;
b3=&b2;
b3->pages=400;
b2.price=700.00;
clrscr();
printf("\n Pages : %d",
b2->pages);
printf("\n Price :%g",
b3->price);
}

```

3. variable b2 is not pointer and hence use of -> is invalid.

4.

```

void main()
{
    struct book
    {
        int pages;
        float price;}
    b2;
    b2.pages=400;
    b2.price=700.00;
    clrscr();
    show (b2);
}
show (struct book b1)
{
    printf("\n Pages : %d", b1.
pages);
    printf("\n Price : %f", b1.
price);
}

```

- bug: structure should be defined outside all functions

5.

```
void main()
{
    typedef hours int;
    hours hrs;
    hrs=120/60;
    clrscr();
    printf("\n 120 minutes=%d
    Hours",hrs);
}
```

 {
 unsigned type : 3;
 int fuel :32;
 };
 struct vehicle v;
 v.type=2;
 v.fuel=1;
 clrscr();
 printf("\nd",v.type);
 printf("\nd",v.fuel);}
 bug: type def int hours
 6. int fuel: 4; is expected
6.

```
void main()
{
    struct vehicle
```

ANSWERS

I True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	T	2.	T	3.	F	4.	T	5.	T
6.	F	7.	T	8.	T	9.	F	10.	T
11.	T	12.	F	13.	T	14.	F	15.	T
16.	F	17.	T	18.	F				

II Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	a	4.	a	5.	a
6.	a	7.	a	8.	a	9.	a	10.	a
11.	a	12.	a	13.	a	14.	c		

V What is/are the output/s of the following programs?

Q.	Ans.	Q.	Ans.
2.	SIZE OF e : 34	3.	2 2 4
4.	Name = Sameer Age = 24 Height = 4.9	5.	a=2 b=3 c=4

VI Find the bug/s in the following programs:

Q.	Ans.
1.	*b1 is a pointer and hence operator -> is to be used in place of dot.
2.	Structure should be terminated by semi-colon.
3.	Bug: variable b2 is not pointer and hence use of -> is invalid.
4.	Structure should be defined outside all functions.
5.	Bug : typede int hours.
6.	int type can hold 32 (4 bytes) data int fuel: 4; is expected.

CHAPTER

14 Files

Chapter Outline

- 14.1** Introduction of a File
- 14.2** Definition of File
- 14.3** Streams and File Types
- 14.4** Steps for File Operations
- 14.5** File I/O
- 14.6** Structures Read and Write
- 14.7** Other File Function
- 14.8** Searching Errors in Reading/Writing Files
- 14.9** Low-Level Disk I/O
- 14.10** Command Line Arguments
- 14.11** Application of Command Line Arguments
- 14.12** Environment Variables
- 14.13** I/O Redirection

14.1 INTRODUCTION OF A FILE

A file is nothing but collection of records. Record is group of related data items. All data items related to students, employees, customers etc is nothing but records. In other words, file is a collection of numbers, symbols & text placed onto the secondary devices like hard disk, pen drive, compact disk etc. Files are stored permanently on to the disk and one can access them for further monitoring/processing if needed. In the next few paragraphs/sections, we will learn how files can be read and modified as per requirements.

14.2 DEFINITION OF FILE

A file can be considered as a stream of characters. A file can be a set of records that can be accessed through the set of library functions. These functions are available in `stdio.h`.

The data can be stored in secondary storage devices such as floppy or hard disk. Figure 14.1 shows communication between keyboard, Ram and Secondary device.

As shown in the figure data, read from keyboard is stored in variables. Variables are created in RAM (type of primary memory). Variables hold the data temporarily in the program. On the

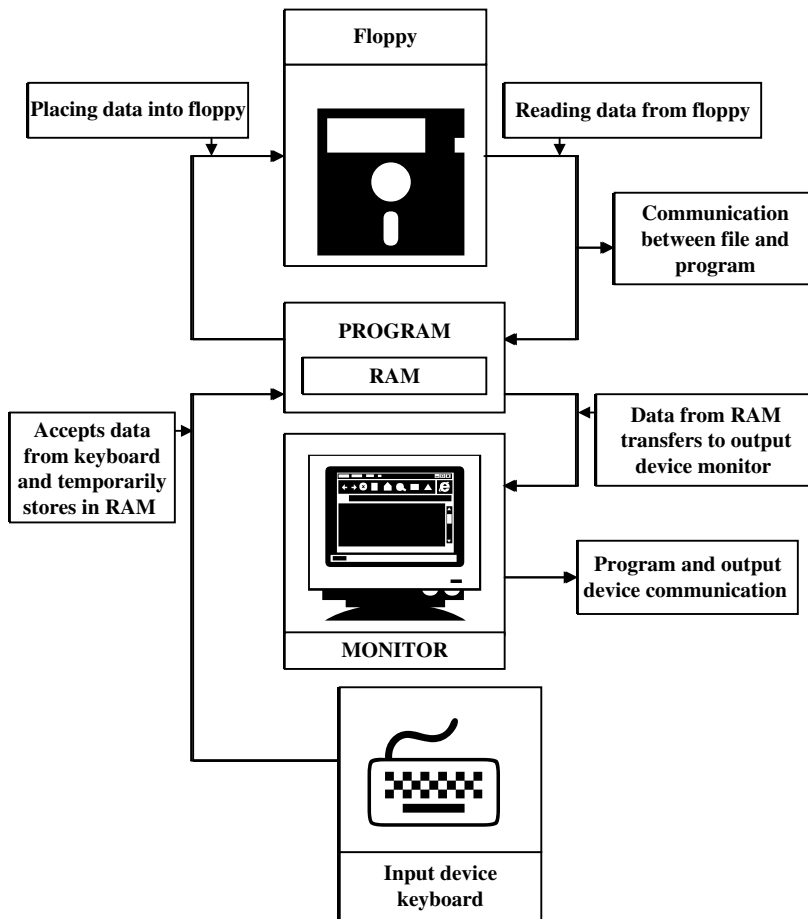


Figure 14.1 Communication between program, file and output device

execution of a program and manipulating data the original data may be lost. It is needed to store the data permanently on to the secondary devices. On applying disk I/O operations, all variables created in RAM can be stored to the secondary storage device such as hard disk or floppy disk.

In Figure 14.1, a floppy disk is shown. It is also possible to read data from it or secondary storage devices. When data is read from such devices it is placed in the RAM and then using console I/O operations it is transferred to screen.

A file is nothing but the accumulation of data stored on the disk created by the user. The programmer assigns file name. The file names are unique and are used to identify the file. No two files can have the same names in the same directory. There are various kinds of files such text files, program files, data files, executable file and so on. Data files contain a combination of numbers, alphabets, symbols called data.

Data communication can be performed between program and output devices or files and program. File streams are used to carry the communication among above-mentioned devices. The stream is nothing but the flow of data in bytes in sequence. If data is received from input devices in sequence then it is called source stream and when the data is passed to output devices then it is called destination stream. Figure 14.2 shows the input and output streams. The input stream brings data to the program and the output stream collects data from program. In this way, input stream extracts data from the file and transfers it to the program while the output stream stores the data into the file provided by the program.

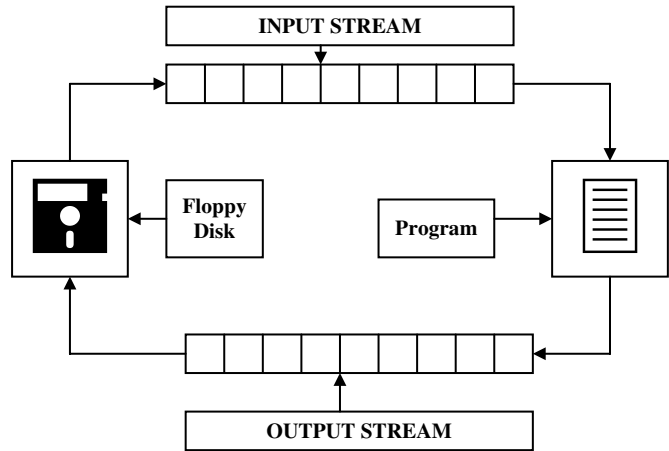


Figure 14.2 Input and output streams

14.3 STREAMS AND FILE TYPES

Data flow from program to a file or vice versa is done with stream, which is a series of bytes. In order to perform I/O operations on the files, reading or writing a file is done with streams. Reading and writing data is done with streams. The streams are designed to allow the user to access the files efficiently. A stream is a file and using physical device like keyboard, information is stored in the file. The FILE object uses these devices such as keyboard, printer and monitor.

The FILE object contains all the information about stream like current position, pointer to any buffer, error and end of file (EOF). Using this information of object, C program uses pointer, which is returned from the stream function `fopen()`. A function `fopen()` is used for opening a file.

14.3.1 | File Types

There are two types of files. (1) Sequential file and (2) Random accesses file.

- (1) **Sequential file:** In this type of file, data are kept sequentially. If we want to read the last record of the file then it is expected to read all the records before it. It takes more time for accessing the records. Or if we desire to access the 10th record then the first nine records should be read sequentially for reaching the 10th record.

- (2) **Random access file:** In this type, data can be read and modified randomly. If the user desires to read the last records of a file, directly the same records can be read. Due to random access of data, it takes access time less as compared to the sequential file.

14.4 STEPS FOR FILE OPERATIONS

Most of the compilers support file handling functions. C language also supports numerous file-handling functions that are available in standard library. A few of these functions are listed in Table 14.1. All the functions that are supported by C for file handling are not listed. The library functions supported by C are very exhaustive ones and hence, all functions are not provided. One should check the C library functions before using them.

Table 14.1 *File functions*

<i>Function</i>	<i>Operation</i>
<code>fopen()</code>	Creates a new file for read/write operation.
<code>fclose()</code>	Closes a file associated with file pointer.
<code>closeall()</code>	Closes all files opened with <code>fopen()</code> .
<code>fgetc()</code>	Reads the character from current pointer position and advances the pointer to the next character.
<code>getc()</code>	Same as <code>fgetc()</code> .
<code>fprintf()</code>	Writes all types of data values to the file.
<code>fscanf()</code>	Reads all types of data values from a file.
<code>putc()</code>	Writes characters one-by-one to a file.
<code>fputc()</code>	Same as <code>putc()</code> .
<code>gets()</code>	Reads a string from the file.
<code>puts()</code>	Writes a string to the file.
<code>putw()</code>	Writes an integer to the file.
<code>getw()</code>	Reads an integer from the file.
<code>fread()</code>	Reads structured data written by <code>fwrite()</code> function.
<code>fwrite()</code>	Writes block of structured data to the file.
<code>fseek()</code>	Sets the pointer position anywhere in the file.
<code>feof()</code>	Detects the end of file.
<code>ferror()</code>	Reports error occurred while read/write operations.
<code>perror()</code>	Prints compilers error messages along with user-defined messages.
<code>ftell()</code>	Returns the current pointer position.
<code>rewind()</code>	Sets the record pointer at the beginning of the file.
<code>unlink()</code>	Removes the specified file from the disk.
<code>rename()</code>	Changes the name of the file.

14.4.1 | Opening of File

A file has to be opened before read and write operations. Opening of a file creates a link between the operating system and the file functions. The name of the file and its mode of operation are to be indicated to the operating system. This important task is carried out by the structure `FILE` that is defined in `stdio.h` header file. So this file must be included.

When a request is made to the operating system for opening a file, it does so by granting the request. If request is granted then the operating systems points to the structure `FILE`. In case, the request is not granted, it returns `NULL`. That is why the following declaration before opening of the file is to be made:

```
FILE *fp
```

where `fp` is the file pointer.

Each file that we open has its own `FILE` structure. The information that is there in the file may be its current size, and its location in memory. The only one function to open a file is `fopen()`.

Syntax:

```
FILE *fp;
fp=fopen("data.txt", "r");
```

Here `fp` is a pointer variable that contains the address of the structure `FILE` that has been defined in the header file `stdio.h`. It is necessary to write `FILE` in the uppercase. The function `fopen()` will open a file 'data.txt' in read mode. The C compiler reads the contents of the file because it finds the read mode ("r"). Here, "r" is a string and not a character. Hence, it is enclosed with double quotes and not with single quotes.

The `fopen()` performs the following important tasks:

- (i) It searches the disk for opening of the file.
- (ii) In case the file exists, it loads the file from the disk into memory. If the file is found with huge contents then it loads the file part by part.
- (iii) If the file is not existing this function returns a `NULL`. `NULL` is a macro defined in the header file 'stdio.h'. This indicates that it is unable to open a file. There may be the following reasons for failure of `fopen()` functions: (A) when the file is in protected or hidden mode and (B) the file may be in use by another program.
- (iv) It locates a character pointer, which points the first character of the file. Whenever a file is opened the character pointer points to the first character of the file.

14.4.2 | Reading a File

Once the file is opened using `fopen()`, its contents are loaded into the memory (partly or wholly). The pointer points to the very first character of the file. The `fgetc()` function is used to read the contents of the file. The syntax for `fgetc()` is as follows:

```
ch=fgetc(fp);
```

where `fgetc()` reads the character from current pointer position and advances the pointer position so that the next character is pointed. The variable `ch` contains the character read by `fgetc()`. There are also other functions to read the contents of file, which are explained below.

14.4.3 | Closing a File

The file that is opened from the `fopen()` should be closed after the work is over, i.e. we need to close the file after reading and writing operations. In other words, the file must be closed after operations.

Also, whenever an opened file needs to be reopened in other mode in such a case also the opened file must be closed first. Closing the file enables to wash out all its contents from the RAM buffer and further the link is disconnected from the file.

Example:

The function to close a file is

```
fclose(fp);
```

This statement closes the file associated with file pointer `fp`. This function closes one file at a time. In order to close all files function syntax used is as follows:

```
fcloseall();
```

This function closes all the opened files and returns the number of files closed. It does not require any argument.

14.4.4 | Text Modes

1. w(write)

This mode opens a new file on the disk for writing. If the file already exists, its contents will be overwritten without confirmation. If the concerned file is not found, a new file is created.

Syntax:

```
fp=fopen ("data.txt", "w");
```

Here `data.txt` is the file name and `"w"` is the mode.

2. r(read)

This mode searches a file and if it is found the same is loaded into the memory for reading from the first character of the file. The file pointer points to the first character and reading operation begins. If the file does not exist then compiler returns `NULL` to the file pointer. Using pointer with the `if` statement we can prompt the user regarding failure of operation. The syntax of read mode is as follows:

Syntax:

```
fp=fopen("data.txt", "r");
if(fp==NULL)
printf("File does not exist");

OR

if(fp=(fopen("data.txt", "r"))==NULL)
printf("File does not exist");
```

Here `data.txt` is opened for reading only. If the file does not exist the `fopen()` returns `NULL` to file pointer '`fp`'. Using the `NULL` value of `fp` with the `if` statement, we can prompt the user for failure of `fopen()` function. A program is illustrated below giving the use of `"w"` and `"r"` modes.

➤ 14.1 Write a program to write data to text file and read it.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char c='.';
    clrscr();
    fp=fopen("data.txt","w");
    if(fp==NULL)
    {
        printf("Can not open file");
        exit(1);
    }
    printf("Write data & to stop press '.' :");
    while(c!='.')
    {
        c=getche();
        fputc(c,fp);
    }
    fclose(fp);
    printf("\n Contents read :");
    fp=fopen("data.txt","r");
    while(!feof(fp))
        printf("%c",getc(fp));
}
```

OUTPUT:

Write data & to stop press '.': ABCDEFGHIJK.
 Contents read: ABCDEFGHIJK.

Explanation:

In the above program, the file named "data.txt" is opened in write mode. The characters are read from the keyboard and stored in variable 'c'. Using `fputc()` the characters are written to a file until '.' (dot) is pressed. The same file is closed and then it is re-opened in read mode. On reopening of the file, character pointer sets to the beginning of the file. The contents of the file will be displayed on the screen using `getc()`.

3. a(append)

This mode opens a pre-existing file for appending data. The data appending process starts at the end of the opened file. The file pointer points to the last character of the file. If the file does not exist, then a new file is opened, i.e. if the file does not exist then the mode of "a" is the same as "w". Due to some or other reasons if file is not opened in such a case NULL is returned. File opening may be impossible due to insufficient space on to the disk and some other reasons. Syntax for opening a file with append mode is as follows:

Syntax:

```
fp=fopen("data.txt","a");
```

Here, if `data.txt` file already exists, it will be opened. Otherwise a new file will be opened with the same name.

- 14.2 Write a program to open a pre-existing file and add information at the end of file. Display the contents of the file before and after appending.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char c;
    clrscr();
    printf("Contents of file before appending :\n");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    {
        c=fgetc(fp);
        printf("%c",c);
    }
    fp=fopen("data.txt","a");
    if(fp==NULL)
    {
        printf("File can not appended");
        exit(1);
    }
    printf("\n Enter string to append :");
    while(c!='.')
    {
        c=getche();
        fputc(c,fp);
    }
    fclose(fp);
    printf("\n Contents of file After appending :\n");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    {
        c=fgetc(fp);
        printf("%c",c);
    }
}
```

OUTPUT:

```
Contents of file before appending :
String is terminated with '\0'.
Enter string to append :
This character is called as NULL character.
Contents of file After appending :
String is terminated with '\0'.
This character is called as NULL character.
```

4. w+ (Write + read)

This mode starts for file search operation on the disk. In case the file is found, its contents are destroyed. If the file is not found, a new file is created. It returns `NULL` if it fails to open the file. In this file mode, new contents can be written and thereafter reading operation can be done.

Example:

```
fp=fopen("data.txt", "w+");
```

In the above example, `data.txt` file is open for reading and writing operations.

➤ 14.3 Write a program to use `w+` mode for writing and reading of a file.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char c=' ';
    clrscr();
    fp=fopen("data.txt", "w+");
    if(fp==NULL)
    {
        printf("Can not open file");
        exit(1);
    }
    printf("Write data & to stop press '.' :");
    while(c!='.')
    {
        c=getche();
        fputc(c, fp);
    }
    rewind(fp);
    printf("\n Contents read :");
    while(!feof(fp))
        printf("%c", getc(fp));
}
```

OUTPUT:

```
Write data & to stop press '.': ABCDEFGHIJK.
Contents read : ABCDEFGHIJK.
```

Explanation:

Instead of using separate read and write modes, one can use `w+` mode to perform both the operations. Hence, in the above program `w+` is used. At first writing operation is done. It is not essential to close the file for reading the contents of it. We need to set the character pointer at the beginning. Hence, `rewind()` function is used. The advantage of using `w+` is to reduce the number of statements.

5. a+ (append + read)

In this file operation mode the contents of the file can be read and records can be added at the end of file. A new file is created in case the concerned file does not exist. Due to some or the other reasons if a file is unable to open then NULL is returned.

Example:

```
fp=fopen("data.txt", "a+");
```

Here data.txt is opened and records are added at the end of file without affecting the previous contents.

➤ 14.4 Write a program to open a file in append mode and add new records in it.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char c='.';
    clrscr();
    fp=fopen("data.txt", "a+");
    if(fp==NULL)
    {
        printf("Can not open file");
        exit(1);
    }
    printf("Write data & to stop press '.' :");
    while(c!='.')
    {
        c=getche();
        fputc(c, fp);
    }
    printf("\n Contents read :");
    rewind(fp);
    while(!feof(fp))
        printf("%c", getc(fp));
}
```

OUTPUT:

Write data & to stop press '.' : This is append and read mode.
Contents read : This is append and read mode.

Explanation:

In the above program, a file named "data.txt" is opened in read and append mode (a+). If a file does not exist, a new file is created. Write operation is performed first and the contents are read thereafter. Before reading character pointer is set at the beginning of file using `rewind()`.

Notes:

1. In case read operation is done after write operation, character pointer should be set to beginning of the file using `rewind()`.
2. In case write/append operation is done after read operation, it is not essential to set the character pointer at the beginning of file.

6. r+ (read + write)

This mode is used for both reading and writing. We can read and write the record in the file. If the file does not exist, the compiler returns NULL to the file pointer. It can be written as follows:

Example:

```
fp=fopen("data.dat", "r+");
if(fp==NULL)
printf("File not found");
```

In the above example, `data.dat` is opened for the read and write operation. If `fopen()` fails to open the file then it returns NULL. The `if` statements check the value of file pointer `fp`; and if it contains NULL then a message is printed and program terminates.

- 14.5 Write a program to open a file in read/write mode in it. Read and write new information in the file.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char c=' ';
    clrscr();
    fp=fopen("data.txt", "r+");
    if(fp==NULL)
    {
        printf("Can not open file");
        exit(1);
    }
    printf("\n Contents read :");
    while(!feof(fp))
    printf("%c", getc(fp));
    printf("Write data & to stop press '.' :");
    while(c!='.')
    {
        c=getche();
        fputc(c, fp);
    }
}
```

OUTPUT:

```
Contents read: Help me.
Write data & to stop press '.': I am in trouble.
```

Explanation:

In the above example, file is opened in read and write mode (r+). The `getc()` function reads the contents of file which is printed through `printf()` function. The `getche()` function reads characters from the keyboard and the read characters are written to the file using `fputc()` function.

14.4.5 | Binary Modes

When numerical data is to be transferred to disk from RAM, the data occupies more memory space on disk. For example, a number 234567 needs 3 bytes memory space in RAM and when transferred to disk requires 6 bytes memory space. For each numerical digit one byte space is needed. Hence, total requirement for the number 234567 would be 6 bytes. Thus, text mode is inefficient for storing large amount of numerical data because space occupation by it is large. Only solution to this inefficient memory use is to open a file in binary mode, which takes lesser space than text mode. Few binary modes are described below.

1. wb(write) :

This mode opens a binary file in write mode.

Example:

```
fp=fopen("data.dat", "wb");
```

Here data.dat file is opened in binary mode for writing.

2. rb(read) :

This mode opens a binary file in read mode.

Example:

```
fp=fopen("data.dat", "rb");
```

Here data.dat file is opened in binary mode for reading.

- 14.6 Write a program to open a file for read/write operation in binary mode. Read and write new information in the file.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char c=' ';
    clrscr();
    fp=fopen("data.dat", "wb");
    if (fp==NULL)
    {
        printf("Can not open file");
        exit(1);
    }
    printf("Write data & to stop press '.' :");
    while(c!='.')
    {
        c=getche();
        fputc(c, fp);
    }
    fclose(fp);
    fp=fopen("data.dat", "rb");
```

```
printf("\n Contents read :");
while(!feof(fp))
printf("%c",getc(fp));
}
```

Explanation:

This program is the same as explained earlier. The only difference is that the file-opening mode is binary.

3. ab(append):

This mode opens a binary file in append mode i.e. data can be added at the end of file.

Example:

```
fp=fopen("data.dat", "ab");
```

Here "data.dat" file is opened in append mode.

4. r+b(read + write):

This mode opens a pre-existing file in read and write mode, i.e. a file can be read and written.

Example:

```
fp=fopen("data.dat", "r+b");
```

Here, the file "data.dat" is opened for reading and writing in binary mode.

5. w+b(read + write):

This mode creates a new file in read and write mode.

Example:

```
fp=fopen("data.dat", "w+b");
```

Here, the file "data.dat" is created for reading and writing in binary mode.

6. a+b(append+ write):

This mode opens a file in append mode, i.e. data can be written at the end of file. If file does not exist then a new file is created.

Example:

```
fp=fopen("data.dat", "a+b");
```

Here the file "data.dat" is opened in append mode and data can be written at the end of file.

14.5 FILE I/O

1. fprintf() :

This function is used for writing characters, strings, integers, floats, etc. to the file. The `fprint()` function is used for writing characters in various formats. Hence, this function is called the formatted function. It contains one more parameter that is file pointer, which points the opened file.

The operation of `fprintf()` and `fscanf()` functions are identical to that of `printf()` and `scanf()` except that former function works with files.

The format of `fprintf()` is as follows:

```
fprintf() (fp, "control string", text);
```

where `fp` is a file pointer associated with an opened file in write mode. The text can be variables, constants or strings.

A programming example on this function is as follows.

- 14.7 Write a program to open a text file and write some text using `fprintf()` function. Open the file and verify the contents.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    char text[30];
    fp=fopen("Text.txt", "w");
    clrscr();
    printf("Enter Text Here :");
    gets(text);
    fprintf(fp, "%s", text);
    fclose(fp);
}
```

OUTPUT:

Enter Text Here: Have a nice day.

Explanation:

In the above program, `fprintf()` function writes the string to the file pointed by `fp`. The string is collected through `gets()` function into character array `name[30]`.

2. fscanf() :

This function reads characters, strings, integer, floats, etc. from the file pointed by file pointer. This is also a formatted function. A program is illustrated below based on this.

The syntax of this function is as follows:

```
fscanf(fp, "control string", text);
```

With this statement reading operations from the designated file is done.

- 14.8 Write a program to enter data into the text file and read the same. Use “w+” file mode. Use `fscanf()` to read the contents of the file.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    char text[15];
    int age;
    fp=fopen("Text.txt", "w+");
    clrscr();
    printf("Name\t AGE\n ");
    scanf("%s %d", text, &age);
    fprintf(fp, "%s %d", text, age);
    printf("Name\t AGE\n ");
    fscanf(fp, "%s %d", text, &age);
    printf("%s\t%d\n", text, age);
    fclose(fp);
}
```

OUTPUT:

Name	AGE
AMIT	12
Name	AGE
AMIT	12

Explanation:

In the above program, `fscanf()` reads the data from the file named “Text.txt”.

3. `getc()` :

This function reads a single character from the opened file and moves the file pointer. It returns EOF, if end of file is reached.

For example, in the statement `c=getc(f);`, a character is read from the file whose file pointer is `f`. A program is illustrated below on the basis of this function.

- 14.9 Write a program to read the contents of the file using `getc()` function.

```
# include <stdio.h>
# include <process.h>
# include <conio.h>
void main()
{
    FILE *f;
    char c;
    clrscr();
    f=fopen("list.txt", "r");
    if(f==NULL)
```

```

{
    printf("\nCannot open file");
    exit(1);
}
while((c=getc(f))!=EOF)
    printf("%c",c);
fclose(f);
}

```

OUTPUT:

```

aman
akash
amit
ajay
ankit

```

Explanation:

In the above program, the `getc(f)` reads character from the file “list.txt”. Some text is to be written before reading this file.

4. `putc()`:

This function is used to write a single character into a file. If an error occurs then it returns EOF.

For example, in the statement `putc(c, fp)`;, a character contained in character variable `c` is written in the file whose file pointer is `fp`. A program is illustrated below on the basis of this function.

➤ 14.10 Write a program to write some text into the file using `putc()` function.

```

# include <stdio.h>
# include <conio.h>
void main()
{
    int c;
    FILE *fp;
    clrscr();
    printf("\n Enter Few Words '*' to Exit\n");
    fp=fopen("words.doc", "w");
    while( (c=getchar())!='*')
        putc(c, fp);
    fclose(fp);
}

```

OUTPUT:

```

Enter Few Words '*' to Exit
This is saved into the file *

```

Explanation:

The `putc()` function writes character read through `getchar()` in the file "words.doc". User should enter "*" to stop reading character.

5. `fgetc()` :

This function is similar to the `getc()` function. It also reads a character and increments the file pointer position. If any error or end of file is reached then it returns EOF.

➤ 14.11 Write a program to read a C program file and count the following in the complete program:

- (a) Total number of statements.
- (b) Total number of included files.
- (c) Total number of blocks and brackets.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fs;
    int i=0,x,y,c=0,sb=0,b=0;
    clrscr();
    fs=fopen("PRG2.C","r");
    if (fs==NULL)
    {
        printf("\n File opening error.");
        exit(1);
    }
    while((x=fgetc(fs))!=EOF)
    {
        switch(x)
        {
            case ';' :
                c++;
                break;
            case '{' :
                sb++;
                break;
            case '(' :
                b++;
                break;
            case '#' :
                i++;
                break;
        }
    }
    fclose (fs);
    printf("\n Summary of 'C' Program\n");
```



```

printf("=====");
printf("\n Total Statements   : %d ",c+i);
printf("\n Include Statements : %d",i);
printf("\n Total Blocks {}    : %d",sb);
printf("\n Total Brackets ()   : %d",b);
printf("\n=====");
}

```

OUTPUT:

```

Total Statements : 25
Include Statements : 4
Total Blocks {} : 5
Total Brackets () : 25

```

Explanation:

In the above program, the `fgetc()` function reads the file "prg2.c" character by character and returns the read character to variable 'x'. The variable 'x' is passed then to `switch case`. Depending upon the contents of the variable 'x' respective counter is incremented in each `CASE` statement. Thus, at last all summary is printed.

6. fputc() :

This function writes the character to the file shown by the file pointer. It also increments the file pointer.

Syntax:

```
fputc(c, fp);
```

where `fp` is the file pointer and `C` is a variable written to the file pointed by file pointer.

➤ 14.12 Write a program to write text to a file using `fputc()` function.

```

# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    char c;
    clrscr();
    fp=fopen("lines.txt","w");
    if(fp==NULL)
        return;
    else
    {
        while( (c=getche())!='*')
            fputc(c,fp);
    }
}

```

```
    fclose(fp);
}
```

OUTPUT:

```
India is my country*
```

Explanation:

In the above program, the text entered by the user is written into the file "lines.txt" using the `fputc()` function.

7. fgets() :

This function reads a string from a file pointed by file pointer. It also copies the string to a memory location referenced by an array.

➤ 14.13 Write a program to read text from the given file using `fgets()` function.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    char file[20],text[50];
    int i=0;
    printf("Enter File Name :");
    scanf("%s",file);
    fp=fopen(file,"r");
    if(fp==NULL)
    {
        printf("File not found\n");
        return;
    }
    else
    {
        if(fgets(text,50,fp)!=NULL)
        while(text[i]!='\0')
        {
            putchar(text[i]);
            i++;
        }
    }
}
```

OUTPUT:

```
Enter File Name : IO.C
# include <stdio.h>
```

Explanation:

In the above program, the function `fgets()` reads a string from the file pointed by file pointer. In this example, the `IO.C` file is opened and its first line is displayed on the screen.

8. `fputs()` :

This function is useful when we want to write a string into the opened file.

➤ 14.14 Make a program to write a string into a file using `fputc()` function.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    char file[12],text[50];
    clrscr();
    printf("\n Enter the name of file :");
    scanf("%s", file);
    fp=fopen(file,"w");
    if(fp==NULL)
    {
        printf("\nFile can not opened\n");
        return;
    }
    else
    {
        printf("\n Enter Text Here : ");
        scanf("%s",text);
        fputs(text,fp);
    }
    fclose(fp);
}
```

OUTPUT:

```
Enter the name of file : data.dat
Enter Text Here : Good Morning
```

Explanation:

In the above example, "`data.dat`" file is opened in write mode. The text entered by the user is written in the file using the `fputs()` function. The user can read the contents of file from DOS prompt or one can read the contents of the file using the `fgets()` function as explained in the previous example.

9. `putw()` :

This function is used to write an integer value to file pointed by file pointer. This function deals with integer data only.

➤ 14.15 Write a program to enter integers and write them in the file using the `putw()` function.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    int v;
    clrscr();
    fp=fopen("num.txt", "w");
    if(fp==NULL)
    {
        printf("\n File dose not exist");
        exit(1);
    }
    else
    {
        printf("\n Enter Numbers :");
        while(1)
        {
            scanf("%d",&v);
            if(v==0)
            {
                fclose(fp);
                exit(1);
            }
            putw(v, fp);
        }
    }
}
```

OUTPUT:

Enter Numbers : 1 2 3 4 5 0

Explanation:

In the above program, the file "num.txt" is opened in the write mode. Integers are then entered and written in the file using the `putw()` function. When the '0' is entered, writing of data is stopped and file is closed.

10. `getw ( )` :

This function returns the integer value from a file and increments the file pointer. This function deals with integer data only.

➤ 14.16 Write a program to read integers from the file using `getw()` function.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    int v;
    clrscr();
    fp=fopen("num.txt","r");
    if(fp==NULL)
    {
        printf("\n Entered  numbers :");
        exit(1);
    }
    else
    {
        printf("\n Entered numbers :");
        while((v=getw(fp))!=EOF)
            printf("%2d",v);
        fclose(fp);
    }
}
```

OUTPUT:

Entered numbers: 1 2 3 4 5

Explanation:

In the above program, the same file 'num.txt' used in the previous program is opened for reading the integers. Here, the function `getw()` reads integers from the file.

14.6 STRUCTURES READ AND WRITE

It is important to know how numerical data is stored on the disk by `fprintf()` function. Text and characters require one byte for storing them with `fprintf()`. Similarly for storing numbers in memory two bytes and for floats four bytes will be required.

All data types are treated as characters, for example the data 3456; it occupies two bytes in memory. But when it is transferred to the disk file using `fprintf()` function it would occupy four bytes. For each character one byte would be required. Even for `float` also each digit including dot (.) requires one byte. For example 12.34 would require five bytes. Thus, large amount of integers or `float` data requires large space on the disk. Hence, in the text mode storing of numerical data on the disk turns out to be inefficient. To overcome this problem the files should be read and written in binary mode, for which we use functions `fread()` and `fwrite()`.

1. `fwrite()` & `fread()`

- (1) `fwrite()`: This function is used for writing an entire structure block to a given file.
- (2) `fread()`: This function is used for reading an entire block from a given file.

- 14.17 Write a program to write a block of structure elements to the given file using `fwrite()` function.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    struct
    {
        char name[20];
        int age;
    }
    stud[5];
    FILE *fp;
    int i,j=0,n;
    char str[15];
    printf("Enter the file name :");
    scanf("%s",str);
    fp=fopen(str,"rb");
    if(fp==NULL)
    {
        printf("File dose not exist \n");
        exit(1);
    }
    else
    {
        printf("How Many Records :");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("Name :");
            scanf("%s",&stud[i].name);
            printf("Age :");
            scanf("%d",&stud[i].age);
        }
        while(j<n)
        {
            fwrite(&stud,sizeof(stud),1,fp);
            j++;
        }
    }
    fclose(fp);
}
```

OUTPUT:

```
How Many Records : 2
Name                : SANTOSH
Age                  : 22
Name                  : AMIT
Age                   : 14
```

Explanation:

In the above program, a file is opened in write mode. After successfully opening the file the number of records to be entered is asked. Using the `for` loop records are entered. Using the `while` loop and `fwrite()` statement within it records are written in the file.

- 14.18 Write a program to write and read the information about the player containing player's name, age and runs. Use `fread()` and `fwrite()` functions.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
struct record
{
    char player[20];
    int age;
    int runs;
};
void main()
{
    FILE *fp;
    struct record emp;
    fp=fopen("record.dat","w");
    if(fp==NULL)
    {
        printf("\n Cannot open the file");
        exit(1);
    }
    clrscr();
    printf("\n Enter Player Name, Age & Runs Scored \n");
    printf(" ==== ===== === = ===== \n");
    scanf("%s %d %d",emp.player, &emp.age,&emp.runs);
    fwrite(&emp,sizeof(emp),1,fp);
    fclose(fp);
    if((fp=fopen("record.dat", "r"))==NULL)
    {
        printf("\n Error in opening file");
        exit(1);
    }
    printf("\n Record Entered is \n");
    fread(&emp,sizeof(emp),1,fp);
    printf("\n%s %d %d", emp.player, emp.age, emp.runs);
    fclose(fp);
}
```

OUTPUT:

```
Enter Player Name, Age & Runs Scored
==== ===== === = =====
Sachin 25 10000
Record Entered is
Sachin 25 10000
```

Explanation:

In the above program, a user writes the information of the player using the `fwrite()` function. The entire record of players which is containing information given in the structure is written first using the `fwrite()` function.

Similarly, the information written in the file can be read by using the `fread()` function. Thus, the program reads entire data file with single `fread()` and writes the data with single `fwrite()` function. These two functions are efficiently used for handling I/O files in comparison to `fscanf()` and `fprintf()`.

- 14.19 Write a program to write a block of structure elements to the given file using `fwrite()` function. User should press 'Y' to continue and 'N' for termination.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char next='Y';
    struct bike
    {
        char name[40];
        int avg;
        float cost;
    };
    struct bike e;
    fp=fopen("bk.txt","wb");
    if(fp==NULL)
    {
        puts("Cannot open file");
        exit(1);
    }
    clrscr();
    while (next=='Y')
    {
        printf("\nModel Name, Average, Prize : ");
        scanf("%s %d %f",&e.name,&e.avg,&e.cost);
        fwrite(&e,sizeof(e),1,fp);
        printf("\nAdd Another (Y/N :)");
        fflush(stdin);
        next=getche();
    }
    fclose(fp);
}
```

OUTPUT:

```
Model Name, Average, Prize : HONDA 80 45000
Add Another (Y/N : Y
Model Name, Average, Prize : SUZUKI 65 43000
Add Another (Y/N : Y
```



```
Model Name, Average, Prize : YAMAHA 55 48000
Add Another (Y/N : N
```

Explanation:

The above program is the same as the previous one. Here, the user has to press 'Y' to continue and 'N' to stop the program. After every key press of 'Y' a new record is added to the file.

- 14.20 Write a program to read the information about the bike like name, average and cost from the file using `fread()` function.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    struct bike
    {
        char name[40];
        int avg;
        float cost;
    };
    struct bike e;
    fp=fopen("bk.txt","rb");
    if(fp==NULL)
    {
        puts("Cannot open file");
        exit(1);
    }
    clrscr();
    while(fread(&e,sizeof(e),1,fp)==1)
        printf("\n %s %d %.2f",e.name,e.avg,e.cost);
    fclose(fp);
}
```

OUTPUT:

```
Model Name, Average, Prize : HONDA 80 45000.00
Model Name, Average, Prize : SUZUKI 65 43000.00
Model Name, Average, Prize : YAMAHA 55 48000.00
```

Explanation:

In the above program, the records are written in the binary mode on the disk in the file "bk.txt". Record writing will be over after the user presses 'N'.

The same file is opened in read mode. The `fread()` function reads the records from the disk which is to be placed in the `printf()` statement to display on the screen. After the file detection of end of the file `fread()` function no more reads anything. It returns a '0'.

Functions `fread()` and `fwrite()` store the numbers more efficiently and the reading and writing of structures are easy.

14.7 OTHER FILE FUNCTION

- (a) The `fseek()` function: The `fseek()` can be used to access the part of the file. The file pointer can be moved to any position in a file. It positions file pointer on the stream.

The format of `fseek()` is as follows:

`fseek(filepointer, offset, position)`

Thus, three arguments are to be passed through this function. They are

- (1) file pointer.
- (2) Offset: offset may be positive (moving in forward from current position) or negative (moving backwards). The offset being a variable of type long.
- (3) The current position of file pointer.

Table 14.2 displays the various values of location of file pointer.

Table 14.2 *Locations of file pointer*

<i>Integer Value</i>	<i>Constant</i>	<i>Location in the File</i>
0	SEEK_SET	Beginning of the file.
1	SEEK_CUR	Current position of the file pointer.
2	SEEK_END	End of the file

Example:

`fseek(fp, 10, 0)` or `fseek(fp, 10, SEEK_SET)`

The file pointer is repositioned in the forward direction by 10 bytes.

- 14.21 Write a program to read the text file containing some sentence. Use `fseek()` and read the text after skipping *n* characters from beginning of the file.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    int n, ch;
    clrscr();
    fp=fopen("text.txt", "r");
    printf("\nContents of file\n");
    while((ch=fgetc(fp))!=EOF)
        printf("%c", ch);
    printf("\nHow many characters including spaces would you like
    to skip ? :");
    scanf("%d", &n);
    fseek(fp, n, SEEK_SET);
    printf("\n Information after %d bytes\n", n);
}
```

```

while((ch=fgetc(fp))!=EOF)
printf("%c",ch);
fclose(fp);
}

```

OUTPUT:

Contents of file:

THE C PROGRAMMING LANGUAGE INVENTED BY DENNIS RITICHE

How many characters including spaces would you like to skip ? : 18

Information after 15 bytes

LANGUAGE INVENTED BY DENNIS RITICHE

Explanation:

In the above program, `while` statement is used for checking the end of the file. The file pointer is initially positioned at the beginning of the file and the whole text is printed.

To reposition the file pointer the statement `fseek()` is used. The file pointer is to be positioned at *n*th character from the beginning of the file and the characters from *n*th position onwards will be printed on the screen.

In the above program, one can use the statement `fseek(fp, -n, SEEK_END)` in place of `fseek(fp, n, SEEK_SET)` to read from backward direction from the end of the file.

- 14.22 Write a program to read the last few characters of the file using the `fseek()` statement.

```

# include <dos.h>
# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    int n,ch;
    clrscr();
    fp=fopen("text.txt","r");
    printf("\nContents of file\n");
    while((ch=fgetc(fp))!=EOF)
    printf("%c",ch);
    printf("\n How many characters including spaces would you like
to skip ? :");
    scanf("%d",&n);
    fseek(fp, -n-2, SEEK_END);
    printf("\nLast %d characters of a file\n", -n-2);
    while((ch=fgetc(fp))!=EOF)
    printf("%c",ch);
    fclose(fp);
}

```

OUTPUT:

How many characters including spaces would you like to skip? : 4

Last 5 characters of a file

WORLD

Explanation:

The statement `fseek(fp, -n-2, SEEK_END);` repositions the file pointer `-n-2` bytes in the backward directions from the end of file. With this statement last characters of a statement can be printed on the screen, i.e. printing information will be from the position till the end of file. Here, the value of 'n' entered is 4. The value of `-n-2` in this example works out to be `-6`. So the last six characters are displayed. The result shown here is 'WORLD' which contains five characters and the last is `NULL`. `NULL` is skipped.

- 14.23 Write a program to display C program files in current directory. The user should select one of the files. Convert the file contents in capital and display the same on the screen. Also calculate total characters and lines.

```
# include <ctype.h>
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    int l=0,c=0,ch;
    static char file [12];
    clrscr();
    system("dir *.c/w");
    printf("\n Enter a file name :");
    scanf("%s",file);
    fp=fopen(file,"r");
    clrscr();
    printf("\nContents of 'c' program File in capital case \n");
    printf("=====\n");
    while((ch=fgetc(fp))!=EOF)
    {
        c++;
        if(ch=='\n')
            l++;
        printf("%c",toupper(ch));
    }
    printf("\n Total Characters : %d",c);
    printf("\n Total Lines : %d",l);
}
```

OUTPUT:

```
Enter a file name : IO10.C
Contents of 'c' program File in capital case
main()
{
    printf("HELLO WORLD");
}
Total Characters : 31
Total Lines : 4
```

Explanation:

In the above program, `system()` statement is used for invoking MS-DOS command. The `dir *.c/w` displays the contents of the current directory. The function `fgetc()` reads the text of the entered file character by character till the end of the file. In the `while` loop, total characters and lines are counted. The line termination is to be identified by `'\n'`. Hence, the `if` statement is used for the identification of termination of line. In the same way, `'\t'` can be used to take the account of tabs. The text is also converted to capital by using `toupper()` function and it is displayed.

- (b) `feof()`: The macro `feof()` is used for detecting if the file pointer is at the end of file or not. It returns non-zero if the file pointer is at the end of file, otherwise it returns zero.

➤ 14.24 Write a program to detect the end of file using the function `feof()`. Display the file pointer position for detecting end of file.

```
# include <conio.h>
# include <stdio.h>
void main()
{
    FILE *fp;
    char c;
    fp=fopen("text.txt", "r");
    c=feof(fp);
    clrscr();
    printf("File pointer at the beginning of the file : %d\n",c);
    while(!feof(fp))
    {
        printf("%c",c);
        c=getc(fp);
    }
    c=feof(fp);
    printf("File pointer at the end of file : %d ",c);
}
File pointer at the beginning of the file : 0
TECHNOCRATS LEAD THE WORLD
File pointer at the end of file : 32
```

Explanation:

At starting, the `feof()` function returns 0 (zero) because the file pointer is at the beginning of file. After reading the contents of file the file pointer reaches at the end of file. At this stage, the `feof()` function returns non-zero value which is 32 in this program.

14.8 SEARCHING ERRORS IN READING/WRITING FILES

While performing read or write operation a few times, we do not get results successfully. The reason may be that the attempt of reading or writing operation may not be correct. The provision must be provided for searching the error while read/write operations are carried out.

The C language provides standard library function `ferror()`. This function is used for detecting any error that might occur during read/write operation of a file. It returns a '0' while the attempt is successful otherwise non-zero in case of failure.

- (c) `ferror()`: The `ferror()` is used to find out error when file read write operation is carried out.

- 14.25 Write a program to detect an error with read/write operation of a file.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char next='Y';
    char name[25];
    int marks;
    float p;
    fp=fopen("marks.dat","r");
    if(fp==NULL)
    {
        puts("Can not open file");
        exit(1);
    }
    clrscr();
    while(next=='Y')
    {
        printf("\n Enter Name, Marks, Percentage");
        scanf("%s %d %f",name,&marks,&p);
        p=marks/7;
        fprintf(fp,"%s %d %f",name,&marks,&p);
        if(ferror(fp))
        {
            printf("\n Unable to write data ? ");
            printf("\n File opening mode is incorrect.");
            fclose(fp);
            exit(1);
        }
        printf("Continue(Y/N)");
        fflush(stdin);
        next=getche();
    }
    fclose(fp);
}
```

OUTPUT:

```
Enter Name, Marks
KAMAL 540
Unable to write data?
File opening mode is incorrect.
```

Explanation:

In the above program, the `fprintf()` function fails to write the entered data to the file because it is opened in the read mode. Hence, write operation cannot be done. The error generated is caught by the `ferror()` function and the program terminates.

- 14.26 Write a program to catch the error that occurs while read operation of a file using `ferror()` function.

```
# include <stdio.h>
# include <process.h>
# include <conio.h>
void main()
{
    FILE *f;
    char c;
    clrscr();
    f=fopen("io8.c", "w");
    if(f==NULL)
    {
        printf("\nCannot open file");
        exit(1);
    }
    while((c=fgetc(f))!=EOF)
    {
        if(ferror(f))
        {
            printf("\nCan't read file.");
            fclose(f);
            exit(1);
        }
        printf("%c", c);
        getch();
    }
    fclose(f);
}
```

OUTPUT:

Can't read file.

Explanation:

In the above program, the file 'io8.c' is opened in write mode. Whereas the function `fgetc()` would certainly fail because it used for reading operation the `fgetc()` is trying to read the file which is impossible. This never happens because the file that is opened in write mode cannot be read and vice versa.

- (d) `perror()`: It is a standard library function which prints the error messages specified by the compiler. A program is illustrated below for understanding.

➤ 14.27 Write a program to detect and print the error message using `perror()` function.

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fr;
    char c, file[]="lines.txt";
    fr=fopen(file, "w");
    clrscr();
```

```

while(!feof(fr))
{
    c=fgetc(fr);
    if(ferror(fr))
    {
        perror(file);
        exit(1);
    }
    else
        printf("%c",c);
}
fclose(fr);
}

```

OUTPUT:

lines.txt : Permission Denied

Explanation:

In the above program to print the error message a user can use `perror()` function instead of `printf()`. The output of the above program is 'lines.txt: Permission Denied'. We can also specify our own message together with the system error message. In the above example, 'file' variable prints the file name 'lines.txt' together with compiler's message 'Permission Denied'.

- (e) `ftell()`: It is a file function. It returns the current position of the file pointer. It returns the pointer from the beginning of file. The current position of the file is detected with this function.

- 14.28 Write a program to print the current position of the file pointer in the file using the `ftell()` function.

```

# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    char ch;
    fp=fopen("text.txt","r");
    fseek(fp,21,SEEK_SET);
    ch=fgetc(fp);
    clrscr();
    while(!feof(fp))
    {
        printf("%c\t",ch);
        printf("%d\n",ftell(fp));
        ch=fgetc(fp);
    }
    fclose(fp);
}

```


OUTPUT:

```

W 22
O 23
R 24
L 25
D 26
  28

```

Explanation:

In the above program, `fseek()` function sets the cursor position on byte 21. The `fgetc()` function in the `while` loop reads the character after 21st character. The `ftell()` function prints the current pointer position in the file. When `feof()` function is found at the end of file, program terminates.

(f) `rewind()` : This function resets the file pointer at the beginning of the file.

➤ 14.29 Write a program to show how `rewind()` function works.

```

# include <stdio.h>
# include <conio.h>
void main()
{
    FILE *fp;
    char c;
    fp=fopen("text.txt","r");
    clrscr();
    fseek(fp,12,SEEK_SET);
    printf("Pointer is at %d\n", ftell(fp));
    printf("Before rewind() : ");
    while(!feof(fp))
    {
        c=fgetc(fp);
        printf("%c",c);
    }
    printf("\bAfter rewind() : ");
    rewind(fp);
    while(!feof(fp))
    {
        c=fgetc(fp);
        printf("%c",c);
    }
}

```

OUTPUT:

```

Pointer is at 12
Before rewind() : LEAD THE WORLD
After rewind() : TECHNOCRATS LEAD THE WORLD

```

Explanation:

In the above program, the `fseek()` function sets the pointer position on the 12th character. The first `while` loop reads and prints the file from the current position up to the end in which first 12 characters of file are not displayed. Before starting of the second `while` loop the `rewind()` function sets the pointer position at the beginning of file, i.e. on 1 character. The `while` loop reads all the characters of the file from starting to end. In short the `rewind()` function sets the file pointer at the beginning of the file.

- (g) `unlink()` or `remove()`: These functions delete the given file in the directory. It is similar to the `del` command in DOS.

➤ 14.30 Write a program to delete the given file from the disk using `remove()` or `unlink()` function.

```
# include <dos.h>
# include <stdio.h>
# include <conio.h>
# include <process.h>
void main()
{
    FILE *fp;
    char file[15];
    clrscr();
    system("dir *.txt");
    printf(" Enter The File Name :");
    scanf("%s",file);
    fp=fopen(file,"r");
    if(fp==NULL)
    {
        printf("\nFile dose not exist");
        exit(1);
    }
    else
    {
        remove(file);
        printf("\n File (%s) has been deleted !",file);
    }
}
```

OUTPUT:

```
Enter The File Name : TEXT.TXT
File (TEXT.TXT) has been deleted !
```

Explanation:

In the above program, file to be deleted is entered. The entered file is opened in read mode. If the `fopen()` fails to open the file then it returns `NULL` to pointer `*fp`, i.e. file does not exist and program terminates. By using `unlink()` or `remove()` function the entered file is deleted.

- (h) `rename()`: This function changes name of the given file. It is similar to the DOS command `rename`.

- 14.31 Write a program to change the name of the file. Use `rename()` function.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    char old[12], new[12];
    system("dir *.c /w");
    printf("\nOld File Name : ");
    scanf("%s",old);
    printf("New File Name : ");
    scanf("%s",new);
    rename(old,new);
    system("dir *.c/w");
}
```

OUTPUT:

```
Old File Name : old.c
New File Name : new.c
```

Explanation:

In the above program, the `rename()` function changes the name of the given file with a new name. Its meaning is similar to DOS command 'rename'.

- 14.32 Write a program to copy the contents of one file to another file.

```
/* File copy program */
# include <stdio.h>
# include "process.h"
void main()
{
    /* Declaration */
    FILE *ft,*fs;
    int c=0;
    clrscr();
    /* Opening Files */
    fs=fopen("a.txt","r");
    ft=fopen("b.txt","w");
    if(fs==NULL )
    {
        printf("\n Source file opening error.");
        exit(1);
    }
    else
    if(ft==NULL)
    {
        printf("\n Target file opening error.");
        exit(1);
    }
    /* Reading file */
```

```

while(!feof(fs))
{
    fputc(fgetc(fs),ft);
    c++;
}
printf("\n %d Bytes copied from 'a.txt' to 'b.txt.'",c);
/* Closing Files */
c=fcloseall();
printf("\n %d files closed.",c);
}

```

OUTPUT:

```

45 Bytes copied from 'a.txt' to 'b.txt.
2 files closed.

```

Explanation:

In the above program, two files are opened. The source file 'a.txt' is opened in read mode and the target file 'b.txt' is opened in the write mode. With the `fputc()` function the contents read by `fgetc()` is written to the target file. To count the number of characters of source file 'c' counter is used. The `fcloseall()` function closes all the opened files and returns the number of files closed. This return value of `fcloseall()` is collected by the variable 'c'.

➤ 14.33 Read the contents of three files and find the largest file.

```

# include <stdio.h>
# include <conio.h>
# include <alloc.h>
# include <process.h>
void main()
{
    FILE *f[3], *fp;
    int x[3], l, y=0, k=0, t=0;
    char c1;
    char name[3][11]={"1.txt", "2.txt", "3.txt"};
    clrscr();
    for(l=0; l<3; l++)
    {
        fp=fopen(name[l], "r");
        f[l]=fp;
        if(fp==NULL)
        {
            printf("\n %s file not found. ", name[l]);
            exit(1);
        }
    }
    clrscr();
    for(l=0; l<3; l++)
    {
        while(!feof(f[l]))
        {
            c1=fgetc(f[l]);

```

```

        x[l]=y++;
    }
    y=0;
}
clrscr();
fcloseall();
for(l=0;l<=2;l++)
{
    printf("File : %s Bytes : %d\n",name[l],x[l]);
    t=t+x[l];
}
for(l=t;l>=1;l--)
{
    for(k=0;k<3;k++)
    {
        if(l==x[k])
        {
            printf("%s are the largest file.",name[k]);
            exit(1);
        }
    }
}
}

```

OUTPUT:

```

File : 1.txt Bytes : 16
File : 2.txt Bytes : 20
File : 3.txt Bytes : 25
3.txt are the largest file.

```

Explanation:

In the above program, an array of file pointer and a separate file pointer `fp` are declared. The two-dimensional character array is used for storing file names. Using file pointer array in the `for` loop, three files are opened in read mode. Each element of file pointer array points to the corresponding elements of character array i.e. file names.

The `fgetc()` function in the `while` loop reads contents of each file one by one. The numbers of characters are also written to integer array in the same loop. At the end, the values stored in integer array are compared with one another and the file containing the largest number of bytes is detected. The name of the largest file is displayed.

- 14.34 Write a program to copy up to 100 characters from a file to an array. Then copy the contents of an array to another file.

```

# include <stdio.h>
# include <conio.h>
# include <process.h>
# define SIZE 100
void main()
{
    FILE *f,*f2;
    static char c,ch[SIZE];

```

```

int s=0,x=0;
clrscr();
f=fopen("poem.txt","r");
f2=fopen("alpha.txt", "w");
if(f==NULL || f2==NULL )
{
    perror("?");
    exit(1);
}
clrscr();
while(!feof(f))
{
    c=fgetc(f);
    x++;
    if(x==99)
    {
        fcloseall();
        exit(1);
    }
    else
        ch[s++]=c;
}
fclose(f);
for(s=0;s<=100;s++)
    fputc(ch[s],f2);
fclose(f2);
perror("Process Completed");
}

```

OUTPUT:

Process Completed : Error 0

Explanation:

In the above program, the contents of file 'poem.txt' are read using and copied to a character array ch[]. The same array is read and written to the file 'alpha.txt' using the fputc() function.

- 14.35 Write a program to copy the contents of one file to another three files. Display the contents of the three files.

```

# include <stdio.h>
# include <conio.h>
# include "process.h"
void main()
{
    FILE *ft,*fs,*fa,*fb;
    int c=0;
    char k;
    clrscr();
    fs=fopen("intl.txt","r");
    ft=fopen ("a.txt","w");
    fa=fopen("b.txt","w");

```

```

fb=fopen("c.txt", "w");
if(fs==NULL )
{
    printf("\n Source file opening error.");
    exit(1);
}
else
if(ft==NULL || fa==NULL || fb==NULL )
{
    printf("\n Target file opening error.");
    exit(1);
}
while(!feof(fs))
{
    k=fgetc(fs);
    if(c<100)
        fputc(k, ft);
    if(c>=100 && c<200)
        fputc(k, fa);
    if(c>200 && c<=282)
        fputc(k, fb);
    c++;
    if(c==283) break;
}
fcloseall();
ft=fopen("a.txt", "r");
fa=fopen("b.txt", "r");
fb=fopen("c.txt", "r");
if(ft==NULL || fa==NULL || fb==NULL )
{
    printf("\n Target file opening error.");
    exit(1);
}
while(!feof(ft))
{
    k=fgetc(ft);
    printf("%c", k);
}
while(!feof(fa))
{
    k=fgetc(fa);
    printf("%c", k);
}
while(!feof(fb))
{
    k=fgetc(fb);
    printf("%c", k);
}
fcloseall();
}

```

OUTPUT:

```
-----
                Microsoft Windows 98 Seco_nd Edition
                README for Pan-European Regional Settings
                (c) Copyright Micro_ofst Corporation, 1999
                -----
```

Explanation:

In this program, the text file `intl.txt` is opened in the read mode. Another three files `a.txt`, `b.txt` and `c.txt` are opened in the write mode. Total 300 characters are read from the file and 100 characters are written into each file. Later contents of these `a.txt`, `b.txt` and `c.txt` are displayed on the screen.

14.9 LOW-LEVEL DISK I/O

Text files can be copied from one file to other or in many files. However if we attempt to open binary files such as `.EXE` in text mode, unexpectedly the process of getting characters would stop. This is because whenever ASCII value 26 is observed copying work stops due to EOF (). Another approach to copy such binary files are with low-level disk I/O.

In the low-level disk I/O disk operation, data cannot be written as character-by-character or with sequence of characters as it carried in the high-level disk I/O functions. In the low-level disk I/O functions, buffers are used to carry the read and write operations.

Buffer plays an important role in the low-level disk I/O program. The programmer needs to declare the appropriate buffer size. The low-level disk I/O operations are more efficient and quick than the high-level disk I/O operations.

- (a) Opening a file: To open a file or files `open( )` function is used. This function is defined in `'io.h'`. The syntax of `open( )` is as given below.

Syntax:

```
int open(const char *f_name, int access, unsigned mode)
```

In the low-level operation, a number is assigned to the file and the number is used to refer the file. If `open()` returns `-1`, it means that the file could not be opened otherwise the file is successfully opened.

Table 14.3 describes the file-opening modes in the low-level disk I/O operations.

Table 14.3 File-opening modes

Mode	Meaning
O_APPEND	Opens a file in append mode.
O_WRONLY	Creates a file for writing only.
O_RDONLY	Opens a file for reading only.
O_RDWR	Opens a file for read/write operations.
O_BINARY	Opens a file in binary mode.
O_CREATE	Opens a new file for writing.
O_EXCL	When used with O_CREATE, if a file exists it is not overwritten.
O_TEXT	Creates a text file.

When `O_CREATE` flag is used, it also requires one of arguments described in Table 14.4 to verify the read/write status of the file. These arguments are called permission argument. The programmer need to include the header file '`stat.h`' and '`types.h`' along with '`fcntl.h`'.

Table 14.4 *Permission argument*

<code>S_IWRITE</code>	Writing to the file allowed
<code>S_IREAD</code>	Reading from the file allowed

- (b) Writing a file: The `write()` function is used to `write()` data into the file. This function is defined in '`io.h`'. The syntax of `write()` function is as given below.

Syntax:

```
int write(int handle, void *buf, unsigned nbyte);
```

Returns the number of bytes written or `-1` if an error occurs.

- (c) Reading a file: The `read()` function reads a file. The syntax of `read()` function is as follows.

Syntax:

```
int read(int handle, void *buf, unsigned len);
```

Upon successful end, it returns an integer specifying the number of bytes placed in the buffer; if the file was opened in text mode, `read` does not count carriage returns or `ctrl-Z` characters in the number of bytes read. On error, it returns `-1` and sets `errno`.

- (d) Closing a file: The `close()` function closes the file. This function is defined in '`io.h`'. The syntax of `close()` is as per given below.

Syntax:

```
int _close(int handle);
```

```
int close(int handle);
```

Upon successful finish, `close` & `_close` returns `0`; otherwise, they return `-1` and set `errno`.

- 14.36 Write a program to enter text through keyboard and store it on the disk. Use low-level disk I/O operations.

```
# include <stdio.h>
# include <conio.h>
# include <io.h>
# include <fcntl.h>
void main()
{
    char file[10];
    char buff[15];
    int s,c;
    puts("\n Enter a file name :");
    gets(file);
    s=open(file,O_CREAT | O_TEXT);
    if(s==-1)
        puts("\n File does not exists");
    else
    {
        puts( "Enter text below : ");
```

```

        for(c=0;c<=13;c++)
        buff[c]=getche();
        buff[c]='\0';
        write(s,buff,15);
        close(s);
    }
}

```

OUTPUT:

```

Enter a file name :
TEXT.txt
Enter text below : PROGRAMMING IN C

```

Explanation:

In the above program, a prompt appears for asking a file name to be created. A file is created with the name entered by the user. If the file is already present, it would not be overwritten and the entered text would not be written in the file. If `open()` fails to open the file then it will return `-1` and this value is assigned to variable `s`. The `if` statement checks the value of `s` and respective blocks are executed. In the `else` block, the `getche()` within the `for` loop reads 14 characters through the keyboard and after the `for` loop the string is terminated by the NULL character. Here, instead of using `gets()`, `getche()` is used. In `gets()`, there may be a possibility that the entered text may be of less than 15 characters. In such a case, some garbage values are also written in the file. To avoid the garbage we used `getche()` within the `for` loop to exactly read character equal to that mentioned in the `write()` statement, i.e. 15.

- 14.37 Write a program to read text from a specified file from the disk. Use low-level disk I/O operations.

```

# include <stdio.h>
# include <conio.h>
# include <io.h>
# include <fcntl.h>
# include <process.h>
void main()
{
    char file[10],ch;
    int s;
    clrscr();
    puts("\n Enter a file name :");
    gets(file);
    s=open(file,O_RDONLY);
    if (s==-1)
    {
        puts("\n File does not exists.");
        exit(1);
    }
    else

```

```

{
    while(!eof(s))
    {
        read(s,&ch,1);
        putchar(ch);
    }
    close(s);
}
}

```

OUTPUT:

```

Enter a file name : TEXT.txt
PROGRAMMING IN C

```

Explanation:

In the above program, the file is opened in read-only mode. The `read()` declaration within the `while` loop reads single character from the file destined by the file handler `s`. The `putchar()` statement following the `read()` declaration shows the read character on the console. The `while` loop ends when the end of file is detected.

- (e) Setting Buffer: The size of buffer can be set using the `setbuf()` function. This function is defined in '`stdio.h`'. The syntax of `setbuf()` is as follows.

Syntax:

```
void setbuf( FILE *fp, char *buffer);
```

- 14.38 Write a program to set a buffer size using `setbuf()` function.

```

# include <stdio.h>
# include <conio.h>
void main()
{
    char buff[22];
    clrscr();
    setbuf(stdout,buff);
    printf ("\nThis book teaches C");
    fflush(stdout);
}

```

OUTPUT:

```
This book teaches C
```

Explanation:

In the above program, a character array `buff[22]` is declared. The `setbuf()` function sets the buffer size as per the size of `buff[22]` array. The `printf()` statement displays the message written in it. If the characters written in the `printf()` statement are more than the buffer size i.e. 22 the program will be terminated with a critical error. Hence, the text that is to be displayed using any output function should be less or equal to the size of buffer.

14.10 COMMAND LINE ARGUMENTS

An executable program that performs a specific task for operating system is called a command. The commands are issued from the prompt of operating system. Some arguments are to be associated with the commands hence these arguments are called command line arguments. These associated arguments are passed to the program.

In C language, every program starts with a `main()` function and it marks the beginning of the program. We have not provided any arguments so far in the `main()` function. Here, we can make arguments in the main like other functions. The `main()` function can receive two arguments and they are: (1) `argc` and (2) `argv`. The information contained in the command line is passed on to the program through these arguments when the `main()` is called up by the system.

- (1) **Argument `argc`:** An argument `argc` counts the total number of arguments passed from command prompt. It returns a value which is equal to the total number of arguments passed through the `main()`.
- (2) **Argument `argv`:** It is a pointer to an array of character strings which contain names of arguments. Each word is an argument.

Example:

Copy file1 file2

Here, file1 and file2 are arguments and copy is a command. The first argument is always an executable program followed by associated arguments. If you do not specify the argument, the first program name itself is an argument but the program will not run properly and will flag an error.

A program on above concept is explained below.

- 14.39 Write a program to display the number of arguments and their names.

```
# include <stdio.h>
# include <conio.h>
main(int argc, char *argv[])
{
    int x;
    clrscr();
    printf("\n Total number of arguments are %d \n",argc);
    for(x=0;x<argc;x++)
        printf("%s\t",argv[x]);
    getch();
    return 0;
}
```

OUTPUT:

```
Total number of arguments are 4
C:\TC\C.EXE A B C
```

Explanation:

To execute this program, one should create its executable file and run it from the command prompt with required arguments. The above program is executed using the following steps:

- (i) Compile the program.
- (ii) Make its exe file (executable file).
- (iii) Switch to the command prompt. (C:\TC>)

- (iv) Make sure that the exe file is available in the current directory.
- (v) Type the following bold line.

C:\TC> C.EXE HELP ME

In the above example, c.exe is an executable file and 'HELP ME' are taken as arguments. The total numbers of arguments including program file name are three.

14.11 APPLICATION OF COMMAND LINE ARGUMENTS

Below given programs can be used on command prompt similar to that of DOS commands.

(1) TYPE:

- 14.40 Write a program to read any file from command prompt. Use command line arguments. (Save this program as read.c)

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
main( int argc, char *argv[])
{
    FILE *fp;
    int ch;
    fp=fopen(argv[1],"r");
    if(fp==NULL)
    {
        printf("Can not open file");
        exit(0);
    }
    while(!feof(fp))
    {
        ch=fgetc(fp);
        printf("%c",ch);
    }
    fclose(fp);
    return 0;
}
```

Explanation:

The above program after getting its exe file will run similarly as type command of DOS. On the command prompt, one should write first exe file name and file name to be read. The contents of file will be displayed on the screen.

(2) DEL:

- 14.41 Write a program using command line argument to perform the task of DEL command of DOS. (Save this program as cut.c)

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

```

main(int argc, char *argv[])
{
    FILE *fp;
    if(argc<2)
    {
        printf("Insufficient Arguments");
        exit(1);
    }
    fp=fopen(argv[1],"r");
    if(fp==NULL)
    {
        printf("File Not Found");
        exit(1);
    }
    unlink(argv[1]);
    printf("File has been deleted ");
    return 0;
}

```

Explanation:

This program performs the task of DEL command of disk operating system. It deletes only one file at a time. It also creates exe file of this program and executes it on the command prompt. User should give the file name for deleting. Opening it in read mode checks the existence of file. The file will be deleted in case it exists. In the program, appropriate messages are displayed if user makes any mistake. Error messages are ‘insufficient arguments’, ‘file not found’ and “file has been deleted”.

(3) RENAME:

- 14.42 Write a program using command line argument to perform the task of REN command of DOS. (Save this program as **change.c**)

```

# include <stdio.h>
# include <conio.h>
# include <process.h>
main(int argc, char *argv[])
{
    FILE *fp,*sp;
    if(argc<3)
    {
        printf("Insufficient Arguments");
        exit(1);
    }
    fp=fopen(argv[1],"r");
    if(fp==NULL)

```

```

{
    printf("File Not Found");
    exit(1);
}
sp=fopen(argv[2],"r");
if(sp==NULL)
{
    fcloseall();
    rename(argv[1],argv[2]);
}
else
printf("Duplicate file name or file is in use." );
return0;
}

```

Explanation:

In the above program, the `main()` receives two file names. The existence of old file is checked through read mode. If the file does not exist, program terminates. On the other hand, if the second file exists the rename operation can not be done. If the file pointer of the second file contains the NULL value, then only rename operation is performed otherwise program terminates with an appropriate error message.

14.12 ENVIRONMENT VARIABLES

Environment variables provide different system settings/path related to operating system. These variables are available in both MS-DOS and UNIX operating system. The output depends upon the operating system. The following program displays the output related to MS-DOS.

➤ 14.43 Write a program to use the environment variable and display various settings.

```

# include <stdio.h>
# include <conio.h>
void main(int argc, char *argv[],char *env[] )
{
    int i;
    clrscr();
    for(i=0;env[i]!=NULL;++i)
    {
        printf("%s\n",env[i]);
    }
}

```

OUTPUT:

```

TMP=C:\WINDOWS\TEMP
TEMP=C:\WINDOWS\TEMP
PROMPT=$p$g
winbootdir=C:\WINDOWS
COMSPEC=C:\WINDOWS\COMMAND.COM

```

```

PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\JDK1.2.1\BIN;C:\JDK1.2.1;
CMDLINE=WIN
windir=C:\WINDOWS

```

14.13 I/O REDIRECTION

Normally a program receives the input from keyboard, and on processing the input result is provided on the output device such as monitor. Using MSDOS, the feature redirection permits to send the result to disk instead of the monitor. It is also possible to bring information in program from disk instead of the keyboard.

Consider the DOS command `dir > abc`; execute it at the DOS prompt. Here, `dir` is the internal DOS command and `abc` is the name of the file. Here, the list of files and directories, instead of displaying on the screen, is redirected in a text file `abc` i.e. the files and directories would, possibly to be seen in `abc` file. For this use, type `abc` command at DOS prompt.

From the above example, the reader can understand that the output of the command can be redirected to another file. Using redirection the output of the program instead of displaying on the screen is stored on the disk in the form of a file. With this, we can avoid creating a separate function for writing files to the disk or to the printer. Thus, this is an advantage and a convenient approach for writing files to the disk. Using this redirection concept both read and write operations are possible.

The following steps can be adopted to follow the concept of redirection:

- (i) Compile a program
- (ii) Get an executable file `.exe`
- (iii) Execute this `.exe` file on the DOS prompt with redirection

Example: `read.exe > input.txt`

- (iv) Input the data from keyboard
- (v) See the typed data on DOS prompt by typing `Type input.txt`

Executing the following programs can follow the concept of redirection.

- 14.44 Write a program to read a character from the keyboard till the user presses enter.

```

# include <stdio.h>
# include <conio.h>
void main()
{
    char c;
    while((c=getc(stdin))!='\n')
        putc(c, stdout);
}

```

OUTPUT:

```
1 2 3 4 5 6 7 8 9
```

Explanation:

On compiling this program, we would get an executable file `read.exe`. Execute this program on the DOS prompt as given below.

```
C> read.exe > input.txt
```


Now, whatever data is inputted, it is redirected to text file `input.txt`. To confirm, type the file using type command. The output would be

```
1 2 3 4 5 6 7 8 9
```

The redirection operator ‘>’ transfers any output proposed for screen to the file followed by the operator.

It is optional to input data in the program. We can also redirect output of the program generated by itself to the text file. The above program illustrates this point.

➤ 14.45 Write a program to display A to Z characters.

```
# include <stdio.h>
# include <conio.h>
void main()
{
    int a;
    clrscr();
    for(a=65;a<91;a++)
        printf("\t%c\t",a);
    printf("\n");
}
```

OUTPUT:

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y
Z				

Explanation:

After compilation we get the exe file. Execute it at DOS prompt as given below.

```
c> alpha.exe > abc.txt
```

After execution, the output generated by the program is directed to file `abc.txt`. It is also possible to send the output to printer. For this, follow the following syntax:

```
c> alpha.exe > prn
```

We can also redirect the input to the program from a file. Instead of writing with the keyboard, a complete file can be transferred. The below given program explains this point.

```
C> read.exe < abc.txt
```

This command display output is

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y
Z				

Here, the contents of `abc.txt` are redirected to program `read.exe`. The user need not enter any data through the keyboard. The entire contents of `abc.txt` are used as input and the same is displayed. Here, the '`<`' redirection operator is used.

Input and output can be redirected simultaneously. Here, input is taken from a file and transferred to another file. The program acts as mediator between both the files. The following command illustrates this process:

```
C> read.exe < abc.txt > alpha.txt
```

In this process, program reads redirected data from the file `abc.txt`. Also instead of displaying output on the screen, it would be redirected to the file `alpha.txt`. The contents of both the files `abc.txt` and `alpha.txt` will be the same. User can confirm by typing them.

SUMMARY

This chapter explains the procedure for opening files and storing information in them. The various I/O functions related to high level and low-level file disk operations are elaborated with programming examples. After having gone through structures read and write, you are now familiar with `fwrite()` and `fread()` functions. Using other file functions, we described functions related to seeking the particular record, end of file and detecting the errors. Command line arguments to accept arguments from command prompt of the operating system are described. Simulations of various DOS commands with examples are also narrated. Reader is also made familiar with I/O-REDIRECTIONS in which output of the program can be redirected to file or printer. Also, data from file can also be redirected to program.

EXERCISES

I True or false:

1. A file is a set of records.
2. The `FILE` pointer contains all information about a file.
3. Without the file pointer file can be operated in C.
4. In sequential file data can be read directly.
5. In random file the last record can be read first.
6. The `fopen()` is used to open file.
7. The statement `FILE *p` declares file pointer.
8. The '`r`' mode means file is opened for writing only.
9. The '`w`' mode opens a new file on the disk.
10. The function `fputc()` writes data to the file.
11. The `fprintf()` is similar to `printf()`.
12. The function `feof()` finds the end of file.
13. The `getw()` is not associated with any file operation.
14. The `SEEK_CUR` indicates current position of the file pointer.
15. The `ferror()` reports error occurred during file read/write operation.

16. The function `rewind()` reverses the contents of a file.
17. The 'wb' creates a file in binary mode.
18. The `O_APPEND` is a low-level disk operation.
19. The `argc` and `argv` are not command line arguments.

II Match the following correct pairs given in Group A with Group B:

1.

Group A

Group B

Sr. No	File Functions	Sr. No	Used for
1.	<code>fputs()</code>	A	Writes characters one by one to a file.
2.	<code>fputc()</code>	B	Writes strings to the file.
3.	<code>fwrite()</code>	C	Detects the end of file.
4.	<code>feof()</code>	D	Writes the block of structured data to the file.

2.

Group A

Group B

Sr. No	File Functions	Sr. No	Used for
1.	<code>rewind()</code>	A	Returns the current pointer position.
2.	<code>perror()</code>	B	Set the record pointer at the beginning of the file.
3.	<code>ftell()</code>	C	Removes the specified file from the disk.
4.	<code>unlink()</code>	D	Prints compilers error messages along with user defined messages.

III Select the appropriate option from the multiple choices given below:

- The `fscanf()` statements reads data from
 - file
 - keyboard
 - Both (a) and (b)
 - None of the above
- When `fopen()` fails to open a file it returns
 - NULL
 - 1
 - 1
 - None of the above
- A file opened in `w+` mode can be
 - read/write
 - only read
 - only write
 - None of the above
- Command line arguments are used to accept argument from
 - command prompt of operating system
 - through `scanf()` statement
 - Both (a) and (b)
 - None of the above
- The redirection operator '`>`' transfers any output to
 - text file
 - console
 - both (a) and (b)
 - None of the above

6. This function is used to detect the end of file (a) <code>feof()</code> (b) <code>ferror()</code> (c) <code>fputs()</code> (d) <code>fgetch()</code>	7. The EOF is equivalent to (a) <code>-1</code> (b) <code>0</code> (c) <code>1</code> (d) None of the above

IV Attempt the following programming exercises:

1. Write a program to generate a data file containing the list of cricket players, no. of innings played, highest run score and no. of hatricks made by them. Use structure variable to store the cricketer's name, no. of innings played, highest run score and the number of hatricks. 2. Write a program to reposition the file to its 10th character. 3. Write a program to display contents of file on the screen. The program should ask for file name. Display the contents in capital case.	4. Write a program to find the size of the file. 5. Write a program to combine contents of two files in a third file. Add line number at the beginning of each line. 6. Write a program to display numbers from 1 to 100. Re-direct the output of the program to text file. 7. Write a program to write contents of one file in reverse into another file. 8. Write a program to interchange contents of two files.

V Answer the following questions:

1. What is the difference between end of a file and end of a string? 2. Distinguish between text mode and binary mode operation of a file. 3. What is the use of <code>fseek()</code> ? Explain its syntax. 4. Distinguish between the following functions: (a) <code>scanf()</code> and <code>fscanf()</code> (b) <code>getc()</code> and <code>getchar()</code> (c) <code>putc()</code> and <code>fputc()</code> (d) <code>putw()</code> and <code>getw()</code> (e) <code>ferror()</code> and <code>perror()</code> (f) <code>feof()</code> and <code>eof()</code> 5. How does an append mode differs from a write mode?	6. Why the header file <code>stdio.h</code> is frequently used in C Language? 7. Compare between <code>printf</code> and <code>fprintf</code> functions. 8. Distinguish between the following modes: (a) <code>w</code> and <code>w+</code> (b) <code>r</code> and <code>r+</code> (c) <code>rb</code> and <code>rb+</code> (d) <code>a</code> and <code>a+</code> 9. Explain low-level disk operations. 10. Explain command line arguments. 11. Explain environment variables. 12. How redirection of input and output is done? Explain in brief.

VI What will be the output/s of the following program/s?

1.	<pre># include <process.h> void main() { FILE *fp; char c;</pre>

```

clrscr();
printf("Contents of file
before appending :\n");
fp=fopen("data.txt",
"r");
while(!feof(fp))
{
    c=fgetc(fp);
    printf("%c",c);
}
fp=fopen ("data.txt","a");
if(fp==NULL)
{
    printf("File can not
appended");
    exit(1);
}
printf("\n Enter string to
append :");
while(c!='.')
{
    c=getche();
    fputc(c,fp);
}
fclose(fp);
printf("\n Contents of file
After appending :\n");
fp=fopen("data.txt","r");
while(!feof(fp))
{
    c=fgetc(fp);
    printf("%c",c);
}
}

```

2.

```

# include <process.h>
void main()
{
    FILE *fp;
    Char c=' ';
    Clrscr();
    Fp=fopen("data.txt","w+");
    if(fp==NULL)
    {
        printf("Can not open
file");
        exit(1);
    }
    printf("Write data & to stop
press '.' :");
    while(c!='.')

```

```

{
    c=getche();
    fputc(c,fp);
}
rewind(fp);
printf("\n Contents
read :");
while(!feof(fp))
    printf("%c",getc(fp));
}

```

3.

```

# include <process.h>
void main()
{
    FILE *fp;
    char c=' ';
    clrscr();
    fp=fopen("data.txt","a+");
    if(fp==NULL)
    {
        printf("Cannot open
file");
        exit(1);
    }
    printf("Write data & to stop
press '.' :");
    while(c!='.')
    {
        c=getche();
        fputc(c,fp);
    }
    printf("\n Contents read :");
    rewind(fp);
    while(!feof(fp))
        printf("%c",getc(fp));
}

```

4.

```

# include <process.h>
void main()
{
    FILE *fp;
    char c=' '
    clrscr();
    fp=fopen("data.dat","wb");
    if(fp==NULL)
    {
        printf("Cannot open file");
        exit(1);
    }
}

```

```

printf("Write data & to stop
press '.' :");
while(c!='.')
{
    c=getche();
    fputc(c,fp);
}
fclose(fp);
fp=fopen("data.dat","rb");
printf("\n Contents read :");
while(!feof(fp))
printf("%c",getc(fp));
}

```

5.

```

void main()
{
    FILE *fp;
    char c;
    clrscr();
    fp=fopen("lines.txt","w");
    if(fp==NULL)
    return;
    else
    {
        while( (c=getche())!='*')

```

```

        fputc(c,fp);
    }
    fclose(fp);
}

```

6.

```

void main()
{
    FILE *fp;
    char ch;
    fp=fopen("text.txt","r");
    clrscr();
    if(fp==NULL)
    {
        printf("File Not Found");
        exit(0);
    }
    fseek(fp, 21, SEEK_SET);
    ch=fgetc(fp);
    clrscr();
    rewind(fp);
    for(;;)
    {
        printf("%d\n",ftell(fp));
        ch=fgetc(fp);
    }
}

```

VII Find the bug/s in the following program/s:

1.

```

# include <process.h>
void main()
{
    FILE *fp;
    char c=' ';
    clrscr();
    fp=fopen("data.txt","w");
    if(fp==NULL)
    {
        printf("Cannot read file");
        exit(0);
    }
    printf("Write data & to stop
press . : ");
    while(c!='.')
    {
        c=getche();
        fputc(c,fp);
    }
    printf("\n Contents Read :");
    fp=fopen ("data.txt","r");

```

```

    while(!feof(fp))
    printf("%c",getc(fp));
}

```

2.

```

# include <process.h>
void main()
{
    FILE *fp;
    char c= ' ';
    clrscr();
    fp=fopen("data.txt","w");
    if(fp==NULL)
    {
        printf("Cannot read file");
        exit(0);
    }
    printf("Write data & to stop
press . : ");
    while (c!='.')
    {
        c=getche();

```

```

        fputc(c,fp);
    }
    printf("\n Contents
    Read :");
    fp=fopen("data.txt","r");
    while(!feof(fp))
        printf("%c",getc(fp));
}

```

3.

```

void main()
{
    FILE *fp;
    char text[30];
    clrscr();
    fp=fopen("text.txt","w");
    clrscr();
    puts("\n Enter Text Here");
    gets(text);
    fprintf("%s",text);
}

```

4.

```

#include <process.h>
void main()
{
    FILE *fp;
    char c;
    clrscr();
    fp=fopen("text.txt","r");
    if(fp==NULL)
    {
        printf("\n cannot open
        file");
    }
}

```

```

while( (c=getc(fp)) !=EOF)
    printf ("%c",c);
fclose(fp);
}

```

5.

```

void main()
{
    FILE *fr;
    Char c,file[]="text.txt";
    Fr=fopen(file,"w");
    clrscr();
    while(feof(fr))
    {
        c=fgetc(fr);
        if (ferror(fr))
        {
            perror(file);
            exit(0);
        }
        else
            printf("%c",c);
    }
    fclose(fr);
}

```

6.

```

void main()
{
    char buff[10];
    clrscr();
    setbuf(stdout,buff);
    printf("This Book is very
    good");
    fflush (stdout);
}

```

ANSWERS

I True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	T	2.	T	3.	F	4.	F	5.	T
6.	T	7.	T	8.	F	9.	T	10.	T
11.	F	12.	T	13.	F	14.	T	15.	T
16.	F	17.	T	18.	T	19.	F		

II Match the following correct pairs given in Group A with Group B:

1.

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	B	2.	A	3.	D	4.	C

2.

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	B	2.	D	3.	A	4.	C

III Select the appropriate option from the multiple choices given below:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	a	4.	a	5.	c
6.	a	7.	a						

VI What will be the output/s of the following program/s?

Q.	Ans.
1.	Contents of file before appending : String is terminated with '\0'. Enter string to append : This character is called as NULL character. Contents of file After appending : String is terminated with '\0'. This character is called as NULL character.
2.	Write data & to stop press '.' : ABCDEFGHIJK. Contents read: ABCDEFGHIJK.
3.	Write data & to stop press '.' : This is append and read mode. Contents read: This is append and read mode.
4.	Enter Text Here: Have a nice day.
5.	Have a nice day.
6.	File Not Found.

VII Find the bug/s in the following program/s:

Q.	Ans.
1.	After writing data file should be closed.
2.	The file opening mode "W" is invalid.

3. The file pointer is missing in `fprintf()`.
4. In `if()` statement `exit(0)` statement is required.
5. `while (!eof(fr))` is correct condition.
6. The size of the buffer is too small.

CHAPTER

15 Graphics

Chapter Outline

- 15.1** Introduction
- 15.2** Initialization of Graphics
- 15.3** Few Graphics Functions
- 15.4** Programs Using Library Functions
- 15.5** Working with Text
- 15.6** Filling Patterns with Different Colours and Styles
- 15.7** Mouse Programming
- 15.8** Drawing Non-common Figures

15.1 INTRODUCTION

Computer graphics is one of the most interesting and useful topics of computers and C language. The goal of this chapter is to explore information on graphics. Everywhere information is conveyed in the form of pictures, which needs computer graphics understanding. In other words, pictorial presentation can be done with the graphics. We can draw all figures of various shapes with computer graphics. Lines, circles, ellipse, rectangle, triangle of various shapes can be drawn with computer graphics. Besides, nowadays we deal with animation, multimedia which works pre-dominantly using graphics. The intention of introducing this chapter is to interact with graphics using C language in an easy way. The basic concept of graphics is explained with examples. In this chapter, standard library functions are described together with illustrations. Also discussed in brief are computer graphics.

Video display unit is one of the most useful output components of the computer system. The video display system comprises the following components:

- (i) Video screen, where text and picture can be displayed.
- (ii) Video display adapter card, which is plugged in to one of the expansion slots, provided on the motherboard.

The display adapter facilitates the programmer to show the various figures and text on the screen with different resolutions. Resolution is decided by the number of pixels (small dots) available on the screen. The display resolution is a combination of the number of rows from top to bottom and the number of pixels from left to right on each scan line. Higher the resolution more clear is the picture. With higher number of pixels the picture becomes clearer. Commonly available display adapters are video display adapters, colour graphic adapters and super video graph adapters. All these adapters operate both in text and in graphic modes. The clarity and quality of picture with VGA, SVGA, etc. are better than CGA and monochrome adapters.

The display adapter acts as an inter-mediator between the processor and monitor. The display adapter has two parts: (i) VDU memory and (ii) circuitry that passes contents of VDU memory to the screen. The microprocessor writes the information into VDU memory. The display adapter transfers the information written by a processor to screen. Every address of the VDU memory corresponds to the specific location of the monitor screen.

15.2 INITIALIZATION OF GRAPHICS

The header file `graphics.h` is to be included at the beginning of the program. This header file comprises the definitions and explanations of all the graphic functions. The functions described in this chapter are defined in `graphics.h` file, which should be included in every graphics program.

To switch over to the graphics mode, a function `initgraph()` must be invoked. It selects the best resolution and puts the corresponding number in `gm`, where `gm` is the graphics mode.

15.3 FEW GRAPHICS FUNCTIONS

(i) `initgraph()`;

This function initializes the graphics system. Prototype is defined in '`graphics.h`'. It is declared as given below:

```
void far initgraph(int far *gd, int far *gm, char far *pathtodriver);
```

where `gd` is the graphics driver and `gm` is the graph mode.

In real application to detect graphics drives *DETECT* macro is used.

(ii) `closegraph();`

This function shuts down the graphics system. The declaration is given below. Prototype is defined in '*graphics.h*'.

```
void far closegraph(void);
```

(iii) `restorecrtmode();`

This function restores the screen mode to its pre-initgraph setting. Its prototype is defined in '*graphics.h*'.

```
void far restorecrtmode(void);
```

(iv) `graphresult();`

This function returns an error code for the last failed graphics operation. This function is defined as follows:

```
int far graphresult(void);
```

It returns the error code for the last graphics execution that reported an error and resets the error level to `grOK`, where `grOK` is enumerated error code.

(v) `getmaxx();` and `getmaxy();`

The `getmaxx()` returns maximum *x* screen coordinate and `getmaxy()` returns maximum *y* screen coordinate. Its prototype is defined in '*graphics.h*'. These functions can be declared as given here:

```
int far getmaxx(void);
int far getmaxy(void);
```

(vi) `line();`

It draws a line between two specified points. This function is declared as given below:

```
void far line(int x1, int y1, int x2, int y2);
```

Draws a line from (*x1*, *y1*) to (*x2*, *y2*) using the current colour, line style and thickness.

(vii) `circle();`

This function draws a circle at (*x*, *y*) of the given radius. This function is declared as follows:

```
void far circle(int x, int y, int radius);
```

(viii) `arc();`

This function draws an arc. This function is declared as given here:

```
void far arc(int x, int y, int sa, int ea, int r);
```

where (*x*, *y*) is the centre point. The variables *sa* and *ea* are the start and the end angles in degrees and variable *r* is the radius.

(ix) `ellipse();`

This function draws an elliptical arc. This function is declared as follows:

```
void far ellipse(int x, int y, int sa, int ea, int xr, int yr);
```

where (*x*, *y*) is the centre point. The variables *sa* and *ea* are the start and end angles in degrees, respectively. The variables *xr* and *yr* are horizontal and vertical radii.

(x) `bar();`

This function draws a bar using given co-ordinates. It can be declared as given below:

```
void far bar(int left, int top, int right, int bottom);
```

(xi) `setcolor();`

This function sets the current drawing colour. This is declared as follows:

```
void far setcolor(int color);
```

(xii) `outtextxy();`

It displays a string at the specified location (graphics mode). This function is declared as given below:

```
void far outtextxy(int x, int y, char far *textstring);
```

(xiii) `settextstyle();`

This function sets the current text attributes. This function is declared as follows:

```
void far settextstyle(int font, int direction, int fontsize);
```

(xiv) `settextjustify();`

This function sets text justification for graphics mode. This function is declared as given below:

```
void far settextjustify(int horiz, int vert);
```

It affects text output with `outtext()` function. Text is justified horizontally and vertically.

(xv) `getcolor();`

It returns the current drawing colour. This function is declared as follows:

```
void far getcolor(void);
```

15.4 PROGRAMS USING LIBRARY FUNCTIONS

- (i) `arc()`: This function is useful to draw a circular arc. It requires five arguments. The syntax would be as follow:

```
arc(int x, int y, int a, int b, int c );
```

Here, the first two arguments i.e. `x` and `y` are the centre points. The arguments `a` and `b` are the start and end angles and finally `c` is the radius. The following programming example can be referred for drawing an arc.

➤ 15.1 Write a program to demonstrate the use of the `arc()` function.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turbo2\bgi");
    arc(50,20,50,300,100);
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

Consider the statement `initgraph(&gd,&gm,"d:\turboc2\bgi");` which initializes the graphic mode. The required adapter files are present in the specified directory. The `arc()` function draws arc by simply using parameter values. The function `closegraph()` shuts the graphics mode. The function `restorecrtmode();` restores the screen mode to the previous one. The `DETECT` is enum data type of BGI graphic drivers.

- (ii) `rectangle()`: The `rectangle()` function draws the rectangle. It has four arguments. The syntax of the rectangle is as follows:

```
rectangle(int left, int top, int right, int bottom);
```

The following program illustrates this function.

- 15.2 Write a program to demonstrate the use of `rectangle()` function.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    setcolor(WHITE);
    rectangle (50,20,150,200);
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

This program draws the rectangle using the given arguments. The `setcolor()` sets the colour of drawings.

- (iii) `circle()`: It draws the circle of the given radius. It has three arguments.

`circle (int x, int y, int r);` where r is the radius.

- 15.3 Write a program to draw the circle of the given radius.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    setcolor(WHITE);
    circle(160,260,35);
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

This program draws circle using the given arguments.

- (iv) `line(int a, int b, int c, int d)` : This function draws line from (a, b) to (c, d).

➤ 15.4 Write a program to draw the line.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    line(100,100,100,400);
    getch();
    closegraph();
    restorecrtmode();
}
```

- (v) `ellipse()`: the following function draws ellipse using the given arguments.

➤ 15.5 Write a program to draw ellipse.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    ellipse(100,100,100,300,300,200);
    getch();
    closegraph();
    restorecrtmode();
}
```

➤ 15.6 Write a program to draw circle, line and arc using graphics function.

```
# include <graphics.h>
void main()
{
    int gd=DETECT,gm,x,y,c=0;
    initgraph(&gd,&gm,"c:\\tc");
    x=getmaxx();
    y=getmaxy();
    setcolor(WHITE);
    outtextxy(1,20,"Circle");
```

```

circle(40,100,40);
outtextxy(200,10,"Arc");
arc(x/3,y/8,180,70,30);
line(210,150,110,150);
outtextxy(150,140,"Line");
ellipse(215,150,0,70,150,150);
outtextxy(290,10,"Ellipse");
getche();
closegraph();
restorecrtmode();
}

```

Explanation:

In the above program, a 'graphics.h' header file is included. This header file contains all prototypes and the definition of all graphic functions.

Before starting any drawing action, we need to initialize graphics mode. The `initgraph()` function finds out the best graphics mode and sets the corresponding number in the variable `gm`. Using the value of `gm`, we can find out the details of monitor like its type, resolution, number of video pages and supporting colours.

The variable `gd` is used for graphics driver. Graphics drivers are sub-set of device drivers and are used in only graphics mode. The graphics driver files have an extension .`bgi`. Depending on the type of adapter, one of the supporting device driver file is selected. In graphics mode, cursor disappears and the top left corner (0,0) of the console is considered as beginning.

The above program draws circle, arc, ellipse and line depending on the given values. The user can put different values for the different shapes.

15.4.1 | Program on Moving Moon

➤ 15.7 The following program illustrates a moving moon.

Write a program to draw circle and move it on the screen.

```

# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    clrscr();
    for(x=30;x<400;x+=10)
    {
        fillellipse(x,50,30,30);
        sleep(1);
        clrscr();
    }
    closegraph();
    restorecrtmode();
}

```


Explanation:

In this program, using the `fillellipse()` function moon-like shape is drawn. Using the loop, the drawn figure moved on the screen. The `sleep()` function halts the execution for a second.

- (vi) `bar (int left, int top, int right, int bottom)`: The function `bar()` has four arguments. The `bar()` displays bar mostly shown in graphs.

➤ 15.8 Write a program to display bar.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    bar(10,20,50,190);
    getch();
    closegraph();
    restorecrtmode();
}
```

- (vii) `sector()`: This function is used to draw a sector.

➤ 15.9 Write a program to draw a sector.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    sector(150,170,50,60,90,150);
    getch();
    closegraph();
    restorecrtmode();
}
```

- (viii) `getdrivername()`: This function displays the name of graphics driver name. For example: EGA VGA, EGA VGA.
- (ix) `getmaxcolor()`: It returns the maximum colour available with the adapter.
- (x) `getmaxmode()`: It returns the number of display modes available.
- (xi) `getmaxx()`: It returns the number of maximum *x* coordinate on the screen.
- (xii) `getmaxy()`: It returns the number of maximum *y* coordinate on the screen.
- (xiii) `getmodename(int)`: This function returns the name of the graphics mode.
- (xiv) `lineto()`: This function draws a line. It requires two integer arguments. It draws a line from the current position to (*x*,*y*), where *x* and *y* are the coordinates to be specified by the user.
- (xv) `moveto()`: This function moves the current position to (*x*,*y*), where *x* and *y* are the coordinates to be specified by the user.

The following program illustrates these concepts.

- 15.10 Write a program to shift the line coordinates to other positions with `moveto()` and `lineto()` functions.

```
# include <graphics.h>
# include <stdlib.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    line(20,30,200,30);
    moveto(250,50);
    lineto(200,50);
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

In this program, the `line()` function is used for drawing a line. Also, the effect of `moveto()` and `lineto()` functions can be seen by executing this program. You will observe that the line coordinates get changed with these functions.

15.5 WORKING WITH TEXT

There are several functions for display text in different fonts, sizes and directions. Consider the following functions:

- (i) `outtextxy (int, int, char far *)`: This function is used to display the text in graphic mode.
 - (ii) `settextstyle(int,int,int)`: This functions has three integer arguments. First argument indicates font, second direction and third size.
- 15.11 Write a program to display text in different font and size.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    settextstyle(1,0,30);
    outtextxy(0,120,"AMOL");
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

After the execution of the program, the string 'AMOL' is displayed. This is performed by giving different values in the `settextstyle()` function. Another new function that can be used is `settextjustify()` which takes care of alignment of the text. This function is explained in the following program.

- 15.12 Write a program to display text in different size, font, vertically and horizontally using graphic functions.

```
# include <graphics.h>
# include <stdlib.h>
void main()
{
    int gd=DETECT,gm;
    initgraph(&gd,&gm, "c:\\tc");
    settextstyle(1,0,10);
    settextjustify(0,2);
    outtextxy(10,2,"Hello");
    settextjustify(2,3);
    settextstyle(3,1,5);
    outtextxy(14,150,"Hello");
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

After the execution of the program, the string 'Hello' is displayed horizontally and vertically with different font and font size. This is performed by giving different values in the `settextstyle()` function. Another new function used is `settextjustify()` which takes care of alignment of the text.

15.5.1 | Stylish Lines

- 15.13 Write a program to display stylish lines.

```
# include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\\turboc2\\bgi");
    clrscr();
    line(20,9,20,70);
    setlinestyle(1,1,1);
```

```

    line(40,9,40,60);
    setlinestyle(2,1,1);
    line(60,9,60,70);
    getch();
    closegraph();
    restorecrtmode();
}

```

Explanation:

In this program, using `setlinestyle()` function, a line drawn can be displayed in a different style. It requires three integer arguments. The first represents style of line, second represents pattern and the third represents thickness.

- 15.14 Program to draw lines of different styles with increasing length and displaying text.

```

#include <graphics.h>
void main()
{
    int gd =DETECT,gm,x,y,i,k=20,j=200;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    settextstyle(1,0,30);
    settextjustify(100,50);
    setcolor(GREEN);
    outtextxy(20,40,"line");

    if(k<220)
    for(i=0;i<5;i++)
    {
        setlinestyle(i,10,2);
        line(k,30,j,30);
        k=k+20;
        j=j+20;
        sleep(1);
    }
    else
        getch();
        closegraph();
        restorecrtmode();
}

```

Explanation:

This program is the same as that explained in the previous one. In this program, additional function for text is used. The programmer can run the program and see its effects.

15.6 FILLING PATTERNS WITH DIFFERENT COLOURS AND STYLES

Now, we will use a function to fill the different patterns with different colours using different styles. In the following program, a `setfillstyle()` function is used for drawing a bar. This function needs to pass two parameters: the first being the pattern and the second being the colour.

- 15.15 Write a program to use the `setfillstyle()` and show its effect on the VDU.

```
# include <graphics.h>
# include <stdlib.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    setcolor(GREEN);
    setfillstyle(SOLID_FILL,RED);
    bar(20,30,200,40);
    moveto(250,50);
    bar(200,50,250,60);
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

This program uses `setfillstyle()` function for filling various colours with different styles. A bar is drawn with green colour. The programmer can run the program and see its effects.

- 15.16 Write a program to draw the triband flag by using `setfillstyle()` and show its effect on the VDU.

```
# include <graphics.h>
# include <stdlib.h>
void main()
{
    int gd =DETECT,gm,x,y;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    setfillstyle(1,3);
    bar(20,30,200,40);
    rectangle(20,30,200,40);
    setfillstyle(1,4);
    bar(20,40,200,50);
    rectangle(20,40,200,50);
    setfillstyle(1,5);
    bar(20,50,200,60);
    rectangle(20,50,200,60);
    line(20,30,20,150);
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

This program uses `setfillstyle()` function for filling various bars with different colours. Bars are drawn with different colours. The programmer can run the program and see the output.

- 15.17 Write a program to draw the rectangles duly filled with different patterns using `setfillstyle()` and show its effect on the VDU.

```
# include <graphics.h>
# include <stdlib.h>
void main()
{
    int gd =DETECT,gm, x=20,y=20,i;
    initgraph(&gd,&gm,"d:\turboc2\bgi");
    for(i=0;i<10;i++)
    {
        setfillstyle(i,2);
        bar(x,y,x+50,y+50);
        rectangle(x,y,x+50,y+50);
        x=x+50;
        if(x>500)
        {
            y=y+50;
            x=20;
        }
    }
    getch();
    closegraph();
    restorecrtmode();
}
```

Explanation:

This program uses `setfillstyle()` function for filling the rectangles with different patterns. Rectangles are drawn with different colours. The programmer can run the program and see the output.

- 15.18 Write a program to draw boxes using `bar()` function. Fill the boxes with different designs.

```
#include <graphics.h>
# include <stdlib.h>
void main( )
{
    int gd=DETECT,gm,a=40,b=40,ptr,k;
    char txt[10];
    char *design[]=
    {
        "EMPTY_FILL",      "SOLID_FILL",      "LINE_FILL",
        "LTSLASH_FILL",    "SLASH_FILL",      "BKSLASH_FILL",
    }
```

```

"LTBKSASH_FILL",    "HATCH_FILL",    "XHATCH_FILL",
"INTERLEAVE_FILL",  "WIDE_DOT_FILL",
"CLOSE_DOT_FILL",   "USER_FILL"};
    initgraph(&gd,&gm, "c:\\tc");
    setcolor(RED);
    settextstyle(0,0,2);
    outtextxy(100,1,"User-defined Styles");
    settextstyle(0,0,0);
    setcolor(WHITE);
    for (ptr=0;ptr<12;ptr++)
    {
        setfillstyle(ptr,ptr);
        bar(a,b,a+70,b+70);
        rectangle(a,b,a+70,b+70);
        k=1+ptr;
        itoa(k,txt,10);
        outtextxy(a,b+100,txt);
        setcolor(ptr+1);
        outtextxy(a,b+110,design[ptr]);
        a=a+150;
        if (a>490)
        {
            b=b+150;
            a=40;
        }
    }
    getch();
    closegraph();
    restorecrtmode();
}

```

Explanation:

In the above program, the `settextstyle()` function sets text attribute like font size, direction of text, horizontal or vertical and size of the character. The message 'user-defined styles' is displayed in larger size with red colour. Again, the text attribute are set to 0 so that further they will not affect the outputting text. The character pointer is initialized with pre-defined patterns (designs). The `setfillstyle()` function has two argument patterns and colour to be filled. The `bar()` function draws the bar using the attribute of `setfillstyle()`. The `rectangle()` function draws border to the various bars created by `bar()` function. The `itoa()` function converts an integer to string. The numbers 1 to 12 displayed below each bar are due to this function.

- 15.19 Write a program to draw the rectangle using `floodfill()` function and show its effect on the VDU.

```

# include <graphics.h>
# include <stdlib.h>
void main()
{

```

```

int gd =DETECT,gm,x,y;
initgraph(&gd,&gm,"d:\turboc2\bgi");
setcolor(WHITE);
outtextxy(30,20,"RECTANGLE");
setcolor(GREEN);
rectangle(20,40,200,60);
setfillstyle(1,3);
floodfill(21,41, GREEN);
getch();
closegraph();
restorecrtmode();
}

```

Explanation:

This program uses `setfillstyle()` and `floodfill()` functions for filling the rectangles with any pattern. It is not necessary to use `bar()` at first and then enclose it by rectangle as used in the previous program. Straightway, one can use `floodfill()` function for filling a pattern with different colours. The programmer can run the program and see the output.

15.7 MOUSE PROGRAMMING

- 15.20 Write a program to display the status of mouse button pressed and restrict the mouse between given co-ordinates on the screen. Use `int86()` function to call different ROM-BIOS services.

```

# include <process.h>
# include <dos.h>
# include <graphics.h>
union REGS i,o;
int initmouse(void);
void showarrow(void);
void mousearea(int, int, int,int);
void m_pointerat(int *, int *, int *);

void main()
{
    int gd=DETECT, gm,maxx,maxy,button,x,y;
    initgraph(&gd,&gm,"C:\\tc");
    maxx=getmaxx();
    maxy=getmaxy();
    rectangle(0,56,maxx,maxy);
    setviewport(1,57,maxx-1,maxy-1,1);
    gotoxy(26,1);
    printf("\n Mouse Button pressed");
    if(initmouse()==0)
    {
        closegraph();
    }
}

```



```

        restorecrtmode();
        printf("\n Mouse driver not found");
        exit(1);
    }
    mousearea(0,100,maxx-50,maxy-10);
    showarrow();
    gotoxy(52,3);
    printf("\n Press any key to exit");
    while(!kbhit())
    {
        m_pointerat(&button,&x,&y);
        gotoxy(5,3);
        (button & 1)==1 ? printf("LEFT") : printf("*****");
        gotoxy(20,3);
        (button & 2)==2 ? printf("RIGHT") : printf("*****");
        gotoxy(65,2);
        printf("Cursor Position");
        gotoxy(65,3);
        printf("X=%03d y= %03d",x,y);
    }
}
initmouse()
{
    i.x.ax=0;
    int86(0x33,&i,&o);
    return(o.x.ax);
}
void showarrow()
{
    i.x.ax=1;
    int86(0x33,&i,&o);
}
void mousearea(int x1,int y1,int x2,int y2)
{
    i.x.ax=7;
    i.x.cx=x1;
    i.x.dx=x2;
    int86(0x33,&i,&o);
    i.x.ax=8;
    i.x.cx=y1;
    i.x.dx=y2;
    int86(0x33,&i,&o);
}
void m_pointerat(int *button, int *x,int *y)
{
    i.x.ax=3;
    int86(0x33,&i,&o);
    *button=o.x.bx;
    *x=o.x.cx;
    *y=o.x.dx;
}

```

Explanation:

In the above program, the function `initgraph()` switches the mode from text to graphics. The `DETECT` is a macro which requests the `initgraph()` to set the graphics driver. The functions `getmaxx()` and `getmaxy()` obtain the maximum rows and columns available on the screen. The function `setviewport()` defines the viewport area which bounds the drawing operation inside the area.

The `initmouse()` function checks if the mouse drivers are loaded or not. If the mouse is not initialized then the `closegraph()` function off loads the graphics driver. The `restoremode()` function restores the mode.

The function `mousearea()` defines the area in which mouse pointer can be a pointer. In this function, `int86()` is invoked by initializing CPU register with appropriate values.

The functions `mousearea()` and `showarrow()` calls ROM-BIOS services under 33h with appropriate values. The `while()` loop checks the button pressed and displays messages accordingly. The function `m_pointerat()` displays the current cursor position. When the user presses a key, the program terminates.

➤ 15.21 Write a program to change the mouse cursor.

```
# include <graphics.h>
# include <dos.h>
# include <process.h>
union REGS i,o;
struct SREGS s;
int
cursor[50]={0xaaaa,0xbbbb,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb,
            0xaaaa,0xffff,0xdddd,0xbbbb};

void main()
{
    int mouse(void);
    void cur_shape(int *);
    void showpointer(void);
    int gd=DETECT, gm;
    initgraph (&gd,&gm, "C:\\\\tc");
    if ( mouse()==0)
    {
        closegraph();
        printf ("\n Mouse Supporting files are not installed");
        exit(1);
    }
    gotoxy(10,1);
```

```

    printf("\n Press any key to exit..");
    cur_shape(cursor);
    showpointer();
    getch();
}
mouse()
{
    i.x.ax=0;
    int86(0x33,&i,&o);
    /* retrun(o.x.ax); */
}
void showpointer()
{
    i.x.ax=1;
    int86(0x33,&i,&o);
}
void cur_shape(int *shape)
{
    i.x.ax=9;
    i.x.bx=0;
    i.x.cx=0;
    i.x.dx=(unsigned) shape;
    segread(&s);
    s.es=s.ds;
    int86x(0x33,&i,&i,&s);
}

```

Explanation:

In the above program, an integer array cursor is initialized with certain hexadecimal numbers. The structure *SREGS* is declared. The functions `mouse()` and `showpointer()` are described in the previous program. In appropriate values of the `cur_shape()` function are initialized in the CPU registers. The `segread()` function reads segment registers. The read segment addresses of pointer (s) with interrupt 0x33 and the variables of REGS (i) are passed to function `int86x()`. The cursor shape changes to a square-like shape.

15.8 DRAWING NON-COMMON FIGURES

The following program illustrates the use of `drawpoly()` function. With this, patterns of non-common shapes can be drawn.

➤ 15.22 Draw non-common shape pattern.

```

# include <graphics.h>
# include <stdlib.h>
void main()
{
    int maxx,maxy;
    int array[8];

```

```

int gd =DETECT,gm,x,y;
initgraph(&gd,&gm,"d:\turboc2\bgi");

setcolor(WHITE);
outtextxy(30,20,"NONCOMMON SHAPE");
setcolor(GREEN);
maxx=getmaxx();
maxy=getmaxy();
array[0]=20;
array[1]=40;
array[2]=60;
array[3]=80;
array[4]=maxx/3;
array[5]=maxy/3;
array[6]=array[0];
array[7]=array[1];
drawpoly(4,array);
getch();
closegraph();
restorecrtmode();
}

```

Explanation:

This program uses `drawpoly()` function. The programmer can run the program and see the output. You will get non-regular shape, figure on the screen.

SUMMARY

In this chapter, the reader is exposed to the basics of graphics programming in C. Initialization of graphics with library graphics functions concepts are explained. A number of programming examples are provided. The reader is advised to go through other functions provided in `HELP` of C. Last few programs teach you how to do mouse programming.

EXERCISES

I Fill in the blanks:

1. To switch over to graphics mode a function _____ must be invoked.
 - (a) `initgraph()`
 - (b) `restorecrtmode()`
 - (c) `graphresult()`

2. The prototype for `closegraph()` is defined in _____.
 - (a) `graphics.h`
 - (b) `stdio.h`
 - (c) `conio.h`
3. The `bar()` function draws a bar using _____ coordinates.
 - (a) 2
 - (b) 4
 - (c) 6
4. _____ returns maximum y screen coordinate.
 - (a) `getmaxy()`
 - (b) `getmaxx()`
 - (c) `getmayy()`
5. The function displays a string at the specified location.
 - (a) `outtextxy()`
 - (b) `settextstyle()`
 - (c) `setcolor()`

II True or false:

1. The function `getmaxx()` returns maximum x screen coordinate.
2. One must specify the value of diameter for drawing circle using function `circle()`.
3. Video screen is used only for displaying text.
4. The function `setcolor()` sets the current drawing colour.
5. The `outtextxy()` function displays a string at the specified location.
6. The function `closegraph()` shuts the graphics mode.
7. The function `settextjustify()` sets text justification for graphics mode.
8. The header file `graphics.h` comprises the definitions and explanations of all the graphics functions.
9. The `initgraph()` function find outs the best graphics mode.
10. The function `cirarc()` is a library function.

III Select the appropriate option from the multiple choices given below:

1. For drawing a bar number of arguments to be passed are
 - (a) 3
 - (b) 4
 - (c) 5
 - (d) 6

2. The initialization of graphics under C is done with the function
 - (a) `loadgraph()`
 - (b) `initgraph()`
 - (c) `modegraph()`
3. Name the function to close the graphics mode in C
 - (a) `closeall()`
 - (b) `closegraph()`
 - (c) `cls()`

IV Answer the following questions:

1. Write the initialization procedure of graphics under C.
2. What are the functions of video display adapter?
3. What are the display adapters available in the market?
4. List any 10 graphics library functions and explain their details.

V Attempt the following programs:

1. Write a program to draw two overlapping circles.
2. Write a program to fill overlapping circles with different colours.
3. Write a program to draw 10 rectangles having similar size one after other in one vertical line.
4. Draw alternately bars and rectangles one after other.
5. Draw the different shapes of rectangles and fill them with different colours.

ANSWERS

Fill in the blanks:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	a	2.	a	3.	b	4.	a	5.	a

II True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	T	2.	F	3.	F	4.	T	5.	T
6.	T	7.	T	8.	T	9.	T	10.	F

III Select the appropriate option from the multiple choice given below:

Q.	Ans.
1.	(b) 4
2.	(b) <code>initgraph()</code>
3.	(b) <code>closegraph()</code>

CHAPTER

16

Dynamic Memory Allocation and Linked List

Chapter Outline

- 16.1** Dynamic Memory Allocation
- 16.2** Memory Models
- 16.3** Memory Allocation Functions
- 16.4** List
- 16.5** Traversal of a List
- 16.6** Searching and Retrieving an Element
- 16.7** Predecessor and Successor
- 16.8** Insertion
- 16.9** Linked Lists
- 16.10** Linked List with and without Header

16.1 DYNAMIC MEMORY ALLOCATION

For the execution of a program, it is essential to bring the program in the main memory. When a program does not fit into the main memory, parts of it are brought into the main memory one by one and the full program is executed eventually. Of course, the parts of program which are not currently in main memory are resident either at secondary memory such as floppy or hard disk or at any other magnetic disk. When a new segment of a program is to be moved into a full main memory, it must replace another segment already resident in the main memory. There are memory replacement policies and their description is beyond the scope of this book. Hence, a programmer should not worry about the method of transfer of program from secondary to primary memory or memory replacement policies.

There are two techniques used for storing the information of C programs in the main memory. The first technique involves storing global and local variables, which are fixed during compilation and remain constant throughout the run time program. Here, even the variables of structures and arrays are stored in the main memory during compilation. Space for local variables is allocated from the stack each time when the variable comes into existence. Global and local variables are efficiently stored in C. But the programmer must know the amount of space required for every program.

The second technique by which a program can obtain memory space in the main memory is with dynamic allocation. One of the features provided in C is the allocation of memory at the time of execution of a program. The method of allocation of memory at the time of execution of a program is called dynamic memory allocation. Acquiring main memory by a program and freeing it are done by some library functions provided in C. Memory allocation techniques are described in this chapter. These functions are described in the following sections.

The free region of the memory is called the heap. The size of heap does not remain constant all the time for all the programs. Its size keeps on changing during the execution of program. This is due to creation and destruction of variables that are local to the functions and blocks.

Conceptually the C programs are stored as per Figure 16.1. The heap changes depending on the memory model used. 8086 memory models are described in this chapter. The amount of memory requirement is decided by how the program is designed. For example, if a program is developed with many recursive functions then this is implemented with stack

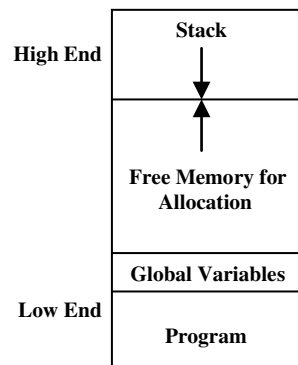


Figure 16.1 A view of a C program's use of memory

16.2 MEMORY MODELS

1. **TINY:** All segment registers are initialized with the identical value and all addressing is accomplished using 16 bits. This means that the code, data and stack all must fit within the same 64-KB segment. Programs are executed quickly in this case.
2. **SMALL:** All codes should fit in a single 64-KB segment and all data should fit in a second 64-KB segment. All pointers are 16 bits in length. Execution speed is the same as tiny model.
3. **MEDIUM:** All data should be fit in a single 64-KB segment; however, the code is allowed to use multiple segments. All pointers to data are 16 bits, but all jumps and calls require 32-bit addresses. Fast access to data is observed, but slower program execution is also noticed with this model.

4. **COMPACT:** All codes should fit in 64 KB segment, but the data can use multiple segments. However, no data item can surpass 64 KB. All pointers to data are 32 bits, but jumps and calls can use 16 bit addresses. There is slow access to data and quick code execution.
5. **LARGE:** Both code and data are allowed to use multiple segments. All pointers are 32 bits in length. However, no single data item can exceed 64 KB. There is slower code execution.
6. **HUGE:** Both code and data are allowed to use multiple segments. Every pointer is of 32-bit in length. Single data item can exceed 64 KB. There is slowest code execution.

16.3 MEMORY ALLOCATION FUNCTIONS

A variable is allowed to use memory to store its value. But permanently memory is not used by the variables. Recent programming languages permit memory space for storing the variables when necessary. They also deallocate the memory of the variables when not required. **Dynamic memory allocation permits to use** memory for the variables at run time with memory allocation functions. In this and the following sections, we discuss the various memory allocation and deallocation functions together with their implementation in the programs.

1. **malloc():** A memory block is allocated with a function called **malloc()**. In other words, the **malloc()** function is used to allocate memory space in bytes to the variables of different data types. The function reserves bytes of determined size and returns the base address to pointer variable. The prototypes are declared in **alloc.h** and **stdlib.h**. The format of the **malloc()** function is as follows:

```
pnt= (data type*) malloc(given size);
```

Here, from data type, compiler understands the pointer type and given size is the size to reserve in the memory. For example:

```
pnt=( int *) malloc(20);
```

Here, in this declaration 20 bytes are allocated to pointer variable **pnt** of type **int** and base address is returned to pointer **pnt**.

A few programs are illustrated below on this function.

- 16.1 Write a program to allocate memory to a pointer variable based on the number of subjects marks to be entered. Compute average marks.

```
# include <alloc.h>
void main()
{
    int k,*p,j=0,sum=0;
    float avg;
    clrscr();
    puts("\n How many subjects marks to be entered : ");
    scanf("%d",&k);
    p=(int *) malloc(k * sizeof(int));
    while(j!=k)
    {
        printf("Subject %d marks=",j+1);
        scanf("%d",p+j);
        j++;
    }
    j=0;
```

```

printf("\n Sum of marks : ");
while(j!=k)
sum=*(p+j++)+sum;
printf("%d",sum);
avg=sum/k;
printf("\n Average marks =%f: ",avg);
getche();
}

```

OUTPUT:

```

How many subjects marks to be entered :
5
Subject 1 marks=88
Subject 2 marks=89
Subject 3 marks=90
Subject 4 marks=87
Subject 5 marks=91
Sum of marks : 445
Average marks =89.000000:

```

Explanation:

In the above program, program prompts user to enter the number of subjects. The `sizeof` function determines the size of data type required to store single value of that type. The size determined by `sizeof` function and the number entered (`k`) are multiplied and the result obtained is nothing but the number bytes to be allocated to a pointer `*p`. The pointer `*p` contains the base address. The first `while` loop reads marks of the subjects through the keyboard. Each time while entering the marks, `'j'` variables value is added to pointer `*p`, which indicates successive address of its type and entered subjects marks are stored in the successive locations. This process is continued till the value of `'j'` reaches to `'k'`. Thus, in a single variable by allocating memory, a programmer can store multiple subjects marks. In the second `while` loop, `sum` and average of marks are calculated.

- 16.2 Write a program to allocate memory to a pointer variable based on the number of items. Display the values of items.

```

void main()
{
    int *ptr;
    int i,item;
    clrscr();
    printf("How many items you want?");
    scanf("%d",&item);
    ptr = (int*)malloc(sizeof(int)*item);
    printf("\nEnter the values of items in Rs:");
    for(i=0;i<item;i++)
        scanf("%d", (ptr+i));
    printf("You have entered the values :\n");
    for(i=0;i<item;i++)
        printf("\n%d", *(ptr+i));
}

```

OUTPUT:

```
How many items you want?3
Enter the values of items in Rs:111
233
100
You have entered the values :
111
233
100
```

Explanation:

Refer to the explanation of Example 16.1.

2. `calloc()`: This function is useful for allocating multiple blocks of memory. It is declared with two arguments. The prototypes are declared in `alloc.h` and `stdlib.h`. The format of the `calloc()` function is as follows:

```
pnt= (int *)calloc(4,2);
```

The above declaration allocates four blocks of memory; each block containing two bytes. The base address is stored in the integer pointer. This function is usually used for allocating memory for array and structure. The `calloc()` can be used in place of the `malloc()` function. The programs illustrated with the `malloc()` function can be executed using the `calloc()` function.

- 16.3 Write a program to allocate memory based on requirement of the number of books to a pointer variable using the `calloc()` function.

```
# include <alloc.h>
void main()
{
    int k,j=0,sum=0;
    int *p;
    clrscr();
    puts("\n How many books to be purchased?");
    scanf("%d",&k);
    p=(int *) calloc(k,2);
    while(j!=k)
    {
        printf("Cost of the book(%d)=",j+1);
        scanf("%d",p+j);
        sum=sum+*(p+j);
        j++;
    }
    printf("\n Total cost of the books: ");
    printf("%d",sum);
}
```

OUTPUT:

```
How many books to be purchased?
4
Cost of the book(1)=100
Cost of the book(2)=200
Cost of the book(3)=400
Cost of the book(4)=500
Total cost of the books: 1200
```

Explanation:

Logic of the above program is the same as the previous one.

Here, the `calloc()` function is used to allocate the memory instead of `malloc()`.

3. `free()`: The `free()` function is used to release the memory if not required. Thus, using this function the wastage of memory is prevented. The declaration of the function is as follows:

```
free(pnt);
```

In the above declaration, `pnt` is a pointer. The `free()` function releases the memory occupied by the pointer variable `pnt`. You can put `free(p)` after the first `while` loop and see its response by executing the program.

- 16.4 Based on the requirement of the number of e-books, write a program to allocate memory using `calloc()` and release it using `free()`. Display the memory locations where the cost of the each e-book is stored and total cost of e-books.

Given below program is supporting `free(pnt)`.

```
# include <alloc.h>
void main()
{
    int j=0,k=5;
    int *p;
    float sum=0.0;
    clrscr();
    p=(int *) calloc(k * sizeof(int),2);
    printf("\nFirst location %u",p);
    printf("\n");
    while(j!=k)
    {
        printf("\nCost of the e-book(%d)=",j+1);
        scanf("%d",p+j);
        printf("(%d)location %u\n",j+1,p+j);
        sum=sum+*(p+j);
        j++;
    }
    printf("\n Total cost of the e-books: ");
    printf("%g",sum);
    free(p);
    printf("\n After freeing the memory the location:%u",p);
}
```

OUTPUT:

```

First location 3176
Cost of the e-book(1)=1000
(1)location 3176
Cost of the e-book(2)=2000
(2)location 3178
Cost of the e-book(3)=3000
(3)location 3180
Cost of the e-book(4)=4000
(4)location 3182
Cost of the e-book(5)=5000
(5)location 3184
    Total cost of the e-books: 15000
    After freeing the memory the location:3176

```

Explanation:

Program logic is the same as used in the earlier programs where `calloc()` is used. In addition to this `free()` is used for freeing the memory. Different locations where the costs of e-books are stored are displayed.

4. `realloc()`: This function reallocates the main memory. This is needed as and when allocated memory is different from the required memory. The prototypes are declared in `alloc.h` and `stdlib.h`. Attempts are made to shrink or enlarge the previously allocated memory by `malloc()` or `calloc()` functions. It returns the address of the reallocated block, which can be different from the original address. If the block cannot be reallocated or size == 0, `realloc()` returns NULL. The declaration of function is as follows:

```
str=(char*) realloc (str, 30)
```

In the above declaration, `str` is a pointer. The `realloc()` function reallocates the memory previously allocated by pointer variable `str` to the new size 30.

- 16.5 Write a program to reallocate memory using `realloc()` function.

```

# include <alloc.h>
# include <string.h>
void main()
{
    char *str;
    str=(char *)malloc(6);
    str("India");
    clrscr();
    printf("str = %s",str);
    str=( char *)realloc(str,30);
    strcpy(str,"Great Researchers' of India");
    printf("\nNow str= %s",str);free(str);
}

```

OUTPUT:

```
str = India
Now str= Great Researchers' of India
```

Explanation:

In the above program, using `malloc()` function six 6 bytes are allocated to character pointer `str`. The character pointer is initialized with string `'India'`. To store more than six characters, we need to allocate more bytes to pointer `str`. Using `realloc()` function memory reallocation takes place. After reallocation the pointer contains 30 bytes. The pointer `str` is again initialized with `'Great Researchers' of India'`. The output displays contents of `str` before and after reallocation. The `free()` function releases the memory allocated.

- 16.6 Write a program to display two numbers with `malloc()` and reallocate memory for displaying three numbers.

```
# include <alloc.h>
# include <string.h>

void main()
{
    int j, *p;
    p=(int *)malloc(2* sizeof(int));
    clrscr();
    for(j=0;j<2;j++)
    {
        printf("number %d=", j+1);
        scanf("\n%d", (p+j));
    }
    printf("\nNumbers are");
    printf("\n");
    for(j=0;j<2;j++)
    printf("number %d= %d\n", j+1, *(j+p));
    printf("\n\n");
    printf("numbers after Reallocation\n");

    p=( int *) realloc (p,6);
    for(j=0;j<3;j++)
    {
        printf("number %d=", j+1);
        scanf("\n%d", (p+j));
    }
    printf("\nnnumbers are\n ");
    for(j=0;j<3;j++)
    printf("number %d= %d\n", j+1, *(j+p));
    free(p);
    getch();
}
```

OUTPUT:

```

number 1=1
number 2=2
Numbers are
number 1= 1
number 2= 2
numbers after Reallocation
number 1=3
number 2=4
number 3=5
numbers are
number 1= 3
number 2= 4
number 3= 5

```

Explanation:

In the above program, using `malloc()` function four bytes are allocated to two integers to pointer `p`. The integer pointer is initialized with `for` loop. To store three integers, we need to allocate more bytes to pointer `p`. Using the `realloc()` function, memory reallocation takes place. After reallocation the pointer contains six bytes. The pointer `p` is again initialized. The output displays contents of `p` before and after reallocation. The `free()` function releases the memory allocated.

5. `coreleft()`: This function returns a measure of unused memory. If the memory model selected is tiny, small or medium, then follow the function declaration as per statement (a). If the memory model selected is compact, large or huge, follow the declaration (b).

(a) `unsigned coreleft(void);`

(b) `unsigned long coreleft (void);`

- 16.7 Write a program to display unused memory using the `coreleft()` function.

```

# include <alloc.h>
void main()
{
    clrscr();
    printf("\n Measure of unused memory = %u",coreleft());
}

```

OUTPUT:

```

Measure of unused memory = 56936

```

Explanation:

In the above program, the function `coreleft()` is invoked without any argument. The function displays unused memory i.e. 56936.

16.4 LIST

A list is a sequential arrangement of elements. The programmer can add, delete or search the individual element in the list. There are two techniques for creating lists. A list is created using an array or linked lists.

First method is to take an array and store the elements. Arrays have many drawbacks. To delete an element from an array or to insert an element into an array, it requires a lot of actions. It may be possible that the required elements in the list may be more/less than declared size of an array. The operations such as deletion, insertion, searching of element can be performed on an array.

The list implemented using an array is called a static list. A list is a series of linearly arranged numbers of the same data type. The list can be of basic data type or custom data type. The elements are positioned one after another and their position number appears in sequence. The first element of the list is called HEAD and the last element is called TAIL.

Please refer to Figure 16.2, where elements having a value 1 is at HEAD position (0th) and element 2 is at TAIL position (5th). The element 5 is a *predecessor* of the element 8 and 4 is a *successor*. Every element can act as a *successor* excluding the first element because it is the first element of the list. The list has the following properties:

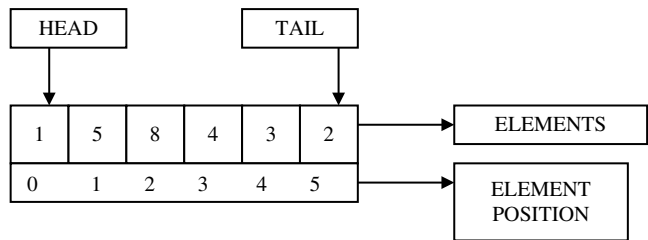


Figure 16.2 Elements of a list

- (i) The list can be enlarged or reduced from the end.
- (ii) The TAIL (ending) position of the list depends on how long the list is extended by the user.
- (iii) Various operations such as transverse, insertion and deletion can be performed on the list.
- (iv) Applying static (array) or dynamic (pointer) a list can be implemented.

16.5 TRAVERSAL OF A LIST

The simple list can be created using an array. In it elements are stored in successive memory locations. Consider the following program.

- 16.8 Write a program to create a simple list of elements. Display the list in original and reverse order.

```
void main()
{
    int arr[5],j;
    clrscr();
    printf("\nEnter five integers : ");
    for(j=0;j<5;j++)
        scanf("%d",&arr[j]);
    printf("\n Elements of List are: ");
    for(j=0;j<5;j++)
        printf(" %d ",arr[j]);
}
```

```

printf("\n Elements of List in reverse :");
for(j=4;j>=0;j--)
printf(" %d",arr[j]);
}

```

OUTPUT:

```

Enter five integers : 4 5 6 7 8
Elements of List are: 4 5 6 7 8
Elements of List in reverse : 8 7 6 5 4

```

Explanation:

Using simple declaration of an array, a list can be implemented. Using `for` loop and `scanf()` statements, five integers are entered. The list can be displayed using `printf()` statement. Once a list is created, various operations such as sorting, searching can be applied. A user is advised to see the chapter *Data Structure: Array* for more information on arrays.

16.6 SEARCHING AND RETRIEVING AN ELEMENT

Once a list is created, we can access and perform operations with the elements. In the last program, all the elements are displayed. One can also specify some conditions such as to display numbers after a specific number or remove the duplicate numbers from the list or finding a specific number or concatenation of two lists, etc. If the list contains large elements, then it may be difficult to find a particular element and its position. Consider the following program on searching an element from the list.

➤ 16.9 Write a program to create a list of integer elements and search the entered number from the list.

```

void main()
{
    int sim[5],j,n,f=0;
    clrscr();
    printf("\nEnter five Integers : ");
    for(j=0;j<5;j++)
    scanf("%d",&sim[j]);
    printf("\n Enter an integer from the entered numbers for
    Search :");
    scanf("%d",&n);
    for(j=0;j<5;j++)
    {
        if(sim[j]==n)
        {
            f=1;
            printf("\n Found ! Position of integer %d is %d ", n,j+1 );
            break;
        }
    }
    if(f==0)
    printf("\n Element not found ! ");
}

```

OUTPUT:

```

Enter five Integers : 1 2 3 4 5

```

Enter an integer from the entered numbers for Search : 4
 Found ! Position of integer 4 is 4

16.7 PREDECESSOR AND SUCCESSOR

In the list of elements, for any location n , $(n-1)$ is the predecessor and $(n+1)$ is successor. In other words, for any location n in the list the left element is a predecessor and the right element is a successor. One can find predecessor and successors of an element in the list. The first element of a list does not have a predecessor and last does not have a successor.

Figure 16.3 shows the predecessor and successor elements of number 10.

The following program displays the predecessor and successor elements of the entered element from the list.

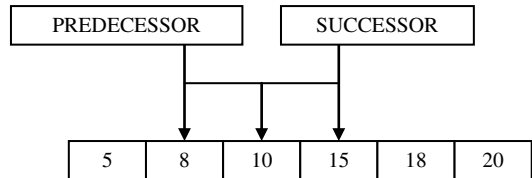


Figure 16.3 Predecessor and successor

➤ 16.10 Write a program to find predecessor and successor of the entered number in a list.

```
void main()
{
    int num[8], j, n, k=0, f=0;
    clrscr();
    printf("\n Enter eight elements : ");
    for(j=0; j<8; j++)
        scanf("%d", &num[j]);
    printf("\n Enter an element : ");
    scanf("%d", &n);
    for(j=0; j<8; j++)
    {
        if(n==num[j])
        {
            f=1;
            (j>0) ? printf("\n The Predecessor of %d is\n", num[j], num[j-1]):
            printf(" No Predecessor");
            (j==7) ? printf("\n No Successor") :
            printf("\n The Successor of %d is %d", num[j], num[j+1]);
            break;
        }
        k++;
    }
    if(f==0)
        printf("\nThe element %d is not found in list", n);
}
```

OUTPUT:

```
Enter eight elements: 1 2 5 8 7 4 4 6
Enter an element: 5
```

```
The Predecessor of 5 is 2
The Successor of 5 is 8
Enter eight elements : 1 2 3 4 5 6 7 8
Enter an element : 1
No Predecessor
The Successor of 1 is 2
Enter eight elements : 12 34 54 76 69 78 85 97
Enter an element : 3
The element 3 is not found in list
```

Explanation:

In this program, eight elements are entered. The user has to enter an element whose predecessor and successors are to be identified. All the elements of the array are checked with the entered number. When the match is found, the next element of the entered number displays as a successor and the previous element displays as a predecessor. If the element entered is the first element of the list then only successor is displayed. If the entered element is the last element of the list then only the predecessor is displayed. The above conditions are checked using *conditional operator* (? :).

16.8 INSERTION

Appending is a process in which a new element is added. However, insertion of an element can also be done in the list. Insertion means an element is added in between two elements in the list. The insertion can be done at the beginning, inside or anywhere in the list.

For a successful insertion of an element, the array-implementing list should have at least one empty location. If the array is full, insertion cannot be possible. The target location where element is to be inserted is made empty by shifting elements downwards by one position and the newly inserted element is placed at that location. Consider Figure 16.4.

5	7	9	10	12		
0	1	2	3	4	5	6

(a)

As per the above figure, two empty spaces are available. Suppose we want to insert 3 in between 7 and 9. All the elements after 7 must be shifted towards the end of the array. The resulting array would be

5	7		9	10	12	
0	1	2	3	4	5	6

(b)

The entered number 3 can be assigned to that memory location and the array would look like. Figure 16.4 describes the insertion of an element. The following program illustrates the insertion operation.

5	7	3	9	10	12	
0	1	2	3	4	5	6

(c)

Figure 16.4 Insertion

- 16.11 Write a program to create a list. Insert some element at the specified location.

```
# include <process.h>
void main()
{
    int num[8]={0},j,k=0,p,n;
    clrscr();
    printf("\n Enter elements ( 0 to exit ) : " );
    for(j=0;j<8;j++)
    {
        scanf("%d",&num[j]);
        if( num[j]==0)
            break;
    }
    if(j<8)
    {
        printf("\n Enter Position number and element : ");
        scanf("%d %d",&p,&n);
        --p;
    }
    else
        while(num[k]!=0)
        {
            k++;
            k--;
            for(j=k;j>=p;j--)
                num[j+1]=num[j];
            num[p]=n;
            for(j=0;j<8;j++)
                printf(" %d ", num[j]);
        }
}
```

OUTPUT:

```
Enter elements (0 to exit) : 1 2 3 4 5 6 0
Enter Position number and element: 5 10
1 2 3 4 10 5 6 0
```

Explanation:

In the above program, an array `num[8]` is declared. The user has to enter elements continuously. A user can enter zero to exit from the continuous input routine. If zero is entered, the `break` statement takes control out of the `for` loop. Thus, the list contains a few empty memory locations, which are enough to carry out insertion operations. When a user enters eight elements then the program ends. In this case, due to the lack of space, insertion operation is impossible.

The position number where an element is to be inserted is prompted by the program. After entering these values by the user, using `while` loop, the position of the first vacant location is determined. From user-specified position number, next all successive elements are shifted one memory location towards down side of array. Due to shifting one memory, space is generated and the entered element is placed at that location.

The reader can perform other operations on list such as the deletion of an element, sorting of elements and merging two lists.

16.9 LINKED LISTS

A linked list is implemented with pointers. This method is called Dynamic implementation because all operations as described above on the list can be implemented without complication. By applying increment, operation on pointer successive locations of memory can be accessed and an element can be stored in that location. Thus, the series of element can be continued to any number of elements. The programmer has to keep the starting address of the pointer in which the first element is stored. Thus, in the same way, the numbers can be viewed or altered. Here is the simple example.

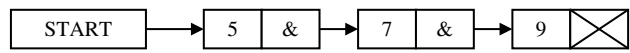
Singly Linked list

In this type of linked list, two successive nodes of the linked list are linked with each other in sequential linear manner, Figure 16.5 describes singly link list. It is like a train, in which two successive bogies are connected. The train is an example of single linked list.

A linked list is a dynamic data structure with the ability to expand and shrink as per the program requirement.

The singly linked list is easy and straightforward data structure as compared to other structure. By changing the link

position, other type of linked list such as circular, doubly linked list can be formed. For creating a linked list, the structure is defined as follows:



& indicates address of the next node

Figure 16.5 Single linked list

```
struct node
{
    int number;
    struct node *p;
};
```

The above structure is used to implement the linked list. In it the *number*, variable entered numbers are stored. The second member is pointer to the same structure. The pointer **p* points to the same structure. Here, though the declaration of `struct node` has not been completed, the compiler permits the pointer declaration of the same structure type. However, the variable declaration is not allowed. This is because the pointers are dynamic in nature whereas variables are formed by early binding. The declaration of objects inside the `struct` leads to the preparation of very complex data structure. This concept is called object composition and its detailed discussion is out of the scope of this book.

We are familiar with the array and we know the importance of the base address. Once a base address is obtained, later successive elements can be accessed. In the linked list, also a list can be created with or without *header node*. The *head* holds the starting address.

➤ 16.12 Write a program to create a simple linked list.

```
struct list
{
    int n;
    struct list *p;
};
```

```

void main()
{
    struct list item0, item1, item2;
    item2.n=3;
    item2.p=NULL;
    item1.n=5;
    item1.p=&item2.n;
    item0.n=7;
    item0.p=&item1.n;
    clrscr();
    printf("\n Linked list elements are :");
    printf(" %d ", item0.n);
    printf(" %d ", *item0.p);
    printf(" %d ", *item1.p);
    printf(" %d ", *item2.p);
}

```

OUTPUT:

Linked list elements are: 7 5 3 0

Explanation:

In the above program, the structure list is declared with two elements `int n` and `struct list *p`. The pointer `*p` recursively points to the same structure. The `struct list item0`, `item1` and `item2` are three variables of type list. Consider the initialization

```

item2.n=3;
item2.p=NULL;

```

The `item2` is the third (last) node of the list. The pointer `*p` is initialized with a null. This is because node 2 is the last node and after this node no node exists and thus it need not require to pointing any address.

```

item1.n=5;
item1.p=&item2.n;

```

In this node, `n` is assigned with five. The pointer points to the data field of the next node, i.e. `item2.n` (7).

```

item0.n=7;
item0.p=&item1.n;

```

16.10 LINKED LIST WITH AND WITHOUT HEADER

16.10.1 | Linked List with Header

The following steps are used to create linked list with header:

1. Three pointers header, first and rear are declared. The header pointer is initially initialized with NULL. For example, `header=NULL`, where the header is pointer to structure. If it remains NULL, it implies that the list has no element. Such list is known as the NULL list or empty list. Figure 16.6 explains header pointer initialization.

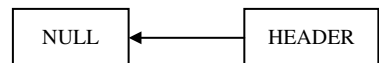


Figure 16.6 Header pointer initialization

- In the second step, memory is allocated for the first node of the linked list. For example, the address of the first node is 1888. An integer say 8 is stored in the variable `num` and value of header is assigned to pointer `next`. Figure 16.7 enlightens memory allocation to the first node.

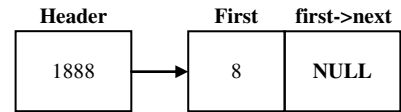


Figure 16.7 Memory allocation to the first node

Both the header and rear are initialized the address of first node.
The statement would be

```
header=first;
rear=first;
```

- The address of pointer `first` is assigned to pointers `header` and `rear`. The `rear` pointer is used to identify the end of the list and to detect the NULL pointer.
- Again, create a memory location, suppose 1890, for the successive node.
- Join the element of 1890 by assigning the value of node `rear->next`. Move the `rear` pointer to the last node. Figure 16.8 provides the last node and other pointers.

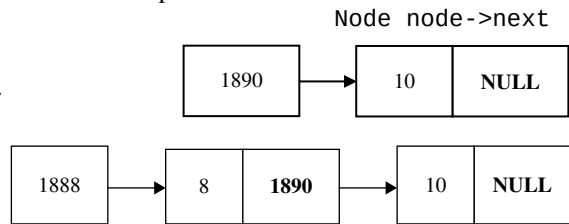


Figure 16.8 Rear pointer to the last node with other pointers

The following program explains the above concept.

➤ 16.13 Write a program to create the linked list with header.

```
# include <malloc.h>
struct num
{
    int num;
    struct num *next;
}
*header, *first, *rear;
void main()
{
    void form(void);
    form();
    clrscr();

    printf("\n Linked list elements are : ");
    while (header!=NULL)
    {
        printf(" %d ", header->num);
        header=header->next;
    }
}
void form(void)
{
    struct num *node;
    printf("\n Enter number : ");
```



```

if(header==NULL)
{
    first=(struct num*)malloc(size of(struct num));
    scanf("%d",&first->num);
    first->next=header;
    header=first;
    rear=first;
}
while(1)
{
    node=(struct num*) malloc(sizeof(struct num));
    scanf("%d",&node->num);
    if(node->num==0) break;
    node->next=NULL;
    rear->next=node;
    rear=node;
}
}

```

OUTPUT:

```

Enter number : 1 3 4 8 7 9 0
Linked list elements are : 1 3 4 8 7 9

```

Explanation:

The above program is an example of linked list with header. Three structure pointers **header*, **first* and **rear* are declared after structure declaration. Initially, these pointers are NULL because they are declared as global. The `form()` function is used to create linked list nodes. Inside the function `form()` another structure pointer **node* is defined and its scope is local to the same function. The procedure for creating the first and later successive nodes is different. The `if()` statement checks the value of the header. The value of header is NULL. The `malloc()` function allocates memory to pointer *first* and the entered number is stored in variable *num* of the node. In the same `if()` block, both the pointer's header and rear are assigned the value of *first*. Once the first node is created, next time the execution of the `if()` block is not required. The `while` loop iterated continuously and successive nodes are created by allocating memory to the pointer *node*. Consider the statements

1. `node->next=NULL;`

The above statement assigns NULL to the pointer next of current node. If users do not want, create more nodes. The linked list can be closed here.

2. `rear->next=node;`

The rear pointer keeps track of the last node, the address of current node (*node*) is assigned to link field of the previous node.

3. `rear=node;`

Before creating the next node, the address of the last created node is assigned to the pointer **rear*, which is used for statement (2). In function `main()`, using while loop the elements of linked list are displayed.

Header: The pointer **header* is very useful in the formation of the linked list. The address of the first node (1888) is stored in pointer **header*. The value of the header remains unchanged until it turns out to be NULL. Pointer **header* only can determine the starting location of the list only.

```

while(header!=NULL)
{
    printf(" %d ",header->num);
    header=header->next;
}

```

We can perform different operations on the linked list. They are as follows.

(a) Traversing a Link List:

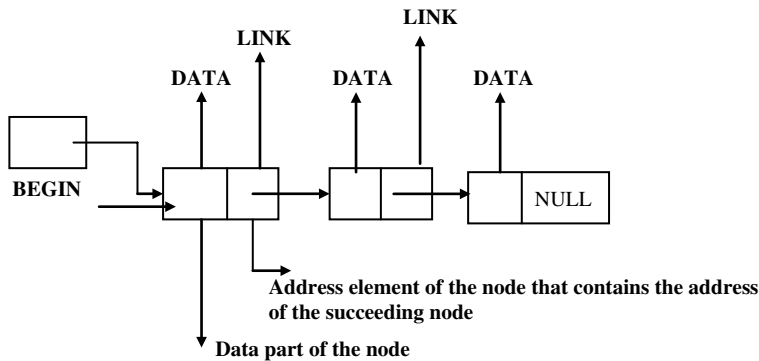


Figure 16.9 Linked list

In the above linked list, the data is the information part of the structure (Figure 16.9). The link is the address of the coming element. **BEGIN** is the address of the first element. We declare **P** as a pointer variable. First **p** points to the address of the **BEGIN** that points to the first element of the list. For accessing the next element, we give the address of the next element to the **P** as:

p=p->LINK

Now, the **P** has the address of the next element. We can traverse the entire list till the **P** has **NULL** address.

- (b) **Searching a Link List :** Searching refers to the searching of an element into a linked list. For searching the element, initially we transverse the linked list and with traversing the list we compare data part of each element with the given element.
- (c) **Insertion into a Linked List:** Insertion into a linked list is possible in two methods (Figure 16.10)
 1. Insertion at beginning
 2. Insertion in between

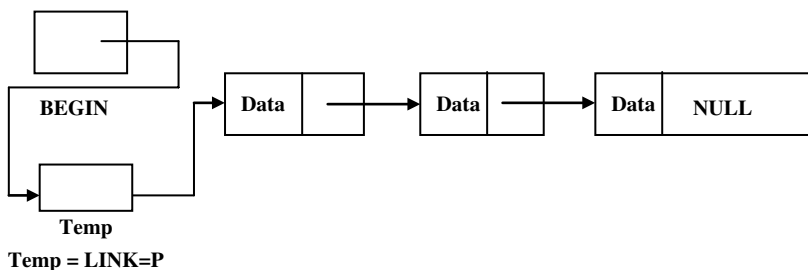


Figure 16.10 Insertion of element at the beginning of the linked list

P is the address of the BEGIN. For inserting at the beginning address of the P is given to the link part.

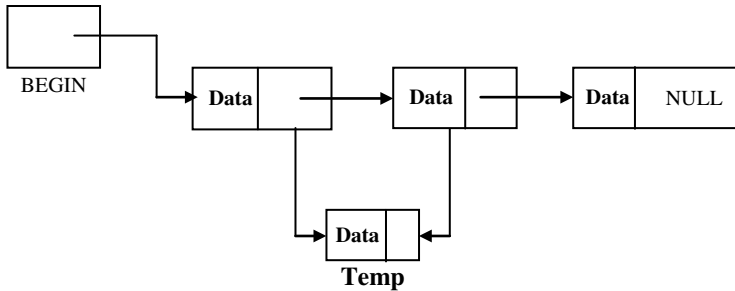


Figure 16.11 Insertion of element in between the linked list

First we transverse the list for finding the node. After that, we insert the element. For inserting the element after the particular node, we give the address of that node to the link part of the inserted node and address of the inserted node is arranged into the LINK part of the former node (Figure 16.11).

(a) Deletion from linked list

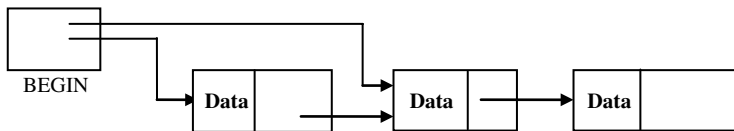


Figure 16.12 Deletion of element at the beginning of linked list

For deleting the node from the linked list, first we transverse the linked list and compare the every element (Figure 16.12). If the element is the first element then we give the address of the link part of node to the pt . Here, pt is another pointer, which points to the address of BEGIN.

$pt = p \rightarrow \text{LINK}$

If the element is other than the first element, then we give the address of the link part of the deleted node to the link part of the previous node (Figure 16.13).

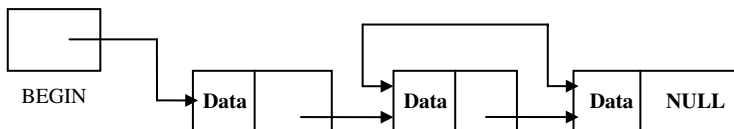


Figure 16.13 Deletion of element in between the linked list

- 16.14 Write a program to create a linked list. Add, delete and insert the element in the linked list. Also display and count the elements of the linked list.

```
# include <alloc.h>

struct node
{
    int data;
    struct node *next;
}
*p;

printf("Enter the element : ");
void addatstart(struct node *pt,int);
void append(struct node *pt, int, int);
void erase(struct node *pt,int);
void show(struct node *pt);
void count(struct node *pt);
void descending(struct node *pt);

void main()
{
    int n,m,po,d,i,j;
    char ch='y';
    clrscr();
    p=NULL;
    do
    {
        clrscr();
        printf(" 1 Generate\n");
        printf(" 2 Add at starting\n");
        printf(" 3 Append \n");
        printf(" 4 Delete\n");
        printf(" 5 Show\n");
        printf(" 6 Count\n");
        printf(" 7 Descending\n");
        printf("Enter your choice : ");
        scanf("%d",&n);

        switch(n)
        {
            case 1:
                printf("\n How many node you want : ");
                scanf("%d",&i);
                for(j=0;j<i;j++)
                {
                    printf("enter the element : ");
                    scanf("%d",&m);
                    gen_rate (p,m);
                }
                show(p);
                break;
            case 2:
                printf("Enter the element : ");
```

```

        scanf("%d",&m);
        addatstart(p,m);
        show(p);
        break;
    case 3 :
        printf("\n Enter the element and position ");
        scanf("%d %d",&m,&po);
        append(p,m,po);
        show(p);
        break;
    case 4 :
        printf("\n Enter the number for deletion :");
        scanf("%d",&d);
        erase(p,d);
        show(p);
        break;
    case 5 :
        show(p);
        break;
    case 6 :
        count(p);
        break;
    case 7:
        descending(p);
        break;
    default:
        printf("\n Enter value between 1 to 7");
    }

    printf("\n Do u want to continue (Y/N)");
    ch=getche();
}
while (ch=='Y' || ch=='y');
}

void gen_rate(struct node *q,int num)
{
    if(q==NULL)
    {
        p=(struct node *)malloc(sizeof(struct node));
        p->data=num;
        p->next=NULL;
    }
    else
    {
        while(q->next!=NULL)
            q=q->next;
        q->next=(struct node *)malloc(sizeof(struct node));
        q->next->data=num;
        q->next->next=NULL;
    }
}

```

```

void addatstart( struct node *q, int num)
{
    p=(struct node *)malloc(sizeof(struct node));
    p->data=num;
    p->next=q;
}
void append( struct node *q,int num, int c)
{
    struct node *tmp;
    int i;
    for(i=0;i<c-2;i++)
    {
        q=q->next;
        if(q==NULL)
        {
            printf("There are less than %d elements\n",c);
            return;
        }
    }
    tmp=(struct node *)malloc(sizeof(struct node));
    tmp->next=q->next;
    tmp->data=num;
    q->next=tmp;
}
void erase( struct node *q, int num)
{
    struct node *t;
    if(q->data==num)
    {
        p=q->next;
        free(q);
        return;
    }
    while(q->next->next!=NULL)
    {
        if(q->next->data==num)
        {
            t=q->next;
            q->next=q->next->next;
            free(t);
            return;
        }
        q=q->next;
    }
    if(q->next->data==num)
    {
        free(q->next);
        p->next=NULL;
        return;
    }
}

```

```

    printf("element %d not found\n", num);
}

void show( struct node *q)
{
    printf("Your list is :\n");
    while(q!=NULL)
    {
        printf("%d\t", q->data);
        q=q->next;
    }
    printf("\n");
}

void count ( struct node *q)
{
    int c=0;
    while (q!=NULL)
    {
        q=q->next;
        c++;
    }
    printf("\n Number of element are %d\n", c);
}

void descending( struct node *x)
{
    struct node *q,*r,*s;
    q=x;
    r=NULL;
    while(q!=NULL)
    {
        s=r;
        r=q;
        q=q->next;
        r->next=s;
    }
    p=r;
    show(p);
}

```

OUTPUT:

```

1 Generate
2 Add at starting
3 Append
4 Delete
5 Show
6 Count
7 Descending
Enter your choice : 1
How many node you want : 4
enter the element : 1

```

```

enter the element : 5
enter the element : 4
enter the element : 7
Your list is :
1   5   4   7
Do u want to continue (Y/N) N

```

Explanation:

In the above program, a self-referential structure node is declared with member variables `int data` and integer pointer `next`. A menu appears with seven options on the screen. The menu options are given as per given in the output. Different functions are defined to perform the tasks given in the menu.

The `gen_rate()` function is used to create a linked list. When the user enters the number of nodes, the `for` loop in the first case structure is executed and at each iteration a number is entered by the user. The entered number and pointer 'p' are passed to function `gen_rate()`. If the pointer 'p' contains `NULL`, the `malloc()` function allocates memory to pointer 'p'. The number entered is assigned to data part of the structure and `NULL` value is assigned to the address part. If the pointer 'p' contains value other than `NULL`, in such a case the `while` loop transverses the entire list and the number is added at the end of the list. The `show()` function displays the total elements of the linked list.

The function `addatstart()` adds the entered elements at the beginning of the linked list. In this function, the `malloc()` function allocates the memory to pointer 'p'. The entered number is assigned to variable and the pointer variable `next` is initialized with the address of the `next` node.

The function `append()` adds the elements at given position in the linked list. The `for` loop is executed till the pointer 'q' is not equal to the position number. When the element number is found the entered number is inserted using `tmpstructure` variable.

The function `erase()` is used to delete the given number from the linked list. In this function, the entered number is checked in the entire list. When the number occurs, the `free()` function releases the memory of that node.

The function `count()` counts the total numbers present in the linked list; the function `descending()` displays the elements in reverse order and the function `show()` displays the number. These functions are self-explanatory.

SUMMARY

In this chapter, you are introduced to the few unique features of C like dynamic memory allocation, linked lists and graphics. You learnt how to allocate the memory using `malloc()`, `calloc()` and `realloc()` functions and release the allocated memory using `free()` function. The linked list is described in brief in this chapter. In the linked list, the creation of linked list, traversing, searching, inserting and deleting element are described with figures. The reader is advised to refer to the book authored by me on *Introduction to Data structures in C* for a detailed understanding of the linked lists.

EXERCISES

I True or false:

1. Singly Linked list uses random searching method.
2. `calloc(m, n) = m*malloc(n)`
3. `malloc(8);` it will allocate the memory of size 8 bytes and initialize it with 0.
4. `calloc(4, 4);` it will allocate 4 blocks of memory with 4 bytes each and also initialize it with 0.
5. Doubly linked list moves in forward and/or backward direction.
6. `free()` function releases the memory reference on invocation.
7. The link list is self referential structure.

II Select the appropriate correct option from the following:

1. The function used to allocate the memory is
 - (a) `alloc()`
 - (b) `malloc()`
 - (c) `free()`
 - (d) none of the above.
2. The different memory manipulation functions are defined in the header file
 - (a) `process.h`
 - (b) `stdio.h`
 - (c) `malloc.h`
 - (d) `alloc.h`
3. The singly linked list structure contains the declaration of this data types.
 - (a) `int` and `struct node *`
 - (b) `int` and `int`
 - (c) `float` and `char *`
 - (d) none of the above.
4. The singly linked list is
 - (a) unidirectional
 - (b) bidirectional
 - (c) moves forward only
 - (d) moves backward only
5. `pnt= (int *)calloc(4,2);` this statement allocates
 - (a) 16 bytes of memory
 - (b) 6 bytes of memory
 - (c) 4 blocks with 2 bytes each
 - (d) a block of size 8 bytes.

III Find out the bugs from the programs given below:

1.


```
#include<alloc.h>
void main()
{
    int *a;
    a=malloc(sizeof(int),2);
    *a=100;
    printf("The value of a is %d", *a);
}
```
2.


```
#include<alloc.h>
struct node{
    int a;
    struct node *p;
}
```

3.

```
#include<alloc.h>
void main()
{
    int *d;
    float *s;
    char *c;
    c=(int *)
    malloc(6*sizeof(char));
    d=(int *)malloc(sizeof(int));
    s=(int *)malloc(sizeof(float));
    *d=100;
    *s=34.2;
    *c="HELLO"
    printf("%d\t%f\t%s", *d, *s, c);
}
```

4.

```
#include<alloc.h>
struct node
{
    int a;
    struct node *p;
}
*k;
struct node* create(int i)
{
```

```
struct node*k;
k=(struct node *)
malloc(sizeof(struct node));
k->a=i;
k->p=NULL;
}
void main()
{
    k=create(10);
    printf("%d ", k->a);
}
```

5.

```
#include<alloc.h>
struct node
{
    int a;
    struct node *p;
}
*head;
void main()
{
    head=(struct node*)
    calloc(sizeof(struct node));
    head->a=100;
    printf("%d ", head->a);
}
```

IV Find the output from the following program:

1.

```
#include<alloc.h>
void main()
{
    int *a[4];
    int i;
    for(i=0;i<4;i++)
    a[i]=(int *)malloc(2);
    for(i=0;i<4;i++)
    *a[i]=i+1;
    for(i=0;i<4;i++)
    printf("%d ", *a[i]);
}
```

2.

```
#include<alloc.h>
struct node
{
    int a;
    struct node *next;
}
```

```
*k;
struct node* make_node(int i)
{
    struct node*m;
    m=(struct node *)
    malloc(sizeof(struct node));
    m->a=i;
    m->next=NULL;
    return m;
}
void main()
{
    k=make_node(239);
    printf("%d ", k->a);
}
```

3.

```
#include<alloc.h>
int main()
{
```

```

int *i;
float *f;
i=(int *)malloc(sizeof(int));
f=(float *)malloc(sizeof(float));
*i=100;
*f=34.2;
printf("%d\t%f", *i, *f);
return 0;
}

```

4.

```

#include<alloc.h>
int main()
{
    int *i;
    int k;
    i=(int *)
    calloc(4, sizeof(int));
    for(k=0; k<4; k++)
        i[k]=k+3;
}

```

5.

```

for(k=0; k<4; k++)
    printf("%d\n", *(i+k), k[i]);
return 0;
}

#include<alloc.h>
void main()
{
    int *a;
    int j;
    a=(int *)calloc(5, 2);
    for(j=0; j<4; j++)
        j[a]=j+1;
    for(j=0; j<4; j++)
    {
        if(a[j]==3)
            printf("\n Yes I got it!");
        else printf("\n Sad...");
    }
}

```

V Attempt the following exercises:

- Write a program to display first five letters of alphabets using `malloc()`.
- Write a program to enter integers using `malloc()` and search for the position of entered number.
- Write a program to find maximum number amongst the float array and allocate the float memory using `calloc()`.
- Write a program a program to allocate initially memory with `malloc()` and then with `calloc()` and check out the difference between them.
- Write a program for singly linked list to display few elements and search in decreasing order.

ANSWERS

I True or false:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	F	2.	T	3.	F	4.	T	5.	T
6.	T	7.	T						

II Select the appropriate correct option from the following:

Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	b	2.	d	3.	a	4.	c	5.	c

III Find out the bugs from the programs given below:

Q.	Ans.
1.	Error is in line no 5: Extra parameter is 2.
2.	A semi-colon is missing after struct node declaration.
3.	All the pointers are type casted with int*. To get the correct output they must be type casted with appropriate data types.
4.	Error return k is missing in the function create(int i).
5.	Error is in calloc() function, one parameter is missing. Line 2 in main() should be head=(struct node*) calloc(1,sizeof (struct node)).

IV Find the output from the following program:

Q.	Ans.	Q.	Ans.	Q.	Ans.
1.	1 2 3 4	4.	3 3	5.	Sad
2.	239		4 4		Sad
3.	100 34.2		5 5		Yes I got it!
			6 6		Sad

APPENDIX A

AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
000	(NUL)	Null char
001	☉ (SOH)	Start of Heading
002	☼ (STX)	Start of Text
003	♥ (ETX)	End of Text
004	♦ (EOT)	End of Transmission
005	♣ (ENQ)	Enquiry
006	♠ (ACK)	Acknowledgment
007	• (BEL)	Bell
008	▣ (BS)	Back Space
009	○ (HT)	Horizontal Tab
010	▣ (LF)	Line Feed
011	♂ (VT)	Vertical Tab
012	♀ (FF)	Form Feed
013	♪ (CR)	Carriage Return
014	♫ (SO)	Shift Out / X-On
015	☼ (SI)	Shift In / X-Off
016	► (DLE)	Data Line Escape
017	◄ (DC1)	Device Control 1 (often XON)
018	↕ (DC2)	Device Control 2
019	!! (DC3)	Device Control 3 (often XOFF)
020	¶ (DC4)	Device Control 4
021	§ (NAK)	Negative Acknowledgement
022	— (SYN)	Synchronous Idle
023	↕ (ETB)	End of Transmit Block
024	↑ (CAN)	Cancel
025	↓ (EM)	End of Medium
026	→ (SUB)	Substitute
027	← (ESC)	Escape
028	^\\ (FS)	File Separator
029	↔ (GS)	Group Separator

(continued)

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
030	▲ (RS)	Record Separator
031	▼ (US)	Unit Separator
032	(BL)	Space (Blank)
033	!	Exclamation mark
034	“	Double quotes (or speech marks)
035	#	Number
036	\$	Dollar
037	%	Percentage
038	&	Ampersand
039	‘	Single quote
040	(	Open parenthesis (or open bracket)
041	)	Close parenthesis (or close bracket)
042	*	Asterisk
043	+	Plus
044	,	Comma
045	-	Hyphen
046	.	Period, dot or full stop
047	/	Slash or divide
048	0	Zero
049	1	One
050	2	Two
051	3	Three
052	4	Four
053	5	Five
054	6	Six
055	7	Seven
056	8	Eight
057	9	Nine
058	:	Colon
059	;	Semi-colon
060	<	Less than (or open angled bracket)
061	=	Equals
062	>	Greater than (or close angled bracket)
063	?	Question mark
064	@	At symbol
065	A	Uppercase A

(continued)

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
066	B	Uppercase B
067	C	Uppercase C
068	D	Uppercase D
069	E	Uppercase E
070	F	Uppercase F
071	G	Uppercase G
072	H	Uppercase H
073	I	Uppercase I
074	J	Uppercase J
075	K	Uppercase K
076	L	Uppercase L
077	M	Uppercase M
078	N	Uppercase N
079	O	Uppercase O
080	P	Uppercase P
081	Q	Uppercase Q
082	R	Uppercase R
083	S	Uppercase S
084	T	Uppercase T
085	U	Uppercase U
086	V	Uppercase V
087	W	Uppercase W
088	X	Uppercase X
089	Y	Uppercase Y
090	Z	Uppercase Z
091	[	Opening bracket
092	\	Backslash
093	]	Closing bracket
094	^	Caret – circumflex
095	_	Underscore
096	`	Grave accent
097	a	Lowercase a
098	b	Lowercase b
099	c	Lowercase c
100	d	Lowercase d
101	e	Lowercase e
102	f	Lowercase f

(continued)

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
103	g	Lowercase g
104	h	Lowercase h
105	i	Lowercase i
106	j	Lowercase j
107	k	Lowercase k
108	l	Lowercase l
109	m	Lowercase m
110	n	Lowercase n
111	o	Lowercase o
112	p	Lowercase p
113	q	Lowercase q
114	r	Lowercase r
115	s	Lowercase s
116	t	Lowercase t
117	u	Lowercase u
118	v	Lowercase v
119	w	Lowercase w
120	x	Lowercase x
121	y	Lowercase y
122	z	Lowercase z
123	{	Opening brace
124		Vertical bar
125	}	Closing brace
126	~	Equivalency sign – tilde
127	△	Delete
128	Ç	Latin capital letter C with cedilla
129	ü	Latin small letter u with diaeresis
130	é	Latin small letter e with acute
131	â	Latin small letter a with circumflex
132	ä	Latin small letter a with diaeresis
133	à	Latin small letter a with grave
134	å	Latin small letter a with ring above
135	ç	Latin small letter c with cedilla
136	ê	Latin small letter e with circumflex
137	ë	Latin small letter e with diaeresis
138	è	Latin small letter e with grave
139	ï	Latin small letter i with diaeresis
140	î	Latin small letter i with circumflex

(continued)

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
141	ì	Latin small letter i with grave
142	Ä	Latin capital letter A with dieresis
143	Å	Latin capital letter A with ring above
144	É	Latin capital letter E with acute
145	æ	Latin small letter æ
146	Æ	Latin capital letter Æ
147	ô	Latin small letter o with circumflex
148	ö	Latin small letter o with dieresis
149	ò	Latin small letter o with grave
150	û	Latin small letter u with circumflex
151	ù	Latin small letter u with grave
152	ÿ	Latin small letter y with dieresis
153	Ö	Latin capital letter O with dieresis
154	Ü	Latin capital letter U with dieresis
155	¢	Cent
156	£	Lira
157	¥	Yen
158	₧	Peseta
159	f	Latin small letter f with hook
160	á	Latin small letter a with acute
161	í	Latin small letter i with acute
162	ó	Latin small letter o with acute
163	ú	Latin small letter u with acute
164	ñ	Latin small letter n with tilde
165	Ñ	Latin capital letter N with tilde
166	ª	Feminine ordinal indicator
167	º	Masculine ordinal indicator
168	¿	Inverted question mark
169	¬	Reversed not
170	¬	Not
171	½	Vulgar fraction one half
172	¼	Vulgar fraction one quarter
173	¡	Inverted exclamation
174	«	Left-pointing double angle quotation
175	»	Right-pointing double angle quotation
176	░	Light shade
177	▒	Medium shade

(continued)

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
178	■	Dark shade
179		Box drawings light vertical
180	├	Box drawings light vertical and left
181	┤	Box drawings vertical single and left double
182	├┐	Box drawings vertical double and left single
183	└┐	Box drawings down double and left single
184	┤┐	Box drawings down single and left double
185	├┤	Box drawings double vertical and left
186	├├	Box drawings double vertical
187	└├	Box drawings double down and left
188	├┬	Box drawings double up and left
189	└┬	Box drawings up double and left single
190	┤┬	Box drawings up single and left double
191	└┴	Box drawings light down and left
192	┬	Box drawings light up and right
193	┬┐	Box drawings light horizontal and up
194	┴	Box drawings light horizontal and down
195	┤	Box drawings light vertical and right
196	—	Box drawings light horizontal
197	┼	Box drawings light vertical and horizontal
198	┤┐	Box drawings vertical single and right double
199	├┐	Box drawings vertical double and right single
200	┬┐	Box drawings double up and right
201	┴┐	Box drawings double down and right
202	├┬	Box drawings double up and horizontal
203	┴┬	Box drawings double down and horizontal
204	├┤	Box drawings double vertical and right
205	==	Box drawings double horizontal
206	├├	Box drawings double vertical and horizontal
207	┬┐	Box drawings up single and horizontal double
208	┬┬	Box drawings up double and horizontal single
209	┴┐	Box drawings down single and horizontal double
210	┴┬	Box drawings down double and horizontal single
211	┬┤	Box drawings up double and right single
212	┬┤	Box drawings up single and right double

(continued)

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
213	⌞	Box drawings down single and right double
214	⌏	Box drawings down double and right single
215	⌐	Box drawings vertical double and horizontal single
216	⌑	Box drawings vertical single and horizontal double
217	⌒	Box drawings light up and left
218	⌓	Box drawings light down and right
219	■	Full block
220	▀	Lower half block
221	▁	Left half block
222	▂	Right half block
223	▄	Upper half block
224	α	Greek small letter Alpha
225	β	Greek small letter Beta
226	Γ	Greek capital letter Gamma
227	π	Greek small letter Pi
228	Σ	Greek capital letter Sigma
229	σ	Greek small letter Sigma
230	μ	Greek small letter Mu
231	τ	Greek small letter Tau
232	Φ	Greek capital letter Phi
233	θ	Greek small letter Theta
234	Ω	Greek capital letter Omega (ohm)
235	δ	Greek small letter Delta
236	∞	Infinity
237	ø	Latin small letter o with stroke
238	Є	Cyrillic capital letter Ukrainian IE
239	∩	Intersection
240	≡	Identical to
241	±	Plus-minus
242	≥	Greater than or equal to
243	≤	Less than or equal to
244	∫	Top half integral
245	∫	Bottom half integral
246	÷	Division
247	≈	Almost equal to

(continued)

<i>ASCII Value</i>	<i>Symbol</i>	<i>Description</i>
248	◦	White bullet
249	•	Bullet
250	.	Decimal point
251	√	Square root
252	η	Greek small letter Eta
253	²	Superscript two
254	■	Black square
255		

APPENDIX B

PRIORITY OF OPERATORS AND THEIR CLUBBING

<i>Operators</i>	<i>Operation</i>	<i>Associativity or Clubbing</i>	<i>Priority</i>
()	Function call	Left to right	1st
[]	Array expression or square bracket		
—>	Structure operator		
.	Structure operator		
+	Unary plus	Right to left	2nd
—	Unary minus		
++	Increment		
--	Decrement		
!	Not operator		
~	Ones complement		
*	Pointer operator		
&	Address operator		
sizeof	Size of an object		
typecast	Type cast		
*	Multiplication	Left to right	3rd
/	Division		
%	Modular division		
+	Addition (Binary plus)	Left to right	4th
—	Subtraction (Binary minus)		
<<	Left shift	Left to right	5th
>>	Right shift		
<	Less than	Left to right	6th
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7th
!=	Inequality (Not equal to)		
&	Bitwise AND	Left to right	8th

(continued)

<i>Operators</i>	<i>Operation</i>	<i>Associativity or Clubbing</i>	<i>Priority</i>
<code>^</code>	Bitwise XOR	Left to right	9th
<code> </code>	Bitwise OR	Left to right	10th
<code>&&</code>	Logical AND	Left to right	11th
<code> </code>	Logical OR	Left to right	12th
<code>?:</code>	Conditional operator	Right to left	13th
<code>=, *=, /=, %=,</code> <code>+=, -=, &=,</code> <code>^=, =, <<=,</code> <code>>==</code>	Assignment operators	Right to left	14th
<code>,</code>	Comma operator	Left to right	15th

APPENDIX C

HEADER FILES AND STANDARD LIBRARY FUNCTIONS

Every C compiler provides the standard library functions. The following table describes few C library functions that can be used by programmers in their programs. Since standard library functions are huge in numbers, it is not possible to include all of them in this appendix. Programmer can refer to the help of the C compiler. Functions supported by compilers may vary from compiler to compiler.

Header files and functions supported by a compiler are as follows.

1. `stdio.h`

Function Name & Syntax	Description	Example
<code>printf:</code> <code>int printf(const char *format, argument, ...);</code>	It applies each argument to corresponding format specifier in <code>*format</code> and sends formatted output to <code>stdin</code> . If successful, returns non-zero; otherwise EOF#.	• <code>printf("%d", d);</code> : prints value of d. (65) • <code>printf("%c", 65);</code> : prints ASCII character of 65 i.e. 'A' • <code>printf("s=%f, t=%d", s, t);</code> : prints float value of s and int value of t. (s=4.5,t=2)
<code>scanf:</code> <code>int scanf(const char *format, address, ...);</code>	It scans a series of input fields one character at a time, format each field according to a corresponding format specifier in <code>*format</code> and stores the formatted input at an address passed as an argument. If successful, returns number of fields scanned; returns zero, if no fields are stored. Returns EOF#, when it attempts to read end of file.	• <code>scanf("%d", &s);</code> : reads the value for s and stores in s. • <code>scanf("%d%c", &i, &c);</code> : reads value for i as int and value for c as char and stores in i and c. • <code>scanf("%s", p);</code> : reads string for p and stores in char array p.
<code>gets:</code> <code>char *gets(char *s);</code>	It collects a string of characters terminated by a new line from the standard input stream <code>stdin</code> and puts it into s by replacing newline by null character (\0). It allows whitespaces in string. If successful, returns string s; otherwise null.	If we enter "C language" in screen for input; • <code>scanf</code> only takes "C" as string • <code>gets</code> takes "C Language" as string • <code>gets(a);</code> : entered string will be stored in char array a.
<code>puts:</code> <code>int puts(const char *s);</code>	It copies the null-terminated string s to the standard output stream <code>stdout</code> and appends a newline character. If successful, returns non-zero; otherwise EOF#.	<code>puts(a);</code> : The entire string with whitespaces stored in char array will get printed on screen.
<code>fopen:</code> <code>FILE *fopen(const char *filename, const char *mode);</code>	<code>fopen</code> opens a file and associate a stream with it and returns a pointer that identifies the stream in subsequent operations. Mode indicates in which mode a file will be accessed (r,w,a).	• <code>FILE *fp=fopen("a.txt", "r");</code> : opens file a.txt in read mode. • <code>FILE *fc=fopen("s.c", "w");</code> : opens file s.C in write mode. • <code>FILE *fd=fopen("p.doc", "a");</code> : opens file p.doc in append mode.

(continued)

<i>Function Name & Syntax</i>	<i>Description</i>	<i>Example</i>
<code>fclose:</code> <code>int fclose(FILE *stream);</code>	<code>fclose</code> closes the named stream and returns 0 on success; otherwise EOF#. All buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing.	<code>fclose(fp);</code> <code>fclose(fc);</code> <code>fclose(fd);</code>
<code>fprintf:</code> <code>int fprintf(FILE *stream, const char *format, argument, ...);</code>	It applies each argument to corresponding format specifier in <code>*format</code> and sends formatted output to stream. If successful, returns non-zero; otherwise EOF#.	<code>fprintf(fp, "s=%f, t=%d", s, t);</code> Writes "s=4.5,t=2" to a file indicated by <code>fp</code> .
<code>fscanf:</code> <code>int fscanf(FILE *stream, const char *format, address, ...);</code>	It scans a series of input fields one character at a time from a stream, format each field according to a corresponding format specifier in <code>*format</code> and stores the formatted input at an address passed as an argument. If successful, returns number of fields scanned; returns zero, if no fields are stored. Returns EOF# when it attempts to read end-of-file.	<code>fscanf(fp, "%s", &str);</code> Reads a string from a file indicated by <code>fp</code> and stores it in char array <code>str</code> .
<code>fgets:</code> <code>char *fgets(char *s, int n, FILE *stream);</code>	It reads characters from stream into the string <code>S</code> . It stops when it reads either <code>n - 1</code> characters or a newline character, whichever comes first. <code>fgets</code> retains the newline character at the end of <code>S</code> and appends a null byte to <code>S</code> to mark the end of the string. <code>fgets</code> returns the string pointed to by <code>S</code> .	<code>fgets(str, 10, fp);</code> Reads 9 characters or newline character, whichever comes first from a file pointed by <code>fp</code> and stores them into char pointer <code>str</code> .
<code>fputs:</code> <code>int fputs(const char *s, FILE *stream);</code>	<code>fputs</code> copies the null-terminated string <code>S</code> to the given output stream. It does not append a newline character, and the terminating null character is not copied. <code>fputs</code> returns the last character written.	<code>fputs(str, fp);</code> Writes the characters in string <code>str</code> to a file pointed by <code>fp</code> .
<code>fgetc:</code> <code>int fgetc(FILE *stream);</code>	It returns the next character on the named input stream. It returns the character read, after converting it to an <code>int</code> on success; EOF# otherwise.	<code>fgetc(fp);</code> Reads next character from file indicated by <code>fp</code> .
<code>fputc:</code> <code>int fputc(int c, FILE *stream);</code>	It outputs character <code>C</code> to the named stream. It returns the character written on success; EOF# otherwise.	<code>fputc(65, fp);</code> Writes 'A' to a file indicated by <code>fp</code> .
<code>fflush:</code> <code>int fflush(FILE *stream);</code>	If the given stream has buffered output, <code>fflush</code> writes the output for stream to the associated file.	

#EOF: end-of-file

2. conio.h

<i>Function Name & Syntax</i>	<i>Description</i>	<i>Example</i>
<code>clrscr:</code> <code>void clrscr(void);</code>	It clears the current text window and places the cursor in the upper left-hand corner (at position 1,1).	<code>clrscr();</code>
<code>getch:</code> <code>int getch(void);</code>	It reads a single character directly from the keyboard, without echoing to the screen and returns the character read from keyboard.	<code>getch();</code>

3. alloc.h

<i>Function Name & Syntax</i>	<i>Description</i>	<i>Example</i>
<code>malloc:</code> <code>void *malloc</code> <code>(size_t size);</code>	It allocates a block of size bytes from the memory heap. It allows a program to allocate memory explicitly as it's needed, and in the exact amounts needed. On success, returns pointer to newly allocated block of memory; otherwise null.	<code>char *str=(char *) malloc(10);</code> Allocates space for 10 chracters pointed by str. Typcasting is needed here.
<code>calloc:</code> <code>void *calloc</code> <code>(size_t nitems,</code> <code>size_t size);</code>	<code>calloc</code> provides access to the C memory heap, which is available for dynamic allocation of variable-sized blocks of memory. <code>calloc</code> allocates a block (<code>nitems * size</code>) bytes and clears it to 0. On success, returns pointer to newly allocated block of memory; otherwise null.	<code>int *str=(int *) calloc(10, sizeof(int));</code> Allocates space for 10 chracters as <code>10*sizeof(int)=10*2=20</code> bytes pointed by str. Typcasting is also needed here.
<code>free:</code> <code>void free(void</code> <code>*block);</code>	<code>free</code> deallocates a memory block allocated by a previous call to <code>calloc</code> , <code>malloc</code> , or <code>realloc</code> .	<code>free(str);</code> Frees the allocated block pointed by str.

Note: Above three functions are also available in `stdlib.h` file.

4. math.h

<i>Function Name & Syntax</i>	<i>Description</i>	<i>Example</i>
<code>abs:</code> <code>int abs(int x);</code>	It returns absolute value of its argument.	<code>abs(-1);</code> returns 1
<code>ceil:</code> <code>double ceil</code> <code>(double x);</code>	It finds the smallest integer not less than X.	<code>ceil(2.6);</code> returns 3.0
<code>floor:</code> <code>double floor</code> <code>(double x);</code>	It finds the largest integer not greater than X.	<code>floor(2.6);</code> returns 2.0
<code>exp:</code> <code>double exp(double x);</code>	It calculates the exponential function; e to the xth power.	<code>exp(2);</code> returns $e^2=7.389056$

(continued)

Function Name & Syntax	Description	Example
log: double log(double x); double log10 (double x);	log calculates the natural logarithm of x while log10 calculates the base 10 logarithm of x.	log(5); returns 1.609438 log10(5); returns 0.698970
pow: double pow(double x, double y);	It calculates x^y i.e. x to the power y.	pow(2, 5); returns 32
Trigonometric: double cos(double x); double sin (double x); double tan(double x);	COS computes the cosine of the input value. SIN compute the sine of the input value. TAN calculates the tangent of the input value. The value to be passed as argument must be in radians.	cos(30*M_PI/180); #returns 0.866025 sin(30*M_PI/180); returns 0.500000 cos(30*M_PI/180); returns 0.577350
sqrt: double sqrt (double x);	It calculates the positive square root of the input value.	sqrt(6.25); returns 2.500000

#M_PI= π =3.141593

5. stdlib.h

Function Name & Syntax	Description	Example
atoi: int atoi (const char *s);	It converts a string pointed to by s to int. If s is inconvertible, it returns zero.	char *s="123"atoi(s); returns an integer 123.
exit: void exit(int status);	It terminates the calling process. Before termination, it closes all files and writes buffered output. status=0 indicates normal exit and non-zero indicates error. This function is also available in process.h file.	exit(0); stops the execution of main function. Hence, terminates the program.
itoa: char *itoa(int value, char *string, int radix);	It converts value to a null-terminated string and stores the result in string. radix specifies the base to be used in converting value (2 to 36).	itoa(123, a, 16); The hexadecimal string format of 123 i.e. 7b will be stored in string a.
random: int random(int num);	It returns a random number between 0 and (num-1).	random(999); may return 10

APPENDIX D

ROM-BIOS SERVICES

These services can be invoked using **int86()** function.

PRINT SCREEN Interrupt : 0×5 Returns : Nothing Use : Sends contents of the screen to the printer	INITIALIZE PRINTER Interrupt : 0×17 Input : AH = 0×01 DX = Printer number (0 = LPT1, 1 = LPT2, 2 = LPT3) Use : Initializes the printer
MEMORY SIZE Interrupt : 0×12 Input : Nothing Returns : AX (memory size)	READ DISK CONTROLLER Interrupt : 0×13 Input : AH = 0×0 DL = drive $0 \times 00-0 \times 7F$ Floppy disks $0 \times 80-0 \times FF$ Hard disks
SET CURSOR SIZE Interrupt : 0×10 Input : AH = 0×01 CH = Starting Scan line CL = Ending Scan line Use : Changes the cursor size.	SET CURSOR POSITION Interrupt : 0×10 Input : AH = 0×02 BH = Display Page Number DH = Row number DL = Column number
REBOOT COMPUTER Interrupt : 0×19 Input : Nothing Use : Restarts the computer	DELAY Interrupt : 0×15 Input : AH = 0×86 CX: DX = Number of micro seconds to halt Use : Same as delay() function
GET TIME Interrupt : 1AH Input : AH = 02H Returns : CH = Hours in BCD CL = Minutes in BCD DH = Seconds in BCD DL = Daylight time code 00h-Standard time 01h-Daylight saving time BCD-Binary coded decimal.	GET DATE Interrupt : 1AH Input : AH = 04H Returns : CH = Century in BCD CL = Year in BCD DH = Month in BCD DL = Day in BCD Carry flag is clear if the clock is running, otherwise carry flag is set.
SHOW MOUSE POINTER Interrupt : 33H Input : AX = 0001H Returns : Nothing Shows the mouse pointer and neutralizes any mouse pointer area previously defined.	HIDE MOUSE POINTER Interrupt : 33H Input : AX = 0002H Returns : Nothing Removes the mouse pointer. The driver keeps on to track the mouse pointer.

These services can be invoked using `intdos()` function. The interrupt number for all services is 0×21 .

MAKE DIRECTORY Input : AH = 0×39 DS : DX = Segment : offset address Returns : If function is successful carry flag is clear otherwise carry flag is set and AX contains error code.	REMOVE DIRECTORY Input : AH = $0 \times 3A$ DS : DX = Segment : offset address Returns : If function is successful carry flag is clear otherwise carry flag is set and AX contains error code.
CHANGE DIRECTORY Input : AH = $0 \times 3B$ DS : DX = Segment : offset address of directory name. Returns : If function is successful carry flag is clear otherwise carry flag is set and AX contains error code.	DELETE FILE Input : AH = 0×41 DS : DX = Segment : offset address Returns : If function is successful carry flag is clear otherwise carry flag is set and AX contains error code.

APPENDIX E

SCAN CODES OF KEYBOARD KEYS

<i>Key</i>	<i>Normal</i>	<i>Shift</i>	<i>Ctrl</i>	<i>Alt</i>
A	1E61	1E41	1E01	1E00
B	3062	3042	3002	3000
C	2E63	2E43	2E03	2E00
D	2064	2044	2004	2000
E	1265	1245	1205	1200
F	2166	2146	2106	2100
G	2267	2247	2207	2200
H	2368	2348	2308	2300
I	1769	1749	1709	1700
J	246A	244A	240A	2400
K	256B	254B	250B	2500
L	266C	264C	260C	2600
M	326D	324D	320D	3200
N	316E	314E	310E	3100
O	186F	184F	180F	1800
P	1970	1950	1910	1900
Q	1071	1051	1011	1000
R	1372	1352	1312	1300
S	1F73	1F53	1F13	1F00
T	1474	1454	1414	1400
U	1675	1655	1615	1600
V	2F76	2F56	2F16	2F00
W	1177	1157	1117	1100
X	2D78	2D58	2D18	2D00
Y	1579	1559	1519	1500
Z	2C7A	2C5A	2C1A	2C00
1	0231	0221		7800
2	0332	0340	0300	7900
3	0433	0423		7A00
4	0534	0524		7B00

(continued)

<i>Key</i>	<i>Normal</i>	<i>Shift</i>	<i>Ctrl</i>	<i>Alt</i>
5	0635	0625		7C00
6	0736	075E	071E	7D00
7	0837	0826		7E00
8	0938	092A		7F00
9	0A39	0A28		8000
0	0B30	0B29		8100
—	0C2D	0C5F	0C1F	8200
=	0D3D	0D2B		8300
[	1A5B	1A7B	1A1B	1A00
]	1B5D	1B7D	1B1D	1B00
;	273B	273A		2700
“	2827	2822		
`	2960	297E		
\	2B5C	2B7C	2B1C	2600
,	332C	333C		
.	342E	343E		
/	352F	353F		
F1	3B00	5400	5E00	6800
F2	3C00	5500	5F00	6900
F3	3D00	5600	6000	6A00
F4	3E00	5700	6100	6B00
F5	3F00	5800	6200	6C00
F6	4000	5900	6300	6D00
F7	4100	5A00	6400	6E00
F8	4200	5B00	6500	6F00
F9	4300	5C00	6600	7000
F10	4400	5D00	6700	7100
F11	8500	8700	8900	8B00
F12	8600	8800	8A00	8C00
BackSpace	0E08	0E08	0E7F	0E00
Del	5300	532E	9300	A300
Down Arrow	5000	5032	9100	A000
End	4F00	4F31	7500	9F00
Enter	1C0D	1C0D	1C0A	A600
Esc	011B	011B	011B	0100
Home	4700	4737	7700	9700
Ins	5200	5230	9200	A200

(continued)

<i>Key</i>	<i>Normal</i>	<i>Shift</i>	<i>Ctrl</i>	<i>Alt</i>
Keypad 5	4C35	8F00		
Keypad*	372A	9600	3700	
Keypad—	4A2D	4A2D	8E00	4A00
Keypad +	4E2B	4E2B	4E00	
Keypad /	352F	352F	9500	A400
Left arrow	4B00	4B34	7300	9B00
Page Down	5100	5133	7600	A100
Page UP	4900	4939	8400	9900
Prn scr	372A			
Right Arrow	4D00	4D36	7400	9D00
SpaceBar	3920	3920	3920	3920
Tab	0F09	0F00	9400	A500
Up arrow	4800	4838	8D00	9800

INDEX

A

- abs () function, 398
- advantages of C program, 17
- algorithm
 - analysis of, 19
 - classification of, 21–22
 - definition, 18
 - examples, 19
 - instruction, 18
 - program design, 19–20
- alphanumeric symbols, 8
- American National Standard Institute (ANSI)
 - standard, 7
- AND Gate, 188–189
- AND (&&) operator, 70, 73
- ANSI C predefined macros, 475
- arc () functions, 592
- array, 337
 - addition of even and odd numbers using, 229–230
 - addition of even numbers and product of odd numbers, 230–231
 - amount of memory required for, 226
 - arranging numbers in increasing and decreasing order, 248–249
 - assigning/equating variables, 224
 - base, 223
 - calculating the total cost, 237–238
 - characteristics of an, 224–226
 - comparison with structure, 491
 - declaration of a one-dimensional, 221
 - definition, 221
 - delete operation with two-dimensional array, 255–269
 - deleting specified element and rearranging the elements, 240–241
 - detecting the occurrence of a character in a given string, 231
 - of different data types, 221
 - displaying balance and code number, 258–259
 - displaying binary bits corresponding to Hexadecimal numbers, 267–268
 - displaying character array with their address, 228
 - displaying contents of two, 234
 - displaying elements of string with their element numbers, 238
 - displaying elements of two arrays in two separate columns, 258
 - displaying matrix form, 259–262
 - displaying names of cities with their base addresses, 266–267
 - displaying names of days of a week using single-dimensional array, 233–234
 - displaying number of days of a given month of a year, 235–236
 - displaying number of days of different months of year, 234–235
 - displaying one-dimensional array using integer pointer, 243–245
 - displaying text using stdin and stdout streams, 247
 - displaying the given message by using putc () and stdout () functions, 238
 - displaying two-dimensional elements together with their addresses, 257
 - elements, 222
 - evaluating price and quantity available in the form of an, 265–266
 - evaluation of series, 250–251
 - even[] and odd[], 229–230
 - finding the average sales of an item, 236
 - finding the occurrence of similar numbers, 236–237
 - four-dimensional, 271–272
 - and functions, 412–418
 - index, 223
 - initialization, 222–223, 226

- initializing single- and two-dimensional arrays, 258–259
- inserting an element at a specified position in, 241–243
- insertion/deletion of an element, 225
- insert operation with two-dimensional array, 253–255
- of integers, 222
- linear, 276
- memory requirement, 226–227
- merging of two or more, 244–245
- multi-dimensional, 269–274
- NULL character, 228
- one-dimensional, 226–238
- operations with, 240–246
- of pointers, 352–357
- predefined streams, 478
- printing a string in reverse order, 231–232
- program to convert binary to gray codes, 268–269
- program to display elements with their addresses, 224–225
- range, 223
- searching a specified element in an, 240–241
- size, 223
- sorting an, 242–243
- sorting numbers in ascending order, 246
- sorting of strings, 245–246
- sprintf () function, 274–276
- sscanf () function, 273
- static type, 226
- storage, 224
- of structures, 491–492
- syntax, 222–223
- terminology, 223
- three-dimensional, 270–271, 273–274
- traversing of, 238–239
- two-dimensional, 251–269
- types, 223
- unsigned character data type, 232–233
- upper boundary of, 225
- word, 223
- arr [p], 354–355
- *(arr1p), 354
- ASCII Code, 34
- assembler, 9
- assembly language, 8

- assertions, 480
- assignment operator (=), 399–400
- assignment operators, 59–62
- asterisk (*) symbol, 225, 337, 642
- atof () function, 315
- atoi () function, 316
- automatic variables, 435–436
- auto variables, 448–450

B

- bar () function, 593, 597, 600–604
- base address, 223, 242, 266–267
- BASIC, 6, 8
- basic combined programming language (BCPL), 6–7
- basic input–output system (BIOS), 5
- binary arithmetic operators, 64
- binary code, 7
- BIOS service, 519–520, 604, 606
- bit field, 510–513
- bits, 337
- bitwise operators, 74–79
 - AND, 76–77
 - bit right, 74–75
 - bits left, 75–76
 - exclusive OR (XOR), 78–79
 - OR, 77–78
- ‘B’ language, 5–6
- Bluetooth technology, 5
- Bohm And Jacopini’s theory, 213
- brackets ([]), 225
- break statement, 129, 130, 132–133, 163
- byte, 337

C

- call1 () function, 448
- calloc () function, 616–618
- C compilers, 7
- Celeron-based laptop, 5
- central processing unit (CPU), 5
- c.exe, 576, 577
- C functions, 6
- change () function, 393–395
- character codes, 34
- character constant, 33–34
- character I/O, 99–100

- cgets (), 103
- cputs (), 103–104
- getcch (), 101–102
- getchar (), 100–101
- gets (), 102–103
- putchar (), 101–102
- puts (), 102–103
- character set, 33–34
- circle () functions, 591
- close () function, 585–587
- closegraph () functions, 592
- clrscr (), 93–95
- COBOL, 6, 8
- coding a program, 21
- command line arguments, 576–577
 - application of, 577–579
- compilers, 9
- compile time initialization, 222
- composite data structure, 221
- computer basics
 - central processing unit (CPU), 5
 - input device, 4
 - memory, 4–5
 - output device, 4
- computers, impact of, 3
- conditional compilation directives, 469–471
 - #ifdef, 471–472
- conditional operator, 63–64
- conditional statements, 69–71
- conio.h header file, 104
- constant declared variables, 50
- constants, 36–38
 - program on various constants, 38
- constant variables, 50
- continue statement, 130, 163
- coreleft () function, 620
- counter-controlled repetitions, 155
- CPU registers, 455, 518
- C standard library, 6
- cube () function, 398–399

D

- data, definition, 534
- data communication, 534
- data types, 40–41
 - floats and double floats, 42–43
 - integer, 41

- signed and unsigned char, 42
- decision-making statements
 - break statement, 129, 133–134
 - continue statement, 130, 163
 - distinction between switch case and nested ifs, 146–147
 - goto statement, 130–132
 - goto unconditional jump, 129
 - if–else–if ladder statement, 121–129
 - if–else statement, 113, 116–120
 - if statement, 114–116
 - loop statement, 113
 - nested if–else statement, 120–121
 - nested switch statements, 144–145
 - switch case statement, 132–143
- declaration of structures, 492–495
- declaration of variables, 44–45
- decrement operator, 65–67
- definite repetition action, 155
- delimiters, 34
- destination stream, 534
- deterministic algorithm, 22
- display adapter, 591
- division operations, 48–49
- dos.h header file, 105
- DOS prompt, 580
- DOS service, 518–524
 - changing the cursor in different sizes, 520
 - creating a directory using, 520–521
 - displaying the system time at a specified cursor position, 519–520
 - to display the attributes of a file using, 521–522
 - to display the given character on the screen using, 521
 - finding memory size of the computer, 519
 - program to delete a file using, 523
- double floating, 41
- double () function, 397
- do-while loop
 - to compare factorial of a given number, 196–197
 - to display a message, 207–208
 - to evaluate series of numbers, 185
 - execution of program if the condition is false, 207
 - to find cubes of numbers, 165

- functions and, 409–412
- to print an entered number in reversed order, 202–203
- program to find age and weight of students, 210–211
- program to find number is prime or not, 210
- syntax, 207
- with while statement loop, 212–213
- do-while statement, 156
- drawpoly () function, 607–608
- dynamic implementation, 626
- dynamic initialization, 46
- dynamic memory allocation, 337, 613

E

- EBCDIC, 34
- ellipse () functions, 591
- enumeration data type, 513–517
 - creating, for 12 months, 513–514
 - displaying the name of month using, 514–516
 - programs using, 514
 - program to identify the type of entered character using, 514–515
- environment variables, 579–580
- escape sequences, 95
- execution of C program, 12–15
- exit (), 104
- EX-OR GATE, 191–192
- external variables, 450–453
- extern keyword, 453
- extra arguments, 380

F

- fact () function, 424–425
- fcloseall () function, 535, 568–569
- feof () function, 561
- fgetc () function, 535–539, 548
- files
 - abc.txt, 581–582
 - alpha.txt, 581–582
 - binary modes, 543–544
 - closing of, 573–574
 - communication with programs and output device, 533
 - data, 533
 - fgetc () function, 535–539, 559–561

- fprint () function, 535, 545
- fputc () function, 538–541
- fread () function, 535, 553–555
- fscanf () function, 545–546
- fseek () function, 558–559
- fwrite () function, 553–555
- low-level disk I/O operations, 572–575
- opening a file for read/write operation in
 - binary mode, 543–544
- opening a file in append mode and add new records in it, 541
- opening a file in read/write mode in it, 542
- opening a pre-existing file and add information at the end of file, 538–539
- opening a text file and write some text using
 - fprintf () function, 545
- opening modes, 572–573
- opening of, 536
- program to copy the contents of one file to another file, 567–568, 570–571
- program to copy up to 100 characters from a file to an array, 569–570
- program to display C program files in current directory, 560–561
- program to read a character from the keyboard till the user presses enter, 580–581
- program to write data to text file and read it, 538
- random access, 534–535
- reading contents of the file, 567–568
- reading contents of the file using getc () function, 546–547
- reading of, 535
- searching errors in reading/writing, 561–562
- sequential, 534–535
- steps for operations, 535–544
- syntax, 535
- syntax of read mode, 536
- text modes, 536
- using w+ mode for writing and reading of, 540–541
- fillexp () function, 596–597
- first C program, 16–17
- flags, 88–99
- floating, 37
- floating-point constants, 36
- floating-point representation, 44

- floodfill () function, 603–604
- floor () function, 167
- floppy disk, 534
- flowchart, 18
- fopen (), 534–536
- for loop statement, 156, 160–163
 - conditions, 146
 - counting of numbers, 161–162
 - displaying a character and its position in alphabets, 171
 - displaying even numbers, 160
 - displaying largest number, 168–169
 - displaying letters using infinite, 161
 - displaying mantissa and exponent, 171–173
 - displaying numbers from 1 to 15 using, 159
 - displaying numbers from 1 to 16 using incrementation operation, 159–160
 - displaying numbers in ascending and descending orders, 160–163, 169–170
 - displaying numbers with square roots, 164–165
 - displaying numbers with squares, 158, 164
 - displaying smallest number, 169
 - evaluation of series using, 165–166
 - execution of, 157
 - explanation, 156–157
 - expressions are, 156
 - flow chart, 157
 - functions and, 405–406
 - generating multiplication of two integers, 169–170
 - generating perfect squares, 167–168
 - generating sum and average of numbers, 171–172
 - generating sum and square of sum, 167
 - generating triangular numbers using, 166
 - infinite, 161–163
 - multiplication of two integers by using repetitive addition, 170–171
 - re-evaluation parameter, 156–157
 - syntax, 156–157
 - test condition, 157
 - various formats, 158
- formal arguments, 380
- formatted functions, 85–86
 - printf () statement, 86–88
 - return values, 85
 - scanf () statement, 86, 91–95
- FORTRAN, 6, 8
- fprint () function, 535, 541
- fputc () function, 535, 538–540, 549–550
- fread () function, 553–555, 557
- free () function, 613–618
- fscanf () function, 535
- fseek () function, 558–559
- ftell () function, 535–536
- functions
 - addition (+), 400–401
 - as an argument, 396–399
 - with arguments and return values, 387–388
 - with arguments but without return values, 388
 - assigning return value of a function to another variable, 400
 - assignment operator (=), 68–71
 - call by value, 393–395
 - comparing two return values, 404–405
 - copying of array elements using user-defined, 415–416
 - and decision statements, 406–409
 - definition, 380
 - division (/), 401–402
 - and do-while loop, 411–412
 - effect of global variables on different, 384
 - evaluating equation $s = \text{sqr}(a \text{ ()} + b \text{ ()})$ using, 405
 - evaluating equation $y = x^1 + x^2 + \dots + x^n$ using, 406
 - graphic, 591–593
 - if else statement with & without, 407
 - increment/decrement operator (++ & --), 402–403
 - initialization of array using, 412–413
 - input/output, 73–74
 - interchanging of array elements of two array using, 416–417
 - library, 437–438
 - and for loop statement, 409–410
 - and loop statements, 409–412
 - main () function, 380, 384
 - mod (%), 403
 - multiplication (*), 401–402
 - names, 383–384
 - need for, 382–383
 - operator (?), 492–493

- with operators, 399–400
- passing array elements to, 413
- passing reverse array to, 413–414
- programs using library, 593–598
- program to send a value by reference to the user-defined function, 379–380
- program to show using similar names for different, 383–384
- reading array elements declared in different functions using global pointer declaration, 418
- reading private array of other, 416
- recursive feature of. *see* recursion
- to return more values per call, 395–396
- return statement, 384–386
- return type of, 383
- and structures, 504–506
- subtraction (–), 400–401
- and switch() statements, 408–409
- types of, 386–393
- user-defined, 381–383
- void, 390
- and while loop, 410–411
- without arguments and but with return values, 386–387
- without arguments and return values, 386–387
- working of, 380–381

fun () function, 384

fwrite () function, 555–556

G

getc () function, 535–538

getche () function, 540–542

getcolor () function, 593

getdrivename () function, 597

getmaxcolor () function, 597

getmaxmode () function, 597

getmaxx () function, 592, 597, 604

getmaxy () function, 597, 604

getmodename () function, 592

getw () function, 552–553

global variables, 10, 383–384, 447, 450, 452–453

goto statement, 130–132, 161, 170

goto unconditional jump, 113

graphics

- arc () functions, 592

- bar () functions, 593
- circle () functions, 592
- closegraph () functions, 592
- drawing noncommon figures, 607–608
- ellipse () functions, 592
- filling patterns with different colours and styles, 600–604
- getcolor () functions, 593
- getmaxx () functions, 592
- graphresult () functions, 592
- initgraph () functions, 591–592
- initialization of, 591
- line () functions, 592
- mouse programming, 604–607
- outtextxy () functions, 593
- programs using library functions, 593–594
- restorecrtmode () functions, 592
- setcolor () functions, 593
- settextjustify () functions, 593
- settextstyle () functions, 593
- working with text, 598–600

graphics.h header file, 591

graphresult () functions, 592

H

Hamming code, 192–193

hash #define directive, 463–466

hash #error directive, 472–473

hash #ifndef directive, 471–472

hash #include directive, 462–463

hash #line directive, 473

hash #pragma directives, 473–475

- ANSI violations and, 474
- errors and, 474
- inline, 473
- portability warnings and, 474
- program to set off certain errors shown by, 475
- saveregs, 473

hash #undef directive, 466

header files, 17–18

header node, 626

hexadecimal code, 9

hexadecimal integer constant, 36

Hexadecimal numbers, 267–268

high-level languages, 8

homogeneous data, 221

I

- identifiers, 35–36
- if–else–if ladder statement, 113, 121–129
 - examples, 121–129
 - flow chart, 120
 - process, 121
 - rules, 123
 - syntax, 121–123
- if–else statement, 113, 116–120
 - examples, 116–119
 - flow chart, 116
 - syntax, 116
 - true and false conditions, 116
 - with & without functions, 407
- if statement, 113–114, 188
 - examples, 113–115
 - flow chart, 113
- include directory, 16
- increment operator, 65–66
- indefinite repetition, 155
- infinite algorithms, 22
- initgraph () function, 591–592, 597
- initialization of variables, 46
- input device, 3–4
- input () function, 399, 400
- input/output functions
 - formatted, 85–88
 - unformatted, 85–88
- insertion, 241–242
- integer constants, 36
- integer representation, 43–44
- interpreters, 9
- int86 () function, 510, 604–606
- int86x () function, 510
- I/O disk operation, 572
- I/O redirection, 580–582
- isalpha () check, 517
- isdigit () check, 517
- iterative process, 430–431

J

- Java compilers, 9

K

- keywords, 35

L

- laptop computer, 5
- lib directory, 14
- library functions, 10
 - commonly used, 104–105
- line () function, 592
- LINE identifier, 470
- lineto () function, 591–592
- linking, 10
- list
 - definition, 621
 - deletion from linked, 631
 - HEAD position, 621
 - insertion into linked, 630–631
 - insertion of an element in, 624–625
 - linked, 626–627
 - linked with and without header, 627–636
 - predecessor, 621, 623–624
 - program to create a linked, 632–636
 - program to create the linked list with header, 628–629
 - searching a linked, 630
 - searching and retrieving an element from, 622–623
 - singly linked, 626–627
 - successor, 621, 623–624
 - TAIL position, 621
 - traversal of, 621–622
 - traversing a linked, 630
- logical operators, 71–74
- long integer, 41–42
- loop statement, 113
 - see also do-while loop; for loop statement; nested for loops; while loop
 - in C, 155
 - definition, 155
 - and functions, 409–411
 - incrimination/decrimination, 155
 - initialization, 155
 - loop variable, 155
 - steps in, 155–156
- low-level disk I/O operations, 572–575
- low-level languages, 9

M

- machine language, 7
- macros

abort (), 480
 ANSI C predefined, 475–477
 with arguments, program on, 468
 assert (), 479–480
 DETECT, 592
 feof (), 561
 isalpha () checks, 479
 isupper () checks, 479
 for logical operators, 465
 memory model, 477–476
 predefined in ctype.h, 478–480
 program to enter text and display it using, 477–476
 program to identify whether the entered character is a letter or digit and capital or small using predefined, 478–480
 standard I/O predefined, in stdio.h, 477–476
 TURBO-C predefined, 476
 main () function, 419–422, 463, 468, 506
 malloc () function, 614–619
 math.h header file, 164
 M-BASIC, 9
 memchr () function, 316
 memcmp () function, 316–318, 367
 memcpy () function, 316
 memicmp () function, 368
 memmove () function, 316
 memory, 4–5
 allocation functions, 614–620
 allocation programs, 614–620
 models, 613–614
 memory address, 337
 memory model macros, 477
 message () functions, 387
 mnemonics, 8
 mouse programming, 604–607
 moveto () function, 597–598
 MS.DOS, 6

N

nested for loops
 adding a parity bit with binary bits, 179–180
 C compiler in sequence, 173–174
 converting a binary number to a decimal number, 186–187
 converting a binary to decimal number, 188
 count number of votes, 176

detection of Armstrong numbers, 182–183
 displaying octal numbers in binary, 179–180
 displaying results of students, 193–194
 displaying series of numbers, 195
 display numbers using ASCII values, 175
 generating pattern of numbers, 184–185
 generating pyramid structure using numerical, 186
 inner loop iterations, 173
 perfect cubes, 165–166
 printf () statement using, 174–175
 program to accept a number and find sum of its individual digits, 178–179
 program to count occurrence of digits from 0 to 9, 175–178
 program to display the stars, 183–184
 program to simulate a digital clock, 176–177
 subtraction of two loop variables, 172–173
 nested if-else statement, 120–121
 flow chart, 120
 rules, 121
 syntax, 120–121
 nested switch statements, 144–145
 non-deterministic algorithm, 22
 NOT operator (!), 71, 73
 NULL character ('0'), 228, 287–291, 293
 numerical constants, 37–38

O

object codes, 9
 object program, 8
 open () function, 580
 operand, 59
 operators
 '&', 337, 342–343
 arithmetic, 64–67
 assignment, 68–71
 bitwise, 74–79
 comma, 63–64
 operators, 16, 35
 conditional, 63–64
 logical, 71–74
 priority of, 59, 649–650
 relational, 67–68
 types of, 59
 OR Gate, 188–189
 OR (||) operator, 71, 73

other () function, 396
 output device, 4
 outtextxy () functions, 593

P

p [arr], 354
 *(p+arr), 354
 Pascal, 6
 Pentium, 5
 period (.) sign, 447, 493
 perror () function, 563–564
 PI (identifier), 463–464
 pointers
 addition using, 351–352
 and address, 338
 to add two numbers through variables using, 343–344
 arithmetic operations with, 348–352
 array of, 360–362
 assigning of, 343–344
 comparison of two, 373
 constant, 347–348
 to copy the elements of one array to another, 356
 declaration, 339
 definition, 337
 to display array elements and their addresses, 367–368
 displaying the addresses of different variables together with their values, 340–341
 to display the address of a variable, 339, 363
 to display the data type that occupies the same space, 341
 to display the string using, 373
 to display the sum of squares and cubes of array elements, 355–356
 to display the value of variables and its location using, 341–342
 division using, 373
 effect of increment and decrement operators used as prefix and suffix, 349–350
 effect of increment on, 348–349
 features, 338
 file, 558
 multiplication using, 373
 to pointers, 361, 367
 to print an element and its address using, 343

 to print value of a variable, 343
 to print variables using ‘*’ and ‘&’ operators, 342–343
 program to access elements of an array using, 354
 program to interchange elements of character array using, 365
 to read two strings through the keyboard, 366, 369–370
 and strings, 364–370
 subtraction using, 351
 sum all the elements of an array using, 355
 and two-dimensional arrays, 357–358
 void, 345–346
 wild, 346–347
 pow () function, 165, 174–177
 precision specifier, 89
 preprocessor, definition, 463
 preprocessor directives
 conditional compilation, 469–471
 hash #pragma directives, 473–475
 identifier for C statements and variables, 464–465
 identifier for displaying double and triple of a number, 465–466
 macros for logical operators, 465
 standard I/O predefined macros in stdio.h, 475–477
 undefining a macro, 466
 printf () function, 74, 96, 118–130, 425–428
 length of the string using, 97–98
 printf () statement, 20, 96, 118–130, 425–428
 addition of two numbers, 98
 average of three real numbers, 93–94
 and conversion symbol, 86–88
 for displaying single character, 98–99
 escape sequences, 95–96
 examples, 95
 flags, 94
 format string, 94
 precision specifier, 89–90
 square of the given number, 98–99
 for swapping values of two variables, 99–100
 use of ‘*’ for formatting, 85
 using nested for loops, 173–174
 width specifier, 88–89
 problem-solving techniques

- algorithm, 17–22
- flowchart, 22–26
- pseudo-code, 26–28
- procedure-oriented languages, 7
 - see high-level languages, 5, 6
- programming language, evolution of, 5–6
- programming rules, 12
- pseudo-code, 26–28
- putc () function, 547–548
- putw () function, 551–552

R

- RAM (random access memory), 4
- rand () function, 437
- random algorithm, 22
- read () declaration, 574–575
- read () function, 572
- read only memory (ROM), 4
- read–write memory, 5
 - see user’s memory, 5
- real constants, 37–38
- realloc () function, 619–620
- recursion
 - advantages, 435–436
 - calculation of triangular number of an entered number with, 414
 - definition, 423–424
 - depth of, 431–432
 - direct, 427–428
 - disadvantages, 435–436
 - efficiency, 435–436
 - essential conditions for, 425
 - indirect, 428–430
 - program to call the main () function recursively, 423
 - rules, 425–426
 - steps, 423
 - terminating statement in, 425
 - types, 424–425
 - vs iteration, 430–431
- register class, 455
- register variables, 454–455
- relational operator, 67–68
- remove () function, 566
- rename () function, 566–567
- restorecrtrmode () functions, 592
- return statement, 384–386

- return () statement, 380
- rewind () function, 540–541
- Ritchie, Dennis, 6

S

- scanf () statement
 - addition of two numbers, 98–99
 - average of three real numbers, 93–94
 - detecting an error using, 93
 - with different formats, 93
 - for displaying inputted name, 97–98
 - for displaying single character, 98
 - effect of mismatch of data types, 92
 - escape sequences, 95
 - for reading and printing integer value, 92
 - square of the given number, 98
 - use of comma, 93
- searching, 240
- segment registers, 519
- semi-colon (;), 156, 160–161
- semiconductor memory, 4
- sentinel-controlled repetition, 155
- sequential execution, 113
- setcolor () functions, 593
- setfillstyle () function, 600–603
- settextjustify () functions, 593
- settextstyle () functions, 593
- setviewport () function, 604
- short integer, 41
- show () function, 429–430
- signed char, 42
- signed integer, 41, 43
- signed magnitude form, 43
- single character constant, 38
- sizeof () function, 227–228
- sleep () function, 104–105
- source code, 9
- source program, 8–9
- source stream, 534
- sprintf () function, 274–275
- sqrt () function, 118
- square () function, 396
- sscanf () function, 274–276
- stacks, 424
- standard directories, 18
- static int, 205
- static variables, 453–454

- stderr, 246
- stdin, 246
- stdio.h header file, 85, 478
- stdout, 246
- storage class, 437
- strcat () function, 292, 308–309
- strchr () function, 292, 305–306
- strempt () function, 291, 301–303
- strcpy () function, 291, 297
- strdup () function, 292, 305
- stricmp () function, 291
- stricmp () function, 300
- string constants, 36
- stringizing operation, 467–468
- strings,
 - applications of, 318–325
 - comparing the source with destination, 32–333
 - concatenation of two, 308–310
 - conversion functions, 315–316
 - copying the contents of one to another, 298–299
 - copying the contents of one to another by using strcpy (), 297–298
 - copying the source to destination, 298–299
 - counting the number of characters in a, 289–290
 - with different formats, 289–291
 - displaying names of cities alphabetically, 322–324
 - displaying successive, 288
 - displaying the length of the entered string, 296–297
 - displaying using various printf () format specifications, 290
 - displaying using while loop, 290–291
 - equivalent ASCII codes of, 293
 - finding no. of characters in a given string including and excluding spaces, 295–296, 365
 - memory functions, 316–318
 - and NULL character, 228, 287
 - palindrome, 321–322
 - printing a given string from the first occurrence of the given character, 314–315
 - removing the occurrences of ‘The’ from text, 294
 - removing the occurrences of vowels in a given text, 294
 - sorting of names, 250
 - standard library functions of, 291–292
 - syntax for, 287
 - using the strcmp () function, 300
- strlen () function, 291–293
- strlwr () function, 292, 304
- strncat () function, 292, 310
- strncmp () function, 292, 302
- strncpy () function, 291
- strnicmp () function, 292
- strnset () function, 292, 312–313
- strpbrk () function, 292, 314
- strrchr () function, 292
- strrev () function, 292, 310–311
- strset () function, 292, 312
- strspn () function, 292, 313
- strstr () function, 292, 307–308
- strtod () function, 315
- structures, of C program
 - array of, 498–501
 - body of the function, 11
 - comments, 11–12
 - copying elements from one object to another object, 494–495
 - creating a user-defined data type from, 50–507
 - declaration and initialization of, 492–495
 - declaration part, 11
 - defining, containing the details of the employee, 509–510
 - executable part, 11
 - features, 491
 - global declaration, 10
 - header files for function, 10
 - main () function, 10–11
 - memory allocation, 492
 - nested, 497–498
 - pointer to, 501–504
 - printing elements using print () function, 505
 - program to display the size of, 493
 - read and write, 553–557
 - reading the values using scanf (), 495
 - string data type, 507
 - within structure, 496–498
 - understanding the working of, 493–494
 - union of, 524
 - user-defined functions, 11

- variable to a user-defined function, 504–505
- strupr () function, 292, 304–305
- switch case statement, 113, 132–143
 - examples, 133–143
 - flow chart, 134
 - switch execution, 133
 - switch expression, 133
 - switch organization, 133
 - syntax, 133
- switch () statement, 40–408
- syntax
 - array, 222–223
 - of close () function, 573
 - do-while loop, 207–212
 - for fgetc (), 536
 - files, 536
 - fscanf () function, 535
 - if–else–if ladder statement, 121–129
 - if–else statement, 116–120
 - for loop statement, 160
 - nested if–else statement, 120–121
 - of read mode, 537
 - strings, 33, 35
 - switch case statement, 132–133
 - while loop, 194–212
- sys directory, 14
- system-programming language, 7, 28

T

- testing programs, 21
- tokens, 35
- tower of Hanoi, 432–435
- transfer of control, 113
- translators, 8–9
- traversing, 238
- trinum () function, 427
- tri_num () function, 423
- Turbo-c editor, 16
- TURBO-C predefined macros, 476–477
- turbo-C version 2 support programs, 7
- type conversion, 47–49
- typedef statement, 507, 509
- type modifiers, 46–47

U

- uabs () function, 398
- unary minus, 61

- unary operators, 61
- unary plus, 61
- understanding of programs, 106
- union, 49–493
 - of structures, 524
- UNIX operating system, 6–7
- unlink () function, 535
- unsigned char, 40
- unsigned integer, 39
- user's memory, 4

V

- variable name, 35
- variables, 447
 - automatic, 448–449
 - definition, 35
 - environment, 579–580
 - external/global, 447, 451–452
 - internal, 447
 - variables, 10, 18
 - lifetime of, 447–448
 - program to declare and initialize, 39
 - rules for defining, 39–40
 - scope, 447
 - static, 447
 - storage class of, 447
 - visibility, 448
- video display unit, 591
- volatile variables, 50–51

W

- while loop, 194–207
 - adding of consecutive numbers, 195–196
 - calculating factorial of a given number, 196–197
 - converting binary number to equivalent decimal number, 198–199
 - converting decimal number to binary number, 197–198
 - converting decimal number to user-defined number system, 197–198
 - counting of positive and negative numbers, 200–201
 - displaying ASCII numbers and their equivalent character, numbers and symbols, 219
 - displaying 'sum' of all digits, 200

- execution of, 194
- flow chart, 194
- functions and, 410–411
- to print an entered number in reversed order, 202–203
- program to convert entered character into its opposite case, 230–231
- program to enter a statement using a combination of capital, small, symbols and numerical, 203–204
- program to print a string, 194–195
- program to sort numbers 0 to 9, 204–205
- program to use three, 206–207
- reading positive integer number ‘n,’ 199
- sorting odd and even numbers, 201–202
- syntax, 194
- while statement, 149
- width specifier, 88
- wrapping around, 49–50
 - program demonstrating, 50
- write () function, 572

THIRD EDITION

Programming in



ASHOK N. KAMTHANE

AMIT A. KAMTHANE

C is one of the most popular programming languages. It runs on most software platforms and computer architecture. This revised edition of our best-selling text *Programming in C* not only maintains the exclusivity of previous editions but also enhances it with the addition of new programs and illustrations.

Challenging concepts are supported with numerous solved and unsolved programs. The new chapter on computer graphics ensures that this book comprehensively covers the syllabi of most universities. The book also uses the Turbo C compiler, which is the most widely used C compiler. With its increased coverage and inclusion of new learning tools, this edition is an invaluable asset for students who aim to improve their programming skills.

FEATURES

- Enhanced pedagogy
 - ▶ New flowcharts and diagrams
 - ▶ More than 100 new programs
- Fully tested and executed programs
- New chapter on Computer Graphics



Online Instructor resources available at
www.pearsoned.co.in/ashoknkamthane



Cover image: echo3005. Shutterstock

ISBN 978-93-325-4355-3



9 789332 154355

www.pearson.co.in