Programming and Problem Solving Using C

Programming and Problem Solving Using C

International Software Research and Development
(ISRD Group)
Lucknow

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices

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

Tata McGraw-Hill

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008.

ISBN 13: 978-0-07-066760-0
ISBN 10: 0-07-066760-8

Managing Director: *Ajay Shukla*

General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*
Asst. Sponsoring Editor: *Shalini Jha*
Editorial Executive: *Nilanjan Chakravarty*
Executive—Editorial Services: *Sohini Mukherjee*
Senior Production Manager: *P L Pandita*

General Manager: Marketing—Higher Education & School: *Michael J Cruz*
Product Manager: SEM & Tech Ed: *Biju Ganesan*

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

Published by the Tata McGraw-Hill Publishing Company Limited, 7 West Patel Nagar, New Delhi 110 008, typeset at Bukprint India, B-180A, Guru Nanak Pura, Laxmi Nagar, Delhi - 110 092 and printed at
Avon Printers, 271, FIE, Patpar Ganj, Delhi 110 092

Cover: Rashtriya

RADYYRDXRCDBA

The ISRD Group

The Instructional Software Research and Development (ISRD) Group is committed to develop and produce high quality learning and instructional material to facilitate all-round professional development of students. The Group continuously strives to enrich their content repository by constantly researching and developing new learning and instructional material. The content thus developed is tailored to take various forms including textbooks, teaching aids, lecture notes, lab assignments and exercises, both in print as well as in electronic media (CDs, etc.).

This particular book has taken its final shape with the noteworthy contributions of

Ms. Roli Srivastava She holds a masters in Computer Applications (MCA) degree. Presently, she is working with the ISRD Group.

Mr. Tamal Chatterjee He has done the 'B' Level Programme from DOEACC and is currently associated with the ISRD Group.

The work would not have been possible without the enriching guidance and support of the following key professionals:

Er. Alok Singhal is an Electronics and Telecommunication Engineer from Roorkee University (now IIT Roorkee), with a master's degree in Systems Engineering. Over 35 years of experience in the areas of entertainment electronics, microcomputers and microprocessor-based products. In his last assignment he was General Manager and Head of Communication Division at UPTRON India Ltd and was handling manufacturing and marketing of EPABX, RAX, radio communication, office communication, data communication, and computer products. He presently holds the position of Vice President at UPTEC.

Prof. K. K. Bhutani holds a bachelor's, master's and doctoral degrees from the University of Allahabad, he has more than 40 years of experience in academics and industry. He has been a professor of Electronics and Head of Computer Centre, University of Allahabad. He has also worked as General Manager (Computers), Computronics India. He is former Chairman, Computer Engineering Board, Institution of Engineers (India). At present, he is associated with UPTEC as a Director.

Er. K. N. Shukla is an Electronics Engineer from IIT, Kanpur. He has more than 35 years of experience in electronics and computer industry. Started his career with J. K. Electronics, Kanpur, and has held several managerial positions at UPTRON, HILTRON and other similar companies. He was Director (Technical) at UPTRON India Limited before joining UPTEC as full-time Director.

Dr. R. K. Jaiswal holds a doctoral degree from the University of Lucknow and has more than 20 years of experience in academics. He started his career with Lucknow University as a Lecturer, and has

worked under College Science Improvement Programme (COSIP), a Government of India scheme. He has more than a dozen research papers in national and international journals to his credit. He presently holds the position of Deputy General Manager at UPTEC.

Er. R. K. Lele holds a bachelor's degree in Electrical Engineering from IIT Kanpur. He has Design and Development experience in embedded software/firmware, system software, computer and telecommunication/data communication hardware and exposure to real-time software in the area of switching and communication protocols. At the managerial position, he has handled various design projects with engineers during all the phases of the product design cycle. He presently holds the position of General Manager (Systems) at UPTEC.

Prof. S. K. Singh holds a bachelor's degree from IIT Kharagpur, master's degree from University of Missouri (USA) and with more than 35 years of professional experience in Bell Systems (USA), IBM, National Institute for Training in Industrial Engineering (NITIE), 10 years as Professor of Computers and MIS at Indian Institute of Management (IIM), Lucknow, has contributed and guided extensively to develop UPTEC series of 'A' level textbooks.

Dr. Upendra Kumar is an Industrial Engineer, and has obtained his Ph.D from IIT, Delhi. Has more than 30 years of experience in industry, academics and research. Dr. Kumar has worked with BHEL Corporate Systems Group, Eicher, UPTRON, National Institute for Training in Industrial Engineering (NITIE) and Indian Institute of Management (IIM), Lucknow. Dr. Kumar had conceived, created and established UPTRON-ACL and Computer Consultancy and Services Division at UPTRON earlier, as General Manager and Divisional Incharge. Has been consultant to many national/international organizations and is presently working as the Managing Director of UPTEC Computer Consultancy Ltd.

Mr. Kirti Kumar Pant He has more than 20 years of experience in academics and has been developing instructional material for UPTEC. He holds a master's degree in Physics and post-graduate diploma in Business Administration (Human Resource Management). At present, he is heading Instructional Software Research and Development (ISRD) Group at UPTEC.

We are also indebted to **Mr. Surya Prakash Sharma**, **Mr. Vikash Bajpayee** and **Mr. Rajan Awasthi** for their help with the DTP work—including data entry, formatting, making corrections, drawing and capturing figures, creating tables, and preparing table of contents and indexes.

Contents at a Glance

Contents

Preface

It gives us great pleasure to introduce this book on programming and problem solving using C. When it comes to choosing programming languages, C has always been the favourite choice of programmers because of it's flexibility, portability and structure, apart from having in itself the features of a high-level language and an assembler. This book provides a graded and step-by-step approach in creating and developing programs in C.

This book primarily caters to diploma students of the computer science stream and students taking the DOEACC 'O' level certificate examination. We have taken care to use simple and lucid language and the pedagogical features have been specially designed to fit into the DOEACC 'O' level examination mould. Students of BCA, MCA, B Sc (IT), M Sc (IT) can also use it as a handy reference.

The salient features of the book are the following:

- Overview of programming has been covered followed by fundamental and advanced programming techniques
- Comprehensive coverage of advanced topics like C Preprocessor, including discussions on Macro Substitution, File Inclusion, and Conditional Compilation Directives
- An exhaustive collection of practice problems which include 105 Multiple-Choice Questions, 90 True/False statements, 102 Fill in the Blanks, and 103 Descriptive Questions

The book is divided into seventeen chapters. Chapter 1 gives an overview of C and introduces algorithms. Chapter 2 takes the student a bit further and discusses algorithm logic, programming and system design techniques. Chapter 3 is on program design and implementation issues, while computer programming languages are covered in Chapter 4. Reasons for the popularity of C are explained in Chapter 5. Chapters 6 and 7 cover variables and type modifiers respectively. Chapter 8 covers input/output functions, while Chapter 9 is on loops.

Chapters 10, 11 and 12 are on arrays, functions and pointers. Structures and unions are dealt with respectively in Chapters 13 and 14. Chapter 15 explains linked lists and dynamic memory allocation. Chapter 16 is on file handling in C, and finally Chapter 17 covers the C preprocessor.

Appendix I discusses standard header files and library functions while Appendix II is on ASCII values of characters.

Question papers of previous years have been included in the end, which will prove advantageous to students in helping them prepare for examinations.

We would like to thank all those who helped us directly or indirectly with the preparation of this book. First of all, we would like to thank the reviewers, Mr. Tapas Adhikari of Dinabandhu Andrews College, Kolkata, and Ms Shanali Walia of DICS Computer Education, New Delhi, for their valuable suggestions and feedback, which went a long way in shaping the contents of this book. Our thanks are also due to the team at Tata McGraw-Hill for their cooperation in bringing out this book on time. Finally, we would like to give a special thanks to all our friends, students, colleagues and respective family members whose support, patience and encouragement is highly cherished by us.

We hope this book satisfies its readers and proves useful to them. Constructive feedback and suggestions for improvement and will always be welcomed.

ISRD Group

CHAPTER

1

Introduction to Computer-based Problem Solving

Key Features

- Problem Definition
- Problem Solving
- Goals and Objectives
- Problem Identification and Definition

Software development is a sequentially interrelated process. Multiple steps are performed before developing a new software. The first step is to determine the information needs of the proposed software. Problem definition is the most important step in this direction. Among some of the general characteristics like speed of processing, volume of work, reliability, and accuracy, there are some other functions that are expected from the software and they vary from application to application.

In this chapter, we are going to discuss in detail about problem definition, problem solving, problem identification, and various similarities between the problems.

PROBLEM DEFINITION

Problem definition is the first and most important step in determining information needs. Information needs of a software can be determined by:

- Studying the existing system
- Using questionnaires
- Stating the tentative information needs
- Suggesting interpretation of information needs with users

First of all, using any of these steps, the requirements are determined. Analysis of the problem produces a clear and accurate problem definition.

PROBLEM SOLVING

Problems are the undesirable situations that prevent any software from fully achieving its objectives. When we wish to transform the existing situation into a more desired one, a problem occurs, and a need to solve it arises. A problem clearly defined in terms of goals and objectives helps largely in problem-solving. Steps involved in problem-solving methodology are shown in the Figure 1.1 ahead.

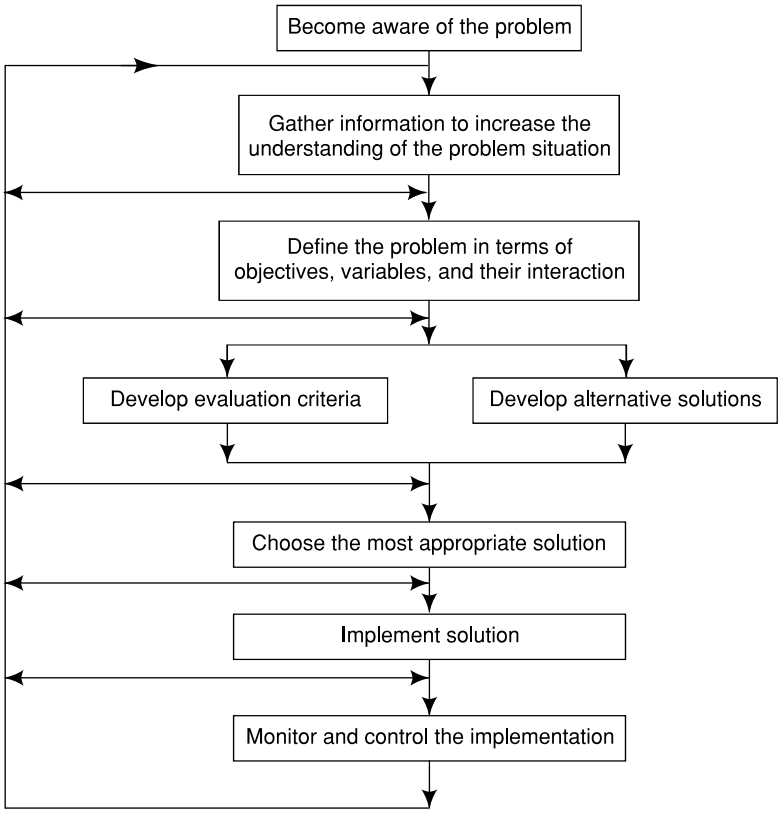


Fig. 1.1 *Steps in problem solving*

GOALS AND OBJECTIVES

All the systems, by definition, must have a purpose. For a business organization, the purpose is to accomplish meaningful goals and objectives. For a software, the goal is to solve the need of the user desired from that software.

There are three related terms that need to be well understood.

- Mission — It is the broad statement of the purpose of the organization.
  Example: "To grow continuously through provision of quality products and services."
- Goal — It is the general statement about what is to be accomplished.
  Example: "Reduce time to respond to a service request without increasing the number of service persons."
- Objective — It is the statement of measurable results to be achieved in a specified time frame.
  Example: "Reduce order processing time from 5 to 3 days".

Business plans are derived from business objectives. Solving business problems, setting business goals and objectives and developing the plans require a diverse range of internal and external information. It is the responsibility of an information system analyst to provide the required information.

PROBLEM IDENTIFICATION AND DEFINITION

The following situations can cause a problem to arise:

- A crisis which, unless tracked, will affect the organization adversely.
  Example:
  Strike, fire, and breakdown
- A change in the environment to which the organization must respond.
  Example:
  Competitor, and reducing prices
- Changes planned to increase the efficiency of operations.

Information has to be provided to the decision makers to make them aware that a problem has arisen which requires solution. Next, the problem has to be defined more concretely and precisely before an appropriate solution can be developed.

Problem definition involves:

- Identifying key persons involved
- Identifying key variables and their interrelationships
- Specifying the objective(s) of the solution

Additional information will be required for problem definition.

Vagueness in Information Requirement Specification

Computer-based information systems are designed to meet the information requirements of users—managers and administrators. Therefore, clear and unambiguous understanding of the information needs of various users is the critical first step. Any vagueness in understanding of information needs is bound to lead to failure.

In trying to develop an understanding of user requirements, analysts face a number of problems, just because the requirements are not spelled out in clear, unambiguous terms. The user's need for immediate access to data emerges over the whole period of study and system analysis, as part of the interactive process of gathering information.

There can be two groups of users in an organization:

First Group : Unaware of advantages of immediate access to information through computers.
Second Group : Well aware of advantages of computers.

The first group never expresses the need for immediate access to data as they have acclimatized to functioning with the reports generated by a batch system or a manual system. Hence, the analyst has to make the group aware of the current technical possibilities and the realization of how immediate access to relevant information could be of value to them. On the other hand, the managers in the second group present the analyst with a collaborate "wish list", often unrealistic. The analyst is faced with the problem of encouraging the first group of managers to use the computer more effectively and, at the same time, avoiding promising anything to the second group that cannot be delivered cost-effectively.

Similarities between Problems

Certain areas of concern which are common to different types of problems are:

- Speed of Processing
- Volume of Work
- Reliability
- Accuracy
- Security
- Cost

Speed of Processing The speed at which computer-based data processing systems can respond adds to their value. Speed of transaction processing is a critical factor in any modern business concern.

Example:

The train reservation system would not be useful if clients had to wait for more than a few minutes for a response. The response required from a computer controlled manufacturing process might be a fraction of a second.

Volume of Work The volume of information to be handled at each location should be calculated, both for normal and peak hours to avoid bottlenecks in the operation. The steps in this analysis are:

- Calculate the number of messages flowing to and from every point in the system.
- Calculate the average characters per message.
- Calculate the total transmission time using the data from the two points above, plus the transmission speed of the type of communication being analyzed.
- Calculate the volumes for the peak periods.

The analysis of urgency requirements may result in procedures to transmit information that have a low urgency during periods of low volume, thereby increasing the capacity available for peak periods.

Reliability This includes:

- The robustness of the design
- Availability of alternative computing facilities in the event of a breakdown
- The provision of sufficient equipment and staff to handle peak loads

Accuracy This determines how accurately the data is being processed by the machine.

Security There are many aspects of security; those which particularly concern the systems analyst are:

- Confidentiality
- Privacy
- Security of data

Confidentiality of certain information is vital to the success of a firm. Therefore, the system has to ensure that only authorized staff have access to such information. The design of the system must guard against any unauthorized access by the individual employees or members of the employee's family. Measures should be taken to guard against unauthorized alteration or destruction of data.

Cost Cost is associated with the activities of development as well as operation of the data processing system. Development comprises all the stages from the initial investigation to the successful hand-over of a final system to the users. Operation includes evaluation and maintenance of the system. Real cost of processing includes five different categories of possible costs:

- Manpower
- Hardware
- Software
- Consumables and Supplies
- Overhead Costs

Summary

- Problem definition is the first and most important step to solve the problem.
- A problem is an undesirable situation that prevents the organization from achieving the goals and objectives.
- Before solving the problem, all the information needs, goals, and objectives, need to be clearly defined.
- Information requirements may vary from user to user and situation to situation.
- There are some areas of similarities between the problems.

Review Exercise

Multiple-Choice Questions

1. Problem definition involves
   (a) identifying key persons involved
   (b) identifying key variables
   (c) specifying the objectives of the solutions
   (d) all of the above
2. Computer based information systems are designed to meet the information requirements of
   (a) managers
   (b) administrators
   (c) both (a) and (b)
   (d) none of the above
3. Real cost of processing includes this category of possible costs
   (a) consumables and supplies
   (b) security
   (c) privacy
   (d) confidentiality
4. Aspects of security which are concerned with the system analyst are
   (a) confidentiality
   (b) security
   (c) privacy
   (d) all of the above

5. Which of the following includes the provision of sufficient equipment and staff to handle peak loads?
   (a) volume of Work
   (b) reliability
   (c) speed of processing
   (d) accuracy

State whether True or False

1. Cost is associated with the activities of development as well as operation of the data processing system.
2. Speed of transaction processing is not a critical factor in any business concern.
3. Mission is the statement of measurable results to be achieved in a specified time frame.
4. Goal is the broad statement of the purpose of the organization.
5. Objective is the general statement about what is to be accomplished.
6. Confidentiality of certain information is vital to the success of a firm.
7. System design must guard against any unauthorized access.

Fill in the Blanks

1. Problems prevent the organization from _____ .
2. Clear statement of _____ and _____ is a vital step towards problem solving.
3. It is the responsibility of a system analyst to _____ the information system.
4. _____ is a critical factor in any modern business concern.
5. _____ determines how accurately the data is being processed by the machine.
6. Operation includes _____ and _____ of the system.

Descriptive Questions

1. What are the steps involved in problem solving?
2. In how many ways can the users be classified?
3. Discuss the similarities between problems.
4. What are the responsibilities of the systems analyst?
5. Discuss the main aspects of security that are of concern to the systems analyst.
6. Why do we learn programming?
7. What is programming language?

CHAPTER

2

Algorithms for Problem Solving

Key Features

- Algorithms—An Introduction
- Properties of an Algorithm
- Classification
- Algorithm Logic
- Examples

Computers can neither think nor make decisions on their own. They require instructions to perform each and every task. They, therefore, depend entirely on human beings for instructions to solve any problem. Hence, before any problem can be solved using a computer, the person writing the program must familiarize himself with the problem and with the way in which it has to be solved. Before coding the problem in a computer language, it is better to represent the problem in small and clear steps by using algorithms.

This chapter gives an insight into the construction and use of algorithms for different types of problems.

ALGORITHMS—AN INTRODUCTION

The concept of an algorithm is one of the basic concepts in mathematics. An algorithm is a finite set of rules, which gives a sequence of operations to solve a specific problem. The aforesaid definition merely reflects the concept of an algorithm.

The word "algorithm" originates from the Arabic word *algorism* which has been derived from the name of a famous Arabic mathematician Abu Jajar Mohammed Ibn Musa Al Khwarizmi (AD 825) who was the first to suggest a mechanical method to add two numbers represented in the Hindu numeral system. This method requires finding the sum of one digit from each of the operands and a previous carry digit, and repetitively carrying out this sequence of operations, from the least significant digit to the most significant end, until both the operand digits are exhausted.

PROPERTIES OF AN ALGORITHM

An algorithm, as has been discussed, is a step-by-step problem-solving procedure that can be carried out by a computer. The essential properties of an algorithm are:

- **Finiteness:** An algorithm should terminate after a finite number of steps, that is, when it is mechanically executed, it should come to a stop after executing a finite number of assignment, decision and repetitive steps.
- **Definiteness:** An algorithm should be simple. Each step of the algorithm should be precisely defined, that is, the steps must be unambiguous so that the computer understands them properly.
- **Generality:** An algorithm should be complete in itself, that is, it should be able to solve all problems of a particular type (for any input data) for which it is designed.
- **Effectiveness:** All the operations used in the algorithm should be basic and capable of being performed mechanically, that is, without any intuitive step. It should be universal and lead to a unique solution of the problem.
- **Input-Output:** An algorithm should take certain precise inputs, or initial data, and the outputs should be generated in the intermediate as well as the final steps of the algorithm. It should also have the capability to handle some unexpected situations which may arise during the solution of a particular problem (division by zero).

Algorithms can be written for any type of problem: Business, scientific, or industrial. Algorithms for scientific and engineering problems involve the use of mathematical formulae. However, algorithms for commercial problems involve very little mathematics and are mostly descriptive.

CLASSIFICATION

Algorithms can be classified according to the manner in which the control shifts from one step to another and also the way in which the repetitive steps are executed.

Deterministic, Non-Deterministic, and Random Algorithms

In an algorithm, if the control logic encounters a decision box with two routes — for yes and for no — after execution of each step, we call it **deterministic**.

Example: Computing the Greatest Common Divisor (GCD)

If, the algorithm is capable of exploring a large number of alternatives simultaneously to reach out to a correct solution, we call it **non-deterministic**.

Example: Searching the minima of a function

If after executing some step, the control logic shifts to another step of the algorithm as dictated by a random device (coin tossing), we call it **random**.

Example: Random search

Direct and Indirect Algorithms

The nature of repetition in algorithms further classifies them as direct and indirect. In direct algorithms, the number of repetitions is known in advance.

Examples:

- Evaluating a polynomial of a specified degree
- Arranging n numbers in a decreasing or an increasing order

In indirect algorithms, the repetitive steps are usually executed a number of times, the number not being known beforehand.

Examples:
- Testing whether a number is prime
- Finding the GCD of two integers

Infinite Algorithms

In some problems, the longer the algorithm continues, the better the estimates of the results. In this case, we must have a measure of the discrepancy, or error, of any particular estimate. If X_i is the i^{th} estimate and X is the true value, then the algorithm should compute $|X_i - X|$ for $i = 1, 2, ...$ If we let $\in$ denote the closeness we seek, our requirements are met by saying $|X_i - X| \leq \in$. Also, in general, since the true value of X is not known, we usually check whether $|X_{i+1} - X_i| \leq \in$ to test for convergence of the result. An algorithm which is based on such convergence tests is called an **infinite algorithm**.

Example: Computing the zeros of a polynomial by the Newton method or other iterative methods.

ALGORITHM LOGIC

Program analysts have found that algorithms developed using three basic components are easier to follow. These three components which are considered as standard units to control the flow of information processing and are used to construct algorithms are the following:

Sequential Flow

In a sequential component, steps are taken in an explicitly prescribed sequence. For example, the payroll problem. The hourly pay rate R, number of hours worked H, and total deductions D for one employee are provided. The gross pay G and take home pay P are to be calculated, a record made, and a cheque printed.

INPUT R, H, D

$G = R * H$

$P = G - D$

Update record

Print cheque

Conditional Flow

If an algorithm involves a decision to be made based on a condition, the flow is said to be conditional. To make a decision, a condition is tested. If the condition is true, then one path is taken; if false, then the other. For example, in the payroll algorithm, theoretically suppose it is possible for the gross pay to be less than deductions then, no cheque is to be printed but records are to be maintained.

INPUT R, H, D

$G = R * H$

$P = G - D$

If *P* is positive,

then print a cheque.

Update record.

Repetitive Flow

The other basic variation in algorithm flow deals with repetition of steps. In this case, if the condition is true, then the steps in the procedure are taken, after which the flow is back to the condition for a possible repetition.

As long as the condition remains true, this path is followed, so it is obvious that some step within the procedure must affect the condition.

The path which is repeated is called a *loop*. When the condition becomes false, looping ends and the algorithm moves onward. For example, in the payroll algorithm, one employee was considered. Suppose the company has *R, H, D* information on file for its entire work force.

Input the first *R, H, D*

While *R* is not zero

execute the PROCESS,

input the next *R, H, D,*

return to test *R* again.

We shall explain the process of setting up of algorithms by taking some specific examples.

EXAMPLES

Example 1: Addition and multiplication of two numbers

Step 1 : Input values for *A* and *B*
Step 2 : Add *B* to *A* and store in SUM
Step 3 : Multiply *B* with *A* and store in MUL
Step 4 : Display value of SUM and MUL
Step 5 : End.

Example 2: Check for even or odd numbers

Step 1 : Accept value in variable NUM
Step 2 : Divide NUM by 2 and store remainder in REM
Step 3 : If REM is zero
Display "Number is even"
else
Display "Number is odd"
Step 4 : End

Example 3: Swapping two variables with the help of the third variable

Step 1 : Input values in A and B
Step 2 : Let variable $T = A$
Step 3 : Let $A = B$
Step 4 : Let $B = T$
Step 5 : Display A and B
Step 6 : End

Example 4: Swapping two variables without using the third variable

Step 1 : Input values in A and B
Step 2 : Let $A = A + B$
Step 3 : Let $B = A - B$
Step 4 : Let $A = A - B$
Step 5 : Display A and B
Step 6 : End

Example 5: To calculate the percentage of marks obtained by a student in an examination, maximum marks and marks obtained are given. The required result is percentage of marks and the formula used is:

$$\% \text{ of marks} = \frac{\text{marks obtained}}{\text{max. marks}} * 100$$

Step 1 : Read name, marks obtained and max. marks
Step 2 : Divide marks obtained by max. marks and store it in Per.
Step 3 : Multiply Per. by 100 to get percentage
Step 4 : Write name and percentage
Step 5 : End

Example 6: Check for prime number

Step 1 : Input a value in NUM
Step 2 : If NUM is 0, 1, 2 or 3 go to step 8
Step 3 : Let variable E = NUM/2 and $I = 2$
Step 4 : Store remainder of NUM/I in REM
Step 5 : If REM = 0 go to step 9
Step 6 : $I = I + 1$
Step 7 : If $I <= E$ go to step 4
Step 8 : Display "Number is prime": go to step 10
Step 9 : Display "Number is not prime"
Step 10 : End

Example 7: Summation of a set of numbers

Step 1 : Input total number of variables, for summation in N
Step 2 : Let Sum = 0, $I = 1$
Step 3 : Input a number in NUM

Step 4 : SUM = SUM + NUM
Step 5 : $I = I + 1$
Step 6 : If $I <= N$ go to step 3
Step 7 : Display SUM
Step 8 : End

Example 8: Display even numbers from 0-50
Step 1 : Let $I = 0$
Step 2 : If remainder of $I/2$ is not zero go to step 4
Step 3 : Display I: go to step 6
Step 4 : $I = I + 1$
Step 5 : If $I <= 50$ go to step 2
Step 6 : End

Example 9: Factorial computation
Step 1 : Input value in N
Step 2 : Let FACT = 1
Step 3 : FACT = FACT * N
Step 4 : $N = N - 1$
Step 5 : If $N >= 1$ go to step 3
Step 6 : Display value of FACT
Step 7 : End

Example 10: Fibonacci series generation
Step 1 : Input the last term of series in N
Step 2 : Let $A = 0$, $B = 1$ and $I = 1$
Step 3 : Display A and B
Step 4 : Let $C = A + B$
Step 5 : Display C
Step 6 : Let $A = B$ and $B = C$
Step 7 : $I = I + 1$
Step 8 : If $I <= N$ go to step 4
Step 9 : End

Example 11: Reversing digits of an integer
Step 1 : Input a value in NUM
Step 2 : Let REV = 0,
Step 3 : Store remainder of NUM/10 in Rem
Step 4 : REV = REV * 10 + REM
Step 5 : NUM = (integer) NUM/10
Step 6 : If NUM > 0 go to step 3
Step 7 : Display REV
Step 8 : End

Example 12: Algorithm to arrange a list of numbers in ascending order

Step 1 : Read in a value of N, that is, total no. of numbers in the list
Step 2 : Assign 1 to I
Step 3 : Read in a number in $A(I)$.
Step 4 : Is I equal to N?
(a) Yes : go to step 5
(b) No : increase I by 1
go to step 3
Step 5 : Assign 1 to I
Step 6 : Assign 1 to J
Step 7 : Is $A(J)$ greater than $A(J + 1)$?
(a) Yes : go to step 8
(b) No : go to step 9
Step 8 : Exchange values of $A(J)$ and $A(J + 1)$
That is, $T = A(J)$
$A(J) = A(J + 1)$
$A(J + 1) = T$
go to step 9
Step 9 : Is J equal to $N – I$
(a) Yes : go to Step 10
(b) No : increase value of J by 1
go to step 7
Step 10 : Is I equal to $N – 1$?
(a) Yes : go to step 11
(b) No : increase value of I by 1
go to step 6
Step 11 : Assign 1 to I
Step 12 : Print $A(I)$
Step 13 : Is I equal to N?
(a) Yes : go to step 14
(b) No : increase I by 1
go to step 12
Step 14 : End

Example 13: Conversion of decimal base number to binary

Step 1 : Input number in DECI
Step 2 : Let BBASE = 0 and $I = 0$
Step 3 : Store remainder of DECI/2 in REM
Step 4 : BBASE = BBASE + REM * (10 power I)
Step 5 : $I = I + 1$
Step 6 : DECI = (Integer) DECI/2

Step 7 : If DECI ≠ 0 go to step 3
Step 8 : Display BBASE
Step 9 : End

Example 14: GCD

Step 1 : Input *A* and *B*
Step 2 : If *A* > *B* swap *A* and *B*
Step 3 : *S* = *B* MOD *A* (remainder after *B*/*A*)
Step 4 : If *S* = 0
Display *A*
go to step 6
Step 5 : *B* = *A* and *A* = *S*
go to step 3
Step 6 : End

Example 15: Square root of a number

Step 1 : Read *n*
Step 2 : If *n* < 0
display "Enter real value"
go to step 1
Step 3 : Let EP = 0.0001
Step 4 : Let GUESS = *n*/2
Step 5 : while
GUESS * GUESS – *n* > EP OR GUESS * GUESS – *n* < – *EP*
begin
Step 6 : GUESS = ((GUESS + *n*) / GUESS)/2
end
Step 7 : Display GUESS
Step 8 : End

Example 16: Evaluate "sin *x*" as sum of series

$$[\sin x = x - \frac{x^3}{31} + \frac{x^5}{51} - \frac{x^7}{71} + ...n]$$

Step 1 : Read *N* and *X*
Step 2 : Let *I* = 1, SUM = 0, PW = 0, *L* = 1
Step 3 : SUM = SUM + ((– 1) power PW * (*X* power *I*/factorial *I*))
Step 4 : *I* = *I* + 2
Step 5 : PW = PW + 1
Step 6 : If *L* < = *N* go to step 3
Step 7 : Display SUM
Step 8 : End

Summary

- An algorithm is a finite set of rules, which gives a sequence of operations for solving a specific type of problem.
- The essential properties of an algorithm are: Finiteness, definiteness, generality, and effectiveness.
- Algorithms can be written for any type of problem: Business, scientific, or industrial.
- Algorithms can be classified as deterministic, non-deterministic, and random depending upon the manner in which the control shifts from one step to another.
- The nature of repetition in algorithms further classifies them as direct and indirect.
- The three components that are considered as standard units for controlling the flow of information processing are sequential, conditional, and repetitive.

Review Exercise

Multiple-Choice Questions

1. The purpose of algorithm is to show the
   (a) syntax of programming language
   (b) logic
   (c) time to execute the program
   (d) none of these
2. Which is the property of an algorithm?
   (a) finiteness
   (b) timelines
   (c) both (a) and (b)
   (d) none of these
3. Algorithm may be
   (a) direct
   (b) indirect
   (c) both (a) and (b)
   (d) none of these
4. Testing whether a number is prime is an example of
   (a) direct algorithm
   (b) indirect algorithm
   (c) both (a) and (b)
   (d) none of these
5. The word algorithm originates from the
   (a) Latin word
   (b) Arabic word
   (c) Indian word
   (d) none of these

State whether True or False

1. Non-deterministic algorithms are capable of exploring a large number of alternatives simultaneously to reach out to a correct solution.
2. In Indirect Algorithms, the number of repetitions is not known in advance.
3. If an algorithm involves a decision to be made based on a condition, the flow is said to be conditional.
4. Computing the Greatest Common Divisor is the example of Infinite Algorithms.
5. Program should be written first followed by an algorithm.

Fill in the Blanks

1. Computers can neither think nor make _____ on their own.
2. An algorithm should be able to solve _____ of a particular type.
3. If an algorithm involves a decision to be made based on condition, the flow is _____.
4. In _____ algorithms the number of repetetions is known in advance.
5. In a _____ employment steps are taken in an explicitly prescribed sequence.

Descriptive Questions

1. What are the essential properties of an algorithm?
2. Algorithms should be written prior to program coding. Discuss.
3. Differentiate between deterministic and non-deterministic algorithms.
4. How are direct algorithms different from indirect algorithms?
5. Explain the three components that are considered as the standard units for controlling the flow of information in algorithms.
6. Write algorithms to solve the following problems :
   (a) Print the table of 12.
   (b) Print first 10 even numbers.
   (c) Find largest number in an array.
   (d) Print elements of upper triangular matrix.
   (e) Multiplication of two matrices.
   (f) Evaluate a polynomial.

CHAPTER

3 Program Design and Implementation Issues

Key Features

- Programming
- System Design Techniques
- Programming Techniques
- Basic Constructs of Structured Programming
- Modular Design of Programs
- Communication between Modules
- Module Design Requirements

A program is a set of computer understandable instructions to solve a computational problem. The task of developing the programs is called programming. Computer programming started on a large scale in the year 1960. At that time, the size and scope of the programs was small and hence linear programming approach was in practice. But as the size and scope of programs grew, the traditional linear approach to programming made programs unstructured and difficult to understand. To overcome these difficulties, structured programming approach was introduced. This chapter describes programming, techniques of programming, and different approaches of programming.

PROGRAMMING

To solve a computation problem, its solution must be specified in terms of a sequence of computational steps, such that they are effectively solved by a human agent or by a digital computer. Computer understandable notation for the specification of such sequence of computational steps are referred to as a programming language.

The specification of the sequence of computational steps in a particular programming language is termed as a program. The task of developing programs is called programming and the person engaged in programming activity is called a programmer.

SYSTEM DESIGN TECHNIQUES

Top Down Design

The top down design approach is based on the fact that large problems become more manageable if they are divided into a number of smaller and simpler tasks which can be tackled separately. What is really required is that each of these parts have the properties of a module.

Top down design approach is performed in a special way. The main program is written first. It is tested before sub programs are written. To do this, the actual sub programs are replaced with **stubs**. The stubs simply test to see if the data is passed correctly. After the main program is written and checked, each module is written and tested in turn. This should first be done without the main program in order to isolate a stub if an error occurs. A simple main program is written to test the sub programs. If the modules run properly, then it is tested with the main program. If the module and the main program do not have a problem, then the next module is written and checked. To describe the program at its highest level, we use something called the "universal program", then by a process of "stepwise refinement", work out the details of each part of the program.

Advantages

- At each stage, the sub programs are tested by themselves and then the main program is tested. Whenever modules are added, they are tested with the main program so if any error occurs it will only be in a module and this will be easy to debug.
- It is desirable for modules to be kept small in general. As far as possible, a module should be less than 100 lines long.

Bottom Up Design

A bottom up approach would be to write the most basic subroutines in the hierarchy first and then use them to make more sophisticated subroutines. The pure bottom up approach is generally not recommended because it is difficult to anticipate which low-level subroutines will be needed for any particular program. It can often be a useful first step to produce a library of basic functions and procedures before embarking on a major project.

In the bottom up approach it is usually assumed that the basic routines created will be general enough to be used more than once. To construct a program using subroutines, repeating the same lines of code can be saved by re-using it. A routine that is used many times has a very difficult status to those higher in the hierarchy. It is more like a basic instruction in the programming language than a large scale program component.

PROGRAMMING TECHNIQUES

Linear Programming

Linear program is a method for straightforward programming in a sequential manner. This type of programming does not involve any decision making. A general model of these linear programs is:

- Read a data value.
- Compute an intermediate result.
- Use the intermediate result to compute the desired answer.
- Print the answer.
- Stop.

Structured Programming

One of the most versatile properties of a digital computer is that it can make a "decision", thus creating a branching point. There are also times when it becomes necessary for a program to "look back" over a

set of statements a number of times. If branching and looping can be used, then more complex iterative algorithms can be written, which in turn results in more complex programs. There are procedures that can be used for writing these complex programs that make them less error prone and easier to debug. The techniques for writing such programs are referred to as structured programming.

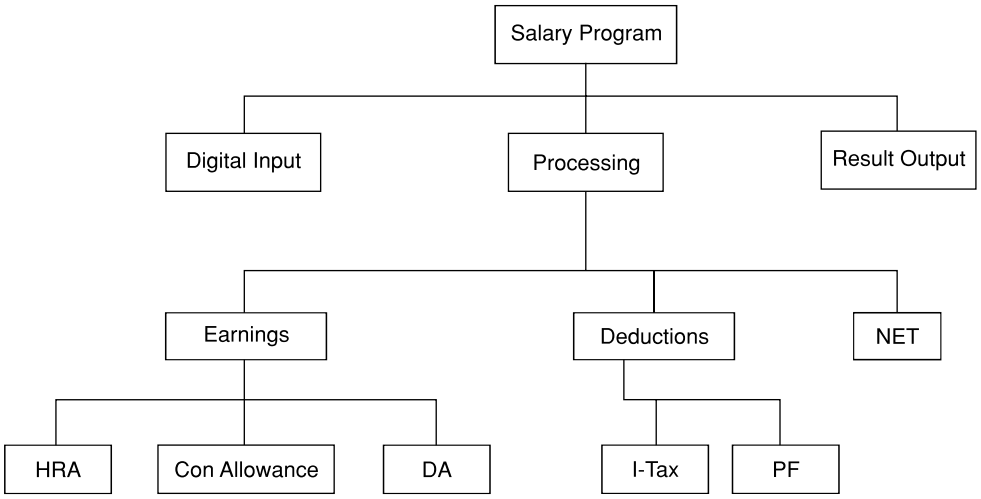


Fig. 3.1 *Example of structured programming*

Structured programming refers to the process in which we break the overall job down into separate pieces of modules. The above figure shows how a salary program is broken down into a number of small modules. These modules, in turn, are broken down into smaller pieces which can also be further subdivided. Modules must be chosen in such a way that we can specify how they are to interact. In effect, there is a contact between each pair of modules. This contact specifies two things:

- What will the module do?
- What are the assumptions it is making about the behaviour of the other modules? In particular, we must specify explicitly what inputs a particular module is to receive from the various other modules and what outputs it is to provide for them.

Advantages of Structured Programming

- Decreases the complexity of the program by breaking it down into smaller logical units
- Allows several programmers to code simultaneously
- Allows common functions to be written once and then used in all the programs needing it
- Decreases debugging time, because modules make it easier to isolate errors
- Amendments to single modules do not affect the rest of the program
- It saves time to use modular structures rather than using self-made structures. If a job can be done well by using what is already available and known to be well tried and tested then trying out something new for the sake of it, is a waste of effort
- Standard method; so less time is required in writing programs
- It is preferable to name modules in such a way that they are easy to find while documenting, and are consistent.

BASIC CONSTRUCTS OF STRUCTURED PROGRAMMING

There are three widely used programming constructs:

- Sequence
- Selection
- Repetition (iteration)

Now let us discuss each in detail.

Sequence

Sequence structure consists of the action followed by another till the desired result is obtained.

Statement 1

Statement 2

- - - - - - - - -

- - - - - - - - -

- - - - - - - - -

Statement *n*

Selection

In conditional execution, there is a need to carry out a logical test and then take some particular action which depends upon the result of that test.

The selection structure consists of a text for a condition followed by two alternative paths for the program to follow. The program selects one of the program control paths, depending upon the outcome of the test condition. After performing one of the two paths, the program control returns to a single point. This pattern can be termed as **if**. . . **else** because of the logic.

```
If (condition is true) then
    sequence of statements
else
    another sequence of statements
endif
```

Iteration

In most cases, programs require that a group of consecutive instructions be executed repeatedly until some logical condition has been satisfied. Such an iteration is called conditional looping.

Another type of repetition is unconditional looping. In such a looping, the instructions are repeated for a specified number of times.

The control structures are easy to use because:

- They are easy to recognize

- They are simple to deal with as they have just one entry and one exit point
- They are free of the complications of any particular programming language

MODULAR DESIGN OF PROGRAMS

One of the key concepts in the application of programming discipline is the design of a program as a set of units referred to as blocks or modules. A program module is defined as the part of a program that performs a separate function. For example: input, input validation, and processing of one type of input. A program module may be quite large, so that it may be further divided into logical sub modules. The process of subdivision continues until all modules are of manageable size in terms of complexity of logic and numbers of instructions.

Programs can be logically separated into the following functional modules:

- Initialization
- Input
- Input data validation
- Processing
- Output
- Error handling
- Closing procedure

The modules reflect a logical flow for a computer program. After initialization, processing proceeds logically with input, input validation, various processing modules, and output. Error handling may be required during execution of any module.

Basic Attributes

A module is a collection of program statements with five basic attributes:

- Input
- Output
- Function
- Mechanism
- Internal data

Control Relationship between Modules

The structure charts show the interrelationships of modules by arranging them at different levels and connecting modules in those levels by arrows. An arrow between two modules mean the program control is passed from one module to the other at the execution time. The first module is said to call or invoke the lower level modules.

There are three rules to control the relationship between modules:

- There is only one module at the top of the structure. This is called the root or boss module.
- The root passes control down the structure chart to the lower level modules. However, control is always returned to the invoking module and a finished module should always terminate at the root.

- There can be no more than one control relationship between any two modules on the structure chart. Thus, if module A invokes module B, then B cannot invoke module A.

COMMUNICATION BETWEEN MODULES

Two types of informations are passed between modules:

- Data : shown by an arrow with empty circle at its tail.
- Control : shown by a filled-in circle at the end of the tail of an arrow.

MODULE DESIGN REQUIREMENTS

A hierarchical (or modular) structure should present many advantages in management, developing, testing, and maintenance. However, such advantages will occur only if modules fulfil following requirements:

- Coupling : Coupling means the strength of relation between the modules in a system so that change in one module has limited effect on other modules. It means that coupling should be minimized.
- Cohesion : Cohesion means strength of relations within the module. Cohesion should be maximized.
- Span of control : It means the number of modules subordinate to the calling module. Limit of span of control may vary from 5 to 7 modules.
- Size : The number of instructions contained in a module should be limited as that module size is small.
- Shared use : Functions should not be duplicated in separated modules, but established in a single module that can be invoked by any other module when needed.

Summary

- A program is a set of computer understandable instructions to solve a problem.
- The task of developing a program is called programming.
- The structured approach adds the concept of hierarchies and of modules with single entry and exit points.
- In top down approach, the program is first designed in outline.
- In bottom up approach, the most basic subroutine would be written first then more sophisticated subroutines will be written.
- Linear programming means writing a program in a sequential manner.
- Sequence, selection, and iterations are the structured programming constructs.
- The purpose of module programming is to break up a complex task into smaller and simpler sub tasks.
- Data and control are two informations passed between the modules.
- Coupling means the strength of relations between modules. Cohesion means the strength of relations within a module.

Review Exercise

Multiple-Choice Questions

1. In which approach is the program first designed in outline?
   (a) Top Down
   (b) Bottom up
   (c) Both (a) and (b)
   (d) None of these
2. Programs can be logically seprated into following functional modules:
   (a) input
   (b) processing
   (c) output
   (d) all of these
3. In which programming approach are stubs used?
   (a) Top Down
   (b) Bottom up
   (c) Both (a) and (b)
   (d) None ot these
4. If statement belongs to
   (a) decision making statement
   (b) iteration statement
   (c) sequential statement
   (d) none of these
5. Advantage of structured programming:
   (a) allows several programmers to code simultaneoulsy
   (b) decreases debugging time
   (c) amendments to single modules do not affect the rest of the program
   (d) all of these

State whether True or False

1. Linear programming involves looping.
2. Top down approach involves working from general to specific.
3. Structured programming diminishes the efficiency of coding of a program.
4. In a sequential file on a disk, it is not possible to access a record directly.
5. Cohesion means strength of relations within the module.

Fill in the Blanks

1. Each module of a structured program should have one _____ point and one _____ point.
2. _____ means strength of relations between modules.
3. _____ means strength of relations within a module.
4. The design approach where the main program is decomposed into lower level module is called _____ .
5. _____ and _____ are two informations passed between modules.

Descriptive Questions

1. What are the advantages of structured programming?
2. Explain the basic constructs used in structured programming.
3. Differentiate between cohesion and coupling.
4. Write short notes on:
   (a) top down design
   (b) iteration constructs
5. Describe the programming concept.

CHAPTER

4

Programming Environment

Key Features

- Computer Programming Languages
- Types of Programming Languages
- Generations of Programming Languages
- Machine Level Language (1940–1950)
- Assembly Language (1950–1958)
- Assembler
- High Level Languages (1958 Onwards)
- Compiler
- Interpreter
- Popular High Level Programming Languages
- Fourth Generation Programming Languages (4GLs): (1985 Onwards)
- Application Program Generators (APGs)

Programming is the art of solving computational problems by computer. To solve these problems, one is required to write programs, which are normally written in programming languages. As a computer cannot understand natural language instructions, it is required to provide the instructions in special language known as programming languages. Programming languages are of two types: low-level and high-level languages.

This chapter primarily deals with programming languages and different high-level languages.

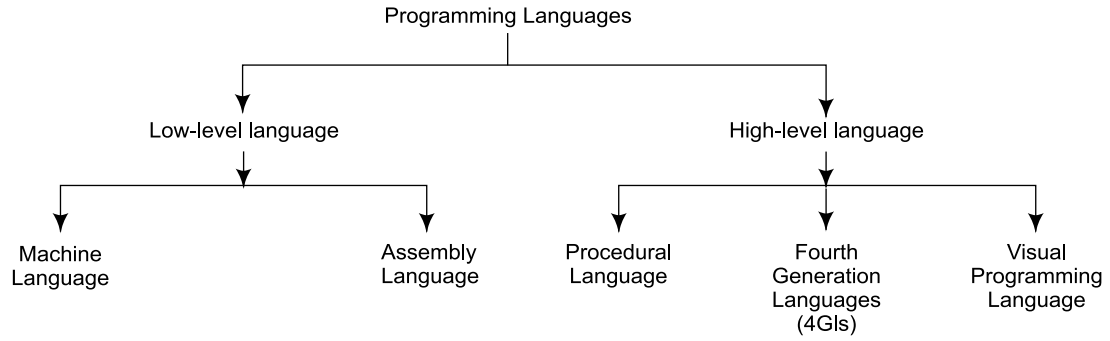
COMPUTER PROGRAMMING LANGUAGES

A computer is an electronic device which works on the instructions provided by the user. As a computer does not understand natural language, it is required to provide the instructions in some computer-understandable language. Such a computer-understandable language is known as programming language.

A computer programming language consists of a set of symbols and characters, words, and grammar rules that permit people to construct instructions in the format that can be interpreted by the computer system.

TYPES OF PROGRAMMING LANGUAGES

Types of programming languages can be easily explained by the following chart.



GENERATIONS OF PROGRAMMING LANGUAGES

On the basis of development, programming languages can be divided into five generations:

- First-Generation Language : Machine Languages (1940–50)
- Second-Generation Language : Assembly Languages (1950–58)
- Third-Generation Language : Procedural Languages (1958–85)
- Fourth-Generation Language : 4GLs (1985–Onwards)
- Fifth-Generation Language : Visual/Graphic Languages (1990–Onwards)

MACHINE-LEVEL LANGUAGE (1940–50)

A computer can understand binary codes (1, 0) only, so the instructions given to the computer can only be in 1 or 0. The language which contains binary codes is called machine-level language.

A typical machine-level language instruction contains essentially two parts:

- Operation part : which specifies what is to be performed.
- Address part : specifies the location of data to be manipulated.

Writing programs in machine-level language was a tedious task as it is difficult for human beings to remember all the binary codes.

Advantages

- Machine-level instructions are directly executable.
- Machine-level language makes most efficient use of computer system resources like storage and register.
- Machine language instructions can be used to manipulate individual bits.

Disadvantages

- As machine-level languages are device dependent, the programs are not portable from one computer to another.
- Programming in machine language usually results in poor programmer productivity.
- Programs in machine language are more error prone and difficult to debug.
- Computer storage locations must be addressed directly, not symbolically.
- Machine language requires a high level of programming skills, which increases programmer training costs.

ASSEMBLY LANGUAGE (1950–58)

Assembly languages are also known as second-generation languages. These languages substitute alphabetic symbols for the binary codes of machine language.

In assembly language, symbols are used rather than absolute addresses to represent memory locations.

Mnemonics are used for operation code, that is, single letters or short abbreviations that help the programmers to understand what the codes represents.

Example: MOV, AX, DX.

Here mnemonic MOV represents "Transfer" operation and AX, DX are used to represent the registers.

Advantages

- Assembly language is easier to use than machine language.
- An assembler is useful for detecting programming errors.
- Programmers do not have to know the absolute addresses of data items.
- Assembly languages encourage modular programming.

Disadvantages

- Assembly language programs are not directly executable.
- Assembly languages are machine dependent and, therefore, not portable from one machine to another.
- Programming in assembly language requires a higher level of programming skill.

ASSEMBLER

An assembly language program is not directly executable. To be executed, it is first required to change it into its machine language equivalent.

An assembler is a program which is used to translate an assembly language program into its machine-level language equivalent. The program in assembly language is termed as source code and its machine language equivalent is called object program.

Once the object program is created, it is transferred into the computer's primary memory using the system's loader. Here, another program called link editor passes computer control to the first instruction in the object program, then the execution starts and proceeds till the end of the program.

HIGH-LEVEL LANGUAGES (1958 ONWARDS)

These are the third-generation languages. These are procedure-oriented languages and are machine independent. Programs are written in English-like statements. As high-level languages are not directly executable, translators (compilers and interpreters) are used to convert them in machine language equivalent.

Advantages

- These are easier to learn than assembly language.
- Less time is required to write programs.
- They provide better documentation.
- They are easier to maintain.
- They have an extensive vocabulary.
- Libraries of subroutines can be incorporated and used in many other programs.

- Programs written in high-level languages are easier to debug because translators display all the errors with proper error messages at the time of translation.
- Programs written in high-level languages are largely machine independent. Therefore, programs developed on one computer can be run on another with little or no modifications.

COMPILER

For execution of high-level language programs, it is required to translate them into machine-level language equivalent. The system programs used to do so are called compilers.

Compiler takes high-level language program (**source code**) as input and produces machine-level language program (**object code**) as output. Compiler allocates addresses to all variables and statements. This generates object program and produces a printed listing of source and object programs if required. It also tabulates a list of programming errors found during compilation.

By and large all the high-level languages are compiler-based languages. Examples of compiler-based languages are **C, Cobol, Pascal,** and **FORTRAN**.

INTERPRETER

It is also used for translating high-level language program into machine-level language. The main difference between compiler and interpreter is that compiler translates entire program first and then produces the list of errors, while the interpreters perform line by line translation. As soon as the error is encountered, interpretation process stops. Due to line-by-line translation, interpreters are slower than compilers. The most well-known interpreter-based language is **BASIC**.

POPULAR HIGH-LEVEL PROGRAMMING LANGUAGES

- **ADA** Named after Lady Augusta Ada Byron (*the first computer programmer*). It was designed by the US Defence Department for its real time applications. It is suitable for parallel processing.
- **APL** Developed by Dr. Kenneth Aversion at IBM. It is a convenient, interactive programming language suitable for expressing complex mathematical expressions in compact formats. It requires special terminals for use.
- **BASIC (Beginners All-purpose Symbolic Instruction Code)** This language was developed by John Kemeny and Thomas Karthy at Dartmouth College. It is widely known and accepted programming language. It is easy to use and is almost coded in real-time conversational mode. This language provides good error diagnostics but has no self-structuring or self-documentation.
- **Pascal** Developed in 1968. It was named after the French inventor Blaise Pascal and was developed by the Swiss programmer Nikolus Wirth. Pascal was the first structured programming language and it is used for both scientific and business data processing applications.
- **FORTRAN (FORmula TRANslation)** Developed by IBM in 1957. It is one of the oldest and most widely used high-level languages. It is widely used by scientists and engineers because this language has huge libraries of engineering and scientific programs.
- The language is suitable for expressing formulae, writing equations, and performing iterative calculations.
- Various versions of FORTRAN are:
  FORTRAN I (1957)
  FORTRAN II (1958)

FORTRAN IV (1962)
FORTRAN 77 (1978)

- **COBOL (COmmon Business Oriented Language)** Developed by a committee of business, industry, government, and academic representatives called *CODASYL* (*CO*nference on *DA*ta *SY*stem *L*anguages) commissioned by the US government in 1959. Statements of COBOL language resemble English language expressions and it makes them easy to understand and use. COBOL is a structured and self documented language.
- **C** Developed by Dennis Ritchie at the Bell Laboratories in 1970. It is a general purpose programming language, which features economy of expression, modern control flow and data structures, and a rich set of operators. Its programs are highly portable (machine independent). C language supports modular programming through the use of functions, subroutines, and macros.
- **LISP (LISt Processor)** Developed in 1960 by Prof. John McGrthy. Lisp and Prolog (PROgramming LOGic) are the primary languages used in artificial intelligence research and applications.
- **RPG (Report Program Generator)** Developed by IBM in the late 1960's, RPG is an important business-oriented programming language. It is primarily used for preparing written reports.

Advantages

- RPG is problem oriented.
- RPG is easy to learn and use.
- Limited programming skills are required.

FOURTH-GENERATION PROGRAMMING LANGUAGES (4GLs): (1985 ONWARDS)

4GLs were developed in the late 1970s and early 1980s. They concentrate on what is to be accomplished rather than how it is to be accomplished. 4GLs emphasize on end results. It stresses on specifying the dimensions of a problem. This speeds up programming.

4GLs facilitate interactive coding in the form of an on-screen menu choice to formulate an enquiry or define a task. Limited training is required as it is very easy to learn a 4GL in comparision to a 3GL.

4GLs offer increased productivity. They have increased memory requirements.

All the database management packages and spreadsheet packages fall in the category of 4GLs.

DBMS: dBase IIIplus, Fox Base, FoxPro, Soft Base, Oracle.

Spreadsheet packages: Lotus-123, Soft calc, MS-Excel.

APPLICATION PROGRAM GENERATORS (APGs)

APGs offer the facilities required in a normal data processing job. Codes can be automatically generated, given input information data definition, procedure definition, and report enquiry.

Summary

- A computer programming language consists of a set of symbols, characters, and grammar rules that permit people to construct computer instructions.
- There are two types of programming languages: low-level and high-level.
- Languages are divided into five generations.
- A computer can understand machine language instructions.

- Assembly language is easier than machine language. It uses symbols in place of binary digits.
- High level languages use English-like statements for instructions.
- Programs used to convert assembly language programs into machine language are called assemblers while those that convert high-level language programs into machine-level languages are called compilers and interpreters.
- Some popular high-level languages are ADA, BASIC, C, COBOL, and RPG.

Review Exercise

Multiple-Choice Questions

1. Example of translator:
   (a) assembler
   (b) compiler
   (c) interpreter
   (d) all of these
2. 'C' was developed in which generation?
   (a) First Generation
   (b) Second Generation
   (c) Third Generation
   (d) None of these
3. Example of popular package:
   (a) 'C'
   (b) ADA
   (c) MS-OFFICE
   (d) none of these
4. Mnemonics are used in
   (a) 'C'
   (b) ADA
   (c) Assembly
   (d) C++
5. FoxPro is in which category?
   (a) Low level language
   (b) Third-generation language
   (c) Fourth-generation language
   (d) 4GL package

State whether True or False

1. Machine-level instructions are directly executable.
2. Assembly languages are also known as third-generation languages.
3. Assembly languages are machine dependent.

4. ADA is a popular fourth-generation programming language.
5. Languages are divided into five generations.

Fill in the Blanks

1. Machine-level programs are _____ dependent.
2. Two parts of machine-level language instructions are _____ and _____ .
3. Computer program called _____ passes control to the first instruction in the object program.
4. Computer accepts _____ program and produce _____ code.
5. _____ is an interpreter-based language.
6. _____ and _____ are used in artificial intelligence research.

Descriptive Questions

1. What is the difference between compiler and interpreter?
2. What are the parts of an assembly language instruction?
3. Discuss the advantages and disadvantages of machine language.
4. Write short notes on the following:
   (a) Pascal
   (b) RPG
   (c) Compiler
   (d) Assembler
   (e) Interpreter
5. Write down the classifications and generation of programming languages.

CHAPTER

5

Overview of C Language

Key Features

- History of 'C' Language
- Features of 'C' Language
- Why is C Language Popular?
- Using Borland 'C' on DOS Platform
- Using Borland 'C' on UNIX Platform
- Components of 'C' Language
- Structure of a 'C' Program
- A Sample 'C' Language Program
- Process of Executing a 'C' Program

'C' is a programming language developed at Bell Laboratories, US, in the year 1972 by Dennis Ritchie. 'C' is an extremely popular language because it is simple, efficient, and reliable. 'C' is considered to be a middle-level language since it has the features of both low-level language and high-level language. On one hand, 'C' has relatively good programming efficiency, on the other, it has relatively good machine efficiency.

This chapter gives the reader an overview of 'C' language and explains about the structure of 'C' programs and various components of 'C' program.

HISTORY OF 'C' LANGUAGE

Developing a common programming language which can be used to solve different types of problems on various hardware platforms has always been a computer professional's dream. One such attempt was development of a language called Combined Programming Language (CPL) at Cambridge University in 1963. However, it turned out to be too complex, hard to learn and difficult to implement. Subsequently, in 1967, a subset of CPL, Basic CPL (BCPL) was developed by Martin Richards incorporating only the essential features of CPL. However, it was also not found to be sufficiently powerful. Around the same time, in 1970, another subset of CPL, a language called B was developed by Ken Thompson at Bell Labs. However, it turned out to be insufficient in general. In 1972, Dennis Ritchie at Bell Labs developed 'C' language incorporating best features of both BCPL and B languages.

FEATURES OF 'C' LANGUAGE

'C' is often termed as a middle level programming language because it combines the power of a high-level language along with the flexibility of a low-level language. High-level languages have a number of built-in features and facilities which result in high programming efficiency and productivity. Low-level languages, on the other hand, are designed to give more efficient programs and better machine efficiency.

'C' is designed to have a good balance between both the extremes. Programs written in 'C' give relatively high machine efficiency as compared to high-level languages (though not as good as low-level languages). Similarly, 'C' language programs provide relatively high programming efficiency as compared to low-level languages (though not as high as those provided by high-level languages). Thus 'C' can be used for a range of applications with equal ease and efficiency.

WHY IS 'C' LANGUAGE POPULAR?

Several features which make 'C' a suitable language to write system programs are as follows:

- 'C' is a machine independent and highly portable language.
- It is easy to learn; has only 28 keywords.
- It has a comprehensive set of operators to tackle business as well as scientific applications with ease.
- Users can create their own functions and add to 'C' library to perform a variety of tasks.
- 'C' language allows manipulation of BITS, BYTES, and ADDRESSES.
- It has a large library of inbuilt functions.
- 'C' operates on the same data types as the computer, so the codes generated are fast and efficient.

USING BORLAND 'C' ON DOS PLATFORM

Login your terminal with correct login name and password. Type BC at the DOS prompt (I:\>) and press *Enter*. The Borland Editor's main screen appears on the display as shown below in figure 5.1.

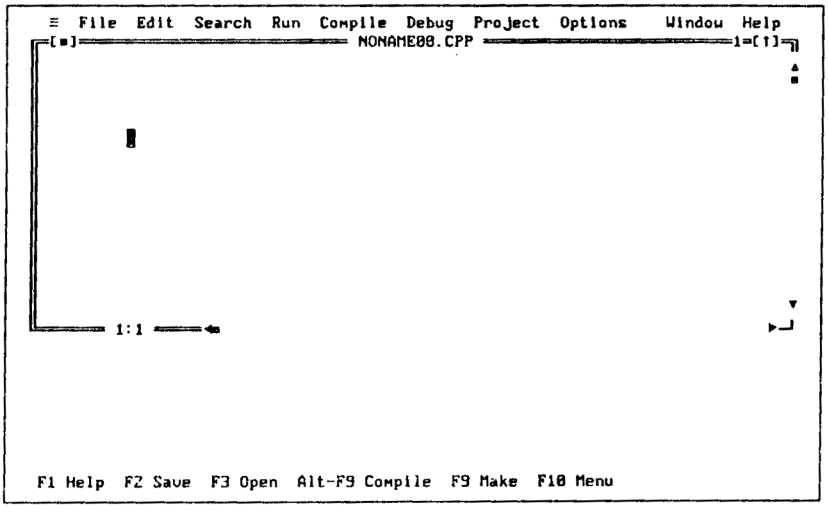


Fig. 5.1 *Borland Editor's main screen*

There are 11 pull-down menus available:

- = : Interface to external programs such as Borland Assembler and Borland Debugger.
- File : File related operations such as opening and saving file.
- Edit : Cut, Copy, Paste operations.

- Search : Find, Find & Replace operations.
- Run : Compile and Run the file currently loaded in the text editor. Debugging such as setting/clearing trace points can also be performed from this menu.
- Compile : This menu item compiles a source file to an object file or an .exe file.
- Debug : Provides interactive debugging. Variables can be examined/set/cleared, and you can watch variables change during execution.
- Project : This menu item controls Borland C++'s handling of large programs that are in multiple source file. All the files of program are specified in .prj file.
- Option : Default options are set during installation. The user can change any option any time through this menu.
- Windows : Window operations include zoom, arranging windows on the screen, and closing windows.
- Help : Borland C++ includes a context sensitive help capability. Select Help, or Press F1 for General Help, Shift F1 for Indexed Help, and Ctrl F1 for Context Sensitive Help.

Accessing Pull Down Menu

Pull down menus can be accessed in several ways:

- Press F10, then move the cursor to the desired menu item using the arrow keys and press Enter.
- Press Alt and first character of menu item. Example: Alt F to pull down the File menu.

Editing File

You may either want to open a new file and start writing your program, or wish to update or append an existing program in storage. To open a new file, first press Alt F, and then *Enter*. The file pull down menu will be activated. Select *New* and Press *Enter.* The display screen will change into a blank screen of text editor with file name NONAME00.CPP on which you can start typing your code through the keyboard.

To open an existing file, first press Alt F and then *Enter.* From the activated File pull down menu, select *Open*. An Open File dialog box will appear. Enter your required file name in the given text box and Press *Enter*. To access a file stored on a drive and directory other than the current ones, you may also change drive and directory through this Open File dialog box. The screen of the text editor will start displaying the contents of your required file. Now using general editing commands, new programs can be written or old programs can be updated.

Saving Your Work

It is important to save your work frequently to avoid any accidental loss of code.

- Press Alt F to activate the File menu.
- Move the cursor to Save As by pressing down arrow key three times.
- Press Enter to activate the Save As selection.
- Type filename alongwith extension .c in the given text box and press Enter to save the program.

Running a Program

Press Alt R to pulldown the *Run* menu and *Enter* to begin compilation. A compile window opens during compilation to show progress. Execution can also be started by pressing Ctrl F9 shortcut key combination.

The output can be viewed in output window. Press Alt W to pull down the *Window* menu. Move the cursor down to *Output* and press *Enter* to view the output when you execute the sample program.

Exiting Borland C

Press Alt X to exit Borland 'C' and return to DOS prompt.

USING BORLAND 'C' ON UNIX PLATFORM

Screen Editor vi

Visual Editor is a utility to create and edit text files. It is a screen-oriented editor.

Invoking vi

vi [filename....]	Opens a new or existing file. More than one file name can be specified with vi. Example: *vi Myfile* to edit *Myfile*
vi +n [filename]	Positions the particular line in the middle of the window and the cursor at the beginning of that line. Example: *vi +5 Myfile* to edit *Myfile* positioning from 5th line.
vi + /Pattern [filename]	Positions the first line containing that pattern in the middle of the window and the cursor at the beginning of that line.

Leaving vi

Esc:wq	Writes buffer to the file and quits *vi.*
Esc:q !	Quits *vi* without saving file.
Esc:W	Writes the current buffer to the file.
Esc:zz	Writes buffer to file and quits *vi*.
Esc:x	Writes buffer to file and quits *vi.*

Basic Cursor Movements

Moving Around the Screen

h or ←	Moves one character left.
i or →	Moves one character right.
j or ↓	Moves one character down.
k or ↑	Moves one character up.
Shift ^or o	Moves cursor to the beginning of the line.
$	Moves cursor to the end of the line.
-	Moves to the beginning of the previous line.
Return	Moves to the beginning of the next line.
H	Moves the cursor to the beginning of the first line of the screen.
M	Moves the cursor to the beginning of the middle line of the screen.
L	Moves the cursor to the beginning of the last line of the screen.

Screen Scrolling

z command is used for screen scrolling with following options:

z <rtn>	Places current line at the top of screen.
z.	Places current line in the middle of screen.

z- Places current line at the bottom of screen.

Scrolling within a window can be done by:

Ctrl-C Scrolls up line-by-line till the end of the screen can be seen.
Ctrl-Y Scrolls down one line and one more line at the top of screen can be seen.
Ctrl-d Scrolls down half the window.
Ctrl-u Scrolls up half the window.

The window can be moved up and down the file with:

Ctrl-f Moves the window forward through file.
Ctrl-b Moves the window backward through file.

Text Operation

Text Addition

a Appends the text after the cursor position.
i Inserts the text from the current cursor position.
o Opens a new line for text entry below the current line.
A Appends the text at the end of the current line.
I Inserts the text at the beginning of the line.
O Opens a new line for text entry above the current line.
* Esc command is used to terminate any insert operation.

Text Deletion

x Deletes a character in the current cursor position.
nx Deletes *n* characters from the current cursor position.
X Deletes a character previous to the cursor position.
nX Deletes *n* characters before the cursor position.

Text Substitution

S Substitute the current character.
nS Substitute *n* characters.

Text Replacement

r Replaces the character in the current cursor position with a letter followed by r.
nr Replaces *n* characters by the single letter following r.
R Replaces until Esc is pressed.

Operations with Words

Movement Across Words

w Moves cursor forward to the beginning of the next word.
e Moves cursor forward to the end of the word.
b Moves cursor backward to the beginning of the word.

Deletion of Words

dw [ndw or dnw] Deletes a word [*n* words].

Changing Words

cw [ncw] Changes a word [*n* words].

Operations with Lines

Deleting Parts of Line

D or d$	Deletes the rest of the line after the current cursor position.
d^	Deletes from the cursor to the first printable character of the line. (Deletes line from the beginning to the current cursor position)

Changing Parts of Line

c$	Changes till the end of the line.
C or c^	Changes from the beginning of the line to the current cursor position.
Esc	Terminates the change operation.

Deleting and Changing Entire Line

dd	Deletes the current line
cc	Changes the current line.

Joining Lines Together

J	Joins the next line to the current line.
nJ	Joins the next *n* lines together.

Move Operation on Text Blocks

In order to move the lines from one position to another, there are two options: either remove the lines from the old position and place them in a new position, or copy the lines from the old position to the new.

Moving Text with Delete and Put

Delete lines from the current position using *ndd* command. Move the cursor to the new position and then use one of the following commands:

P	To patch the deleted block below the current line.
p	To patch the deleted block above the current line.

Copying Text with Yank and Put Copy the lines from current position by using the *nyy* command. Move the cursor to the new position and then use one of the following commands:

P	To put the yanked block below the current line.
p	To put the yanked block above the current line.

Search Operations

	Use/command for finding a particular pattern anywhere in the text.
// or n	It is used to find other occurences of a pattern in forward direction.
?? or N	It is used to find other occurences of a pattern in backward direction.

Using C Compiler on Unix Platform

Login your terminal with correct login and password. On the *$* prompt invoke the *vi* editor by issuing a command *vi filename, vi* screen opens. Press *'i'* for the insert mode and then type the program through the keyboard. After completing the typing, save your work and exit from *vi* by inputting *Esc :wq* command. Then the program is compiled by issuing command *cc file name* at *$* prompt and run by giving command *a.out* on *$* prompt.

COMPONENTS OF 'C' LANGUAGE

The five main components of 'C' language are:

- Character Set
- Data Types
- Constants
- Variables
- Keywords

Character Set

Any alphabet, digit, or special symbol used to represent information is denoted by a character. The characters in C are grouped into four categories:

- Letters : *A - Z* or *a - z*
- Digits : 0, 1, - - - - 9
- Special Symbols : ~ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /
- White spaces : blank space, horizontal tab, carriage return, new line, and form feed.

Data Types

The power of a programming language depends, among other things, on the range of different types of data it can handle. 'C' language is designed to handle five primary data types, namely, character, integer, float, double and void, and several secondary data types like array, pointer, structure, union, and enumeration.

Constants

A constant is a fixed value entity that does not change. It can be stored at a location in the memory of the computer and can be referenced through that memory address.

Variables

A variable is an entity whose value can change during program execution. A variable can be thought of as symbolic representation of address of the memory whose values can change.

Keywords

Keywords are those words which have been assigned specific meaning in the context of 'C' language programs. Keywords should not be used as variable names to avoid problems or compile time errors.

STRUCTURE OF A 'C' PROGRAM

Every 'C' program is made of functions and every function consists of instructions called statements.

Functions

Every 'C' program is structured as a collection of one or more distinct units called function. Each function comprises of a set of finite statements and is designed to perform a specific task. Each function is given a unique name for reference purposes and is treated as a single unit by the 'C' compiler. A function name is always followed by a pair of parenthesis, namely, ( ). The statements within a function

are always enclosed within a pair of braces { }. Every 'C' program must necessarily contain a special function named main( ). The program execution always starts with this function. The main function is normally, but not necessarily, located at the beginning of the program. The group of statements within main( ) are executed sequentially. When the closing brace of the main function is reached, program execution stops and the control is transferred back to the operating system.

Statements

Single 'C' language instruction is called a statement. They are to be written in accordance with the grammar of 'C' language. Blank spaces are inserted between words to improve the readability. However, no blank spaces are allowed within a word. Usually all 'C' statements are entered in lower case letters. 'C' is a free form language. There are no restrictions regarding the format of writing 'C' language statements. The statements can start and terminate anywhere on the line. It can also be split over multiple lines. Every 'C' language statement always ends with a semicolon.

A SAMPLE 'C' LANGUAGE PROGRAM

```
/* Program to calculate area of circle */
#include <stdio.h>
main ( )
{
   float radius, area;
   printf ("Enter the radius\n") ;
   scanf ("%f", &radius) ;
   area = 3.14 * radius * radius ;
   printf ("Area = %f\n", area) ;
}
```

The first line is a comment that identifies the purpose of the program.

The second line contains a reference to a special file (stdio.h) which contains information that must be included in the program. This file contains the important and normally used I/O functions of 'C' language. It is a preprocessor directive that we talked about in the last section.

The third line is a heading for the function *main ( )*.

The remaining five lines of the program are indented (for clear reading, (it does not affect the execution of program)) and enclosed within a pair of braces.

The first indented line is a variable declaration. It establishes the names *radius* and *area* as floating point variables.

The remaining four indented lines are expression statements. The second indented line (*printf*) generates a request (a value for the radius). This value is accepted by the system via the third indented line (*scanf*).

The fourth indented line is an assignment statement. This statement causes the area to be calculated from the given value of the radius.

The last indented line (printf) causes the calculated value for the area to be displayed on the output screen.

Notice that each individual expression statement ends with a semicolon.

PROCESS OF EXECUTING A 'C' PROGRAM

The process of executing a program in 'C' involves the following steps:

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

The process of creating, compiling and executing a 'C' program is shown in the following figure. The steps remain the same irrespective of the operating system though the system commands for implementing the steps and the convention followed for naming files may vary on different systems.

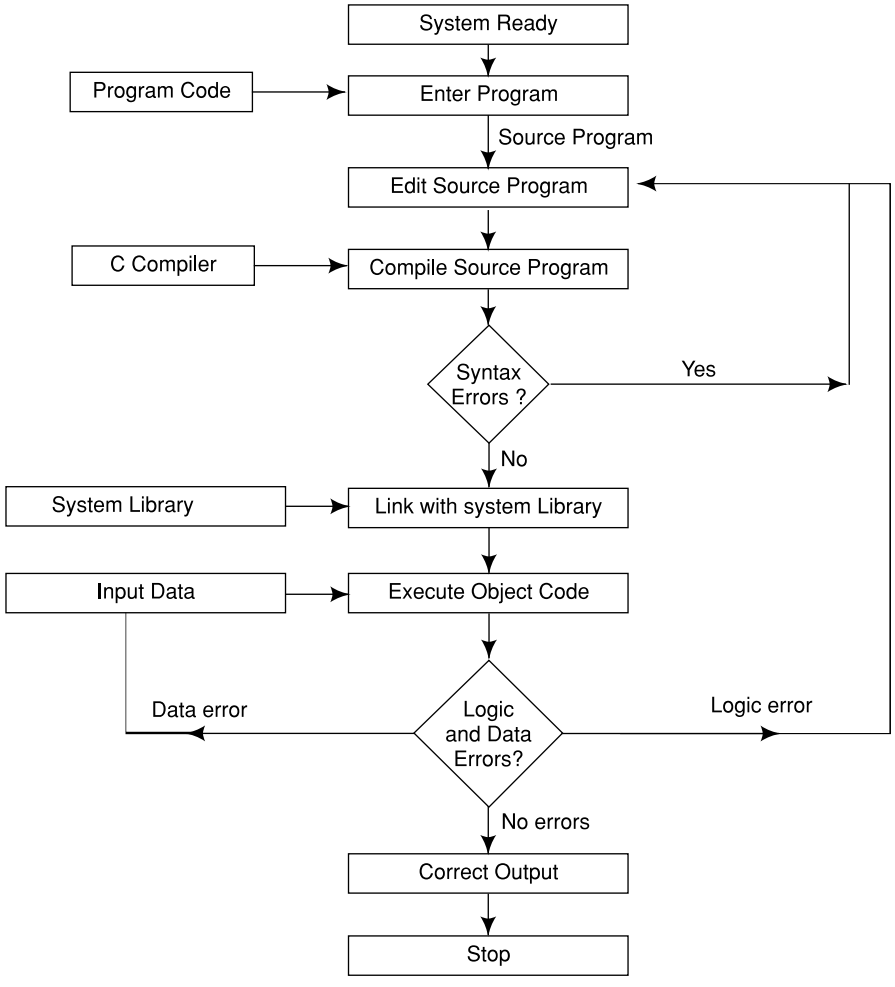


Fig. 5.2 *Process of executing a 'C' program*

Let us discuss each step in detail:

1. ***Creating the Program*** The program that is to be created must be entered into a file. The file name can consist of letters, digits and special characters, followed by the extension *.C*. Examples of valid file names are:
   first.C
   ab12.C
2. ***Compiling and Linking*** During the compilation process, the source program instructions are translated into a form that is suitable for execution by the computer. The translation process checks each and every instruction for correctness and if no errors are reported then it generates the object code.
   During the linking stage the other program files and functions that are required by the program are put together with it.
   If mistakes in the syntax and semantics of the language are discovered, they are listed out and the compilation process ends there. The errors should be corrected in the source program with the help of an editor and the compilation done again.
3. ***Executing the Program*** During execution, we load the executable object code in the computers memory and execute the instruction.
   During execution, the program may request some data through the keyboard.

Summary

- 'C' is a middle-level language with features of both high-level language and low-level language.
- Presently used 'C' language was developed by Dennis Ritchie at Bell Labs.
- Five main components of 'C' language are (1) character set (2) data types (3) constant (4) variable (5) keywords.
- Any 'C' program is made of functions which, inturn, are made of statements.
- Function is a set of valid 'C' statements designed to perform a specific task.
- A single 'C' language instruction is called a statement.

Review Exercise

Multiple-Choice Questions

1. The acronym CPL expands to
   (a) Common Programming Language
   (b) Combined Programming Language
   (c) Complex Programming Language
   (d) None of these
2. 'C' is also often termed as
   (a) assembly language
   (b) high-level language
   (c) middle-level language
   (d) low-level language

3. The shortcut key combination to compile a 'C' program is
   (a) F2
   (b) Ctrl + F2
   (c) Alt + F2
   (d) Alt + F9
4. Every 'C' language statement always ends with a
   (a) colon
   (b) comma
   (c) semicolon
   (d) full stop
5. In 'C' language there are _____ keywords.
   (a) 52
   (b) 16
   (c) 28
   (d) 12

State whether True or False

1. In special cases, keywords can be used as variable names.
2. All 'C' statements are written in lowercase.
3. In a 'C' program, more than one statement can be written in one line.
4. Compiler detects logical errors.
5. A function name is always followed by curly braces.

Fill in the Blanks

1. Basic combined programming language (BCPL) was developed by _____ .
2. 'C' is often termed as _____ level language.
3. 'C' has only _____ keywords.
4. 'C' has _____ primary data types.
5. _____ is an entity whose value can change during program execution.
6. Every 'C' program must contain _____ function.
7. Any 'C' statement must end with _____ .

Descriptive Questions

Write a 'C' program to do the following:

1. Accept length and breadth of a rectangle as input and display the area after calculation.
2. Accept 5 numbers and calculate their average.
3. Display a welcome message on the screen.
4. Print the entered name three times on the screen on separate lines.
5. Print the square of any given integer.

CHAPTER

6 Data Types, Variables, and Constants

Key Features

- Data Types
- Variables
- Constants

To communicate with the computer, it is required to pass values to the computer from the keyboard. The values passed may be of different types, namely, a single character, an integer, or a decimal string. Each of these requires memory area of a different size. Such defined types for the values are termed as data types. The defined type of memory area where the value is stored is termed as a variable. There may be some such values which do not change during the execution of the program. These values are termed as constants.

This chapter will discuss in detail about data types, variables, and constants.

DATA TYPES

Data values passed in a program may be of different types. The 'C' datatypes can be broadly divided into two categories—Primary data types and Secondary data types.

The Figure 6.1 below describes the various data types available in 'C' language.

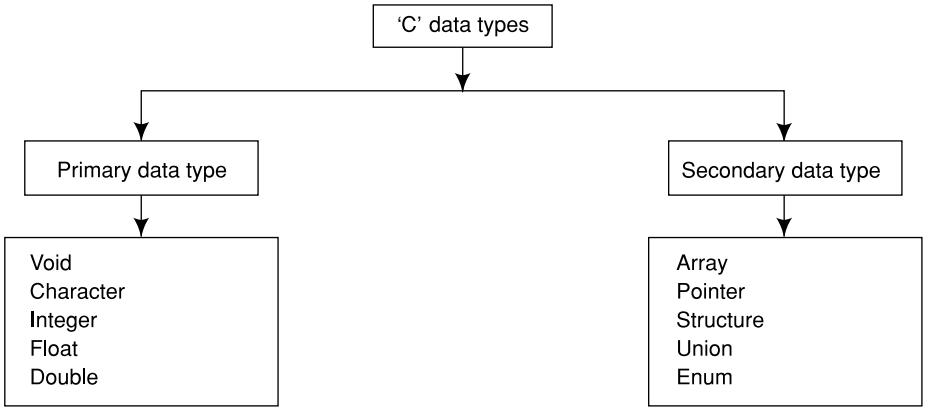


Fig. 6.1 *Data type in 'C' language*

Primary Data Types

There are five foundational data types in 'C':

Character, integer, floating-point, double floating point and valueless. In 'C' they are declared using **char, int, float, double** and **void** respectively. These data types form the basis for various other data types and are hence called primitive data type.

The size and range of these data types depend upon the processor type and compilers. The size of an object of type char is always 1 byte. The size of int is generally same as the word length of operating system. For example for a 16 bit operating system like DOS or Windows 3.1 the size of an int is 16 bits however for a 32 bit operating system like Windows 95/98/NT/2000 the size of an int is 32 bits. Every compiler is free to choose appropriate sizes for its own hardware, subjected only to the restriction that short and int data types are at least 16 bits long data types are at least 32 bits, short is no longer than int, which is turn cannot be longer than long.

Table 6.1 Data types and their details

Date Type	*Meaning*	*Storage Space*	*Format*	*Range of values*
char	A character	16 byte	% c	-127 to 127
int	An integer	1 or 2 byte	% d	-32767 to 32767
float	An single precision floating number	4 bytes	% f	1E-37 to 1E+37 with six digits of precision
double	A double precision floating number	8 byte	% If	1E-37 to 1E+37 with ten digits of precision
void	Value less	0 byte		

Example program 1

```
#include <stdio.h>
main( )
{
char c1, c2;
int i1, i2;
c1 = 'a';
c2 = 'b';
i1 = 65;
i2 = 66;
printf ("c1 and c2 as character values are :%c, %c \n", c1, c2);
printf ("c1 and c2 as integer values are :%d, %d \n", c1, c2);
printf ("i1 and i2 as character values are :%c, %c \n", i1, i2);
printf ("i1 and i2 as character values are :%d, %d \n", i1, i2);
}
```

```
Output:  c1 and c2 as character values are : a, b
         c1 and c2 as integer values are : 97, 98
         i1 and i2 as character values are : A, B
         i1 and i2 as integer values are : 65, 66
```

Explanation Numeric data values are stored in the memory in their binary form while the character data has to be codified as a unique integer and the code number is stored in the internal storage. The integer equivalents of the alphabets are:

Lower case : $a - z \equiv 97 - 122$

Upper case : $A - Z \equiv 65 - 90$

In the above program when the characters are displayed in an integer format, the corresponding ASCII codes are displayed. Similarly when the integers are displayed in a character format, their equivalent characters are displayed.

sizeof( ) operator To determine the size of variable in bytes, sizeof( ) operator is used.

Example:

```
#include <stdio.h>
main( )
{
   int i;
   char c;
   printf ("size of i is % d bytes", sizeof(i));
   printf ("\n size of C is %d bytes", sizeof(c));
   printf ("\n size of float data type is % d bytes", sizeof(float));
}
Output : size of i is 2 bytes
   size of c is 1 bytes
   size of float data type is 4 bytes.
```

Secondary Data Types

Also known as derived data types, secondary data types are derived from the basic data-types. They are five in number.

- Array : Sequence of objects, all of which are of the same type.
  Example: int num[5];
  reserves a sequence of five location of 2 bytes, each, for storing integers.
- Pointer : Used to store the address of any memory location.
- Structure : Sequence of objects of different data types.

Example: A structure of an employee's data, that is, name, age, salary is as:

```
struct emp
{
    char name [20];
    int age;
    float salary;
};
```

- Union : Sequence of objects of different types sharing common memory space.
- Enumerated data type : Its members are the constants that are written as identifiers though they have signed integer values. These constants represent values that can be assigned to the corresponding enumeration variables.
  Enumeration may be defined as
  enum tag {member1, member2 - - member *n*};
  Example: enum colours {red, green, blue, cyan}; colours foreground, background;
  The first line defines an enumeration named "colours" which may have any one of the four colours defined in the curly braces. In the second line, variables of the enumerated data type "colours" are declared.

VARIABLES

Variables are the data items whose values may vary during the execution of the program. A specific location or address in the memory is allocated for each variable and the value of that variable is stored in that particular location.

All the variables must have their type indicated so that the compiler can record all the necessary information about them and generate the appropriate code during the translation of the program.

Rules for Constructing Variable Name

- Variable name may be a combination of alphabet, digits, or underscores and its length should not exceed eight characters.
- First character must be an alphabet.
- No commas or blank spaces are allowed in variable name.
- Among the special symbols, only underscore can be used in variable name.

Example: emp_age and item_4

Variable Declaration and Assignment of Values

All the variables must be declared before their use. Declaration does two things:

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

A variable declaration has the form:

```
type_specifier   list_of_variables;
```

Here *type_specifier* is one of the valid data types. *list_of_variables* is a comma-separated list of identifiers representing the program variables.

Examples:

- int *i, J, K*;
- char ch;

To assign values to the variable, assignment operator (=) is used. Assignment is of the form:

variable_name = value;

But before the assignment, the variable must be declared.

Examples:

- int *i*, *j*;
- *j* = 5;
- *i* = 0;

It is also possible to assign a value to the variable at the time of declaration.

data_type variable_name = value;

The process of giving initial values to the variable is known as **initialization**. More than one variable can be initialized in one statement using multiple assignment operators.

Examples:

- int *i* = 5;
- int *j*, *m*;
  j = *m* = 2;

But there is an exception. Consider the following example:

int *i*, *j* = 2, *k*;

Here the assignment will be

i = garbage value
j = 2
k = garbage value

Example program:

```
/* Example of assignments */
main( )
{
   /* declaration */
   int a1, b1;
/* declaration & assignment */
   int var = 5;
   int a, b = 6, c;
/* declaration & multiple assignment */
   int p, q, r, s;
   p = q = r = s = 5;
```

```
/* printing of values * /
   printf ("var = % d\n", var);
   printf ("a = %d, b = %d, c = %d\n", a, b, c);
   printf ("p = %d, q = %d, r = %d, s = %d", p, q, r, s);
}
Output : var = 5
   a = garbage value, b = 6, c = garbage value
   p = 5, q = 5, r = 5, s = 5
```

CONSTANTS

Constants are the fixed values that remain unchanged during the execution of a program and are used in assignment statements. Constants are stored in variables.

To declare any constant, the syntax is

const datatype var_name = value;

In 'C' language, there are five types of constants:

- Character
- Integer
- Real
- String
- Logical

Now, let us discuss each of these in detail.

Character Constants

A character constant consists of a single character, single digit, or a single special symbol enclosed within a pair of single inverted commas. The maximum length of a character constant is one character.

Certain backslash character sequence constants are also used in 'C' to represent special actions. These are called 'C' escape sequence.

\a	-	Audible Bell
\f	-	Formfeed
\r	-	Carriage return
\v	-	Vertical tab
\'	-	Single quote
\?	-	Question mark
\HHH	-	1 to 3 digit Hex Value
\b	-	Backspace
\n	-	Newline (linefeed)
\t	-	Horizontal tab
\\	-	Backslash
\"	-	Double quote
\000	-	1 to 3 Digit Octal Value

Integer Constants

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

In an octal notation, (0) is written immediately before the octal representation. For example: 076, – 076.

In hexadecimal notation, the constant is preceded by 0x. Example: 0x3E, -0x3E.

No commas or blanks are allowed in integer constants.

Real Constants

A real number consists of three parts:

- A sign (+ or 0) preceding the number portion (optional)
- A number portion (representing the base)
- An exponent portion following the number portion (optional). This starts with E or e followed by an integer. The integer may be preceded by a sign.

Example:

Valid representations	Invalid Representations
+ .72	12
+ 72	7.6 E + 2.2
+ 7.6 E + 2	
24.4 e - 5	

String Constants

A string in 'C' is defined as an array of characters. Each string is terminated by the NULL character, which indicates the end of the string. A string constant is denoted by any set of characters included in double-quote marks. The NULL character is automatically appended to the end of the characters in a string constant when they are stored. Within a program, the NULL character is denoted by the escape sequence \0.

For example if we declare

char str [ ] = "UPTEC"

its internal memory representation will be

U	P	T	E	C	\0

Even though there are five characters in the array, internally it is occupying six memory blocks. The extra block being used to store the null character.

Logical Constants

A logical constant can have either a true value or a false value. In 'C' all the non zero values are treated as true values while 0 is treated as false.

Summary

- Data types are of 5 types.
- char data type holds character value.
- int data type holds signed integers.
- float data type holds floating point numbers.
- double data type also holds floating point numbers but their size is twice as that of the float.
- There are also five derived data types, namely, array, pointer, structure, union, and enum.
- Variable are the data items whose value may change during the execution of a program.
- Before use, variables need to be declared.
- Constants are the values which remain fixed during execution of the program.
- The five type of constants are: character, string, logical, numeric, and real.

Review Exercise

Multiple-Choice Questions

1. Which one of the following is not a primary data type in 'C' language?
   (a) char
   (b) int
   (c) union
   (d) float
2. The sizeof( ) operator determines the size of a variable or data type in
   (a) bits
   (b) kilo bits
   (c) nibbles
   (d) bytes
3. The maximum length of a character constant is
   (a) 1 character
   (b) 8 characters
   (c) 4 characters
   (d) none of these
4. Which one of the following special symbol is allowed in a variable name?
   (a) *
   (b) $
   (c) —
   (d) #
5. The escape sequence used to generate an audible beep sound is
   (a) \b
   (b) \a
   (c) \n
   (d) \\

State whether True or False

1. 84.5 E + 5.76 is a valid representation.
2. Const int var = 2,560; will produce an error message on compilation.
3. Values cannot be assigned to the variables at the time of declaration.
4. Integer constants are the whole numbers with one decimal point.
5. 1ac is a valid variable name.

Fill in the Blanks

1. There are _____ primary data types in 'C' language.
2. Void data type takes _____ bytes.
3. To determine the bytes occupied by a variable _____ operator is used.
4. Format specifier %lf is used for _____ data type.
5. An array is _____ data type.

Descriptive Questions

1. Write a program to accept an uppercase character as input and display its ASCII equivalent.
2. Write a program to accept numbers *m, n* as input and produce the output in form m^n.
3. Write a program to accept your name, age, salary as input. Calculate DA (10 percent of salary) and print the output in the form:
   name :
   age :
   salary :
   DA :
4. Write a program to accept a number and print 75% of that number.
5. Write a program to accept two numbers *a, b* and then swap these numbers without using a third variable.
6. What will be the output of the following programs?

(a)
```
main( )
{
    char ch = 300;
    int j = 300;
    printf ("%d %c", ch, i);
    printf ("%d %c", ch, i);
    ch = 37876;
    i = 40000;
    printf ("%c %d", ch, i);
    printf ("%c %d", ch, i);
}
```

(b)
```
main( )
{
    int h = 15;
    const int p = 5;
    float k = 22.7;
```

```
        printf ("\n h = %d    k = %d",    h, k);
        p = 7;
        printf ("\n p = %d", d);
    }
(c) main( )
    {
        int a = 15;
        char ch = 'B';
        float b = 3.14;
        printf ("%d %c %f \n", a, ch, b);
        printf ("%d %f %f \n", a, ch, b);
        printf ("%f %c %f \n", a, ch, b);
    }
(d) main( )
    {
        int a = 32768, b = -1;
        printf ("\n a = %d", a);
        print ("\n b = %d", b);
    }
```

CHAPTER

7

Operators, Type Modifiers and Expressions

Key Features

- Operators
- Type Modifiers
- Expressions
- Type Definitions Using typedef

An operator specifies an operation to be performed that produces a value. 'C' includes a large number of operators. Variables and constants joined by various operators form an expression. Some operators require one operand, while others require two or three. Type modifiers enable 'C' programmers to invent whatever data type they need. As seen earlier, the primary data types could be of five varieties: character, integer, float, double and void. However, any number of data types can be derived from these five types.

This chapter will discuss in detail about the various categories of operators, expressions, and type modifiers.

OPERATORS

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations on data stored in variables. The variables that are operated are termed as operands.

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

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Bitwise operators
- Special operators

Now, let us discuss each category in detail.

Arithmetic Operators

'C' provides all the basic arithmetic operators. There are five arithmetic operators in 'C' language as given in Table 7.1.

Table 7.1 Arithmetic operators in 'C' language

Operator	*Purpose*
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

The division operator (/) requires the second operand as non zero, though the operands need not be integers.

The operator (%) is known as a modulus operator. It produces the remainder after the division of the two operands. The second operand must be non zero.

Example:

Suppose x and y have values 10 and –4, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	*Value*
$x + y$	6
$x - y$	14
$x * y$	–40
x/y	–2
$x \% y$	2

If we had assigned x value of –10 and y had been assigned 4, then the value of x/y would still be –2 but the value of $x \% y$ would be –2.

All other operators work in their normal way.

Integer Arithmetic When both the operands are integers, the expression is called an integer expression and the operation is known as integer arithmetic. Integer arithmetic always yields an integer value.

Example: if $a = 25$, $b = 4$
then $a + b = 29$
$a - b = 21$
$a * b = 100$
$a/b = 6$ (decimal parts truncated)
$a \% b = 1$

Real Arithmetic When both the operands are real numbers then the operation is called Real Arithmetic. Modulus operator is not permissible for real arithmetic. Floating point values are rounded to the number of permissible significant digits. The final value is an approximation of the correct result.

Mixed Mode Arithmetic When one operand is real and the other is an integer, the expression is called mixed mode arithmetic expression. If either operand is of the real type only then the real operation is performed and the result is always of the real type.

Thus 15/10.0 = 1.5
while 15/10 = 1

Relational Operators

A relational operator is used to compare two operands to see whether they are equal to each other, unequal, or one is greater or lesser than the other.

The operands can be variable, constants, or expression and the result is a numerical value. There are six relational operators.

= =	equality
! =	not equal to
<	less than
< =	less than or equal to
>	greater than
> =	greater than or equal to

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

ae-1 relational operator ae-2

ae-1 and ae-2 are arithmetic expressions, which may be simple constants, variables, or a combination of these. The value of a relational operator is either 1 or 0. If the relation is true, the result is 1, otherwise it is 0.

Example:

Expressions	**Result**
4.5 < = 10	True
4.5 < – 10	False
–35 > = 0	False
10 < 7 + 5	True

Logical Operators

Logical operators are used to combine two or more relational expressions. 'C' provides three logical operators.

Operator	**Meaning**
&&	Logical AND
¦¦	Logical OR
!	Logical NOT

The result of Logical AND will be true only if both the operands are true. While the result of a Logical OR operation will be true if either operand is true. Logical NOT (!) is used to reverse the value of the expression.

The expression which combines two or more relational expressions is termed as a logical expression or a compound relational expression which yields either 1 or 0.

Examples:

- if (age > 50 && weight < 80)
- if (a < 0 ¦¦ ch = = 'a')
- if (! (a < 0))

Assignment Operators

Assignment operators are used to assign the result of an expression to a variable. The most commonly used assignment operator is (=).

An expression with assignment operator is of the following form:

identifier = expression

```
#include <stdio.h>
main( )
{
    int i;
    i = 5;
    printf ("%d", i);
    i = i + 10;
    printf ("\n % d", i);
}
```

Output will be : 5
15

In this program $i = i + 10$; is an assignment expression which assigns the value of $i + 10$ to i.

Expressions like $i = i + 10$; , $i = i - 5$;, $i = i * 2$;, $i = i/6$; and $i = i * 4$; can be rewritten using shorthand assignment operators.

The shorthand assignment operators have the following syntax

v op = exp;

This is equivalent to v = v op exp;

Example: $i = i + 5$; is equivalent to $i + = 5$;

$i = i * (y + 1)$; is equivalent to $i * = (y + 1)$;

The advantage of using assignment operators are:

- The statement is more efficient and easier to read.
- What appears on the left hand side of the assignment operator need not be repeated and therefore, it becomes easier to write.

Example:

```
#include <stdio.h>
main( )
{
```

```
    int a;
    a = 2;
    while (a < 100)
    {
        printf ("%d \n", a);
        a* = a;          /* equivalent to a = a * a */
    }
}
```

Output : 2
4
16

Increment and Decrement Operators

'C' has two very useful operators ++ and – – called increment and decrement operators respectively. These are generally not found in other languages. These operators are called **unary operators** as they require only one operand. This operand should necessarily be variable and not a constant.

The increment operator (++) adds one to the operand while the decrement operator (– –) subtracts one from the operand.

These operators may be used either before or after the operand. When they are used before the operand, it is termed as a *prefix*, where as when used after the operand, they are termed as a *postfix* where operators.

Example:

```
int i = 5;
i ++;
++ i;
– – i;
i – –;
```

When used in an isolated 'C' statement, both prefix and postfix operators have the same effect. But when they are used in expressions, each of them have a different effect.

In expressions, the postfix operator has the effect of "use-then-change" while the prefix operator has the effect of "change-then-use".

Example:

b = *a*++; This is a postfix increment expression. In the expression

firstly *b* = *a*; is performed then *a* = *a* + 1; will be executed; while in prefix increment expression such as

b = – – *a*;

firstly *a* = *a* – 1; is performed then *b* = *a*; will be executed.

Example:

```
#include <stdio.h>
main( )
```

```
{
    int a = 10, b = 0;
    a++;
    printf ("\n a = %d", a);
    b = ++a;
    printf ("\n a = % d, b = % d", a, b);
b = a++;
printf ("\n a = % d, b = % d", a, b);
}
```

Output:
```
a = 11
a = 12   b = 12
a = 13   b = 12
```

Conditional or Ternary Operator

A ternary operator is one which contains three operands. The only ternary operator available in 'C' language is conditional operator pair "? :". It has following syntax:

```
exp1 ? exp2 : exp3 ;
```

This operator works as follows: exp1 is evaluated first. If the result is true then exp2 is executed otherwise exp3 is executed.

Example:

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

in this expression the value of *b* will be assigned to *x*.

Bitwise Operators

'C' provides six operators for bit manipulation which may only be applied to integral operands, that is , char, short, int, and long, whether signed or unsigned.

The six operators are;

&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's compliment (unary)

The bitwise AND operator & is often used to mask-off some set of bits; for example,

```
n = n & 0177;
```

sets to zero all but the low-order 7 bits of *n*.

The bitwise OR operator | is used to turn bits on:

$X = X$ | SET_ON;

sets to one in X the bits that are set to one in SET_ON.

The bitwise exclusive OR operator ^ sets a one in each bit position where its operands have different bits, and zero where they are the same.

The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be positive. Thus if $x = 7$ then $x << 2$ shifts the value of x left by two positions, filling vacated bits with zero; that gives us 28. Right shifting an unsigned quantity always fills vacated bits with zero. Right shifting a signed quantity will fill with sign bits ("arithmetic shift") on some machines and with 0-bits ("logical shift") on others.

The unary operator ~ yields the one's complement of an integer, that is, it converts each 1 bit into a 0-bit and vice versa. For example if $x = (101)_2 = (5)_{10}$ then $\sim x = (010)_2 = (2)_{10}$.

Special Operators

'C' language supports some special operators such as comma operator, sizeof operator, pointer operators (& and *), and member selection operators (. and ->). Pointer operators will be discussed while introducing pointers, while the member selection operator will be discussed with structures and union. Let us discuss the comma operator and the size of operator.

Comma Operator This operator is used to link related expressions together.

Example:

```
int val, x, y;
val = (x = 10, y = 5, x + y);
```

it first assigns 10 to x then 5 to y finally sum $x + y$ to val.

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

Example:

```
int n;
n = sizeof (int);
printf ("%d", n);
```

Output: $n = 2$.

Operator Precedence

Precedence defines the sequence in which operators are to be applied on the operands, while evaluating the expressions involving more than one operator. Operators of the same precedence are evaluated from left to right or right to left, depending upon the level. This is known as the associativity property of an operator.

The various operators along with their precedence and associativity are given in Table 7.2.

Table 7.2 Operator precedence and associativtiy

Description	*Operators*	*Associativity*
Function expression	( )	L → R
Array expression	[ ]	L → R
Structure operator	→	L → R
Structure operator	.	L → R
Unary Minus	–	R → L
Increment / Decrement	+ + - -	R → L
One's complement	~	R → L
Negation	!	R → L
Address of	&	R → L
Value at address	*	R → L
Type cast	(type)	R → L
Size in bytes	sizeof	R → L
Multiplication	*	L → R
Division	/	L → R
Modulus	%	L → R
Addition	+	L → R
Subtraction	–	L → R
Left shift	<<	L → R
Right shift	>>	L → R
Less than	<	L → R
Less than or equal to	< =	L → R
Greater than	>	L → R
Greater than or equal to	> =	L → R
Equal to	= =	L → R
Not equal to	! =	L → R
Bitwise AND	&	L → R
Bitwise XOR	^	L → R
Bitwise OR	|	L → R
Logical AND	&&	L → R
Logical OR	||	L → R
Conditional	?:	R ← L
Assignment	=	R ← L
	* = / = % =	R ← L
	+ = – = & =	R ← L
	^ = | =	R ← L
	< < = > > =	R ← L
Comma	,	R ← L

TYPE MODIFIERS

The basic data types may have modifiers preceding them to indicate special properties of the objects being declared. These modifiers change the meaning of the basic data types to suit specific needs.

These modifiers are unsigned, signed, long and short. It is also possible to give these modifiers in combination. For example, **unsigned long int**.

Modifiers for char Data Type

char data type can be qualified as either signed or unsigned, both occupying one byte each, but have different ranges. A signed char is the same as an ordinary char and ranges from –128 to +127; whereas an unsigned **char** has ASCII value range from 0 to 255. By default **char** is unsigned.

Example:

```
main ( )
{
    char ch = 291 ;
    printf ("%d\t%c\n", ch, ch) ;
}
```

Output : 35 #

Here ch has been defined as a char, and a char cannot take a value bigger than +128. That is why the assigned value of ch, 291, is considered to be 35 (291-128-128).

Modifiers for int Data Type

Integer quantities can be defined as *short int, long int*, or *unsigned int*. Short int occupies 2 bytes of space, whereas long int occupies 4 bytes. A signed int has the same memory requirements as for an unsigned int (or a short int or a long int), the leftmost bit is reserved for the sign. With an unsigned int, all the bits are used to represent the numerical value. The unsigned qualifier can also be applied to another qualified int. For example, unsigned short int or unsigned long int. By default, the modifier assumed with integers is signed.

Modifiers for double and float Data Type

A modifier long is used with double data type but not with float. Long double occupies 10 bytes of memory space.

The following Table 7.3 explains the data types along with their range, size in bytes and the format specifier.

Table 7.3 Data types : Their range and format specifiers

Data Type	*Range*	*Bytes*	*Format*
signed char	–128 to +127	1	%c
unsigned char	0 to 255	1	%c
short signed int	–32768 to 32767	2	%d
short unsigned int	0 to 65535	2	%u

(*Contd*)

Data Type	*Range*	*Bytes*	*Format*
long signed int	–2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	–3.4e38 to 3.4e38	4	%f
double	–1.7e308 to + 1.7e 308	8	%lf
long double	–1.7e4932 to 1.7e4932	10	%Lf

More Type Modifiers

const This modifier should precede the type and name of the variable like const int *a* = 5 ;. Once this variable has been assigned a value during the definition of the variable, the value cannot be changed by the program during the execution.

Example: const int SIZE = 50;

The purpose of **const** is to announce objects that may be placed in the read-only memory, and perhaps to increase opportunities for optimization.

volatile This modifier has the same syntax as const. The purpose of a volatile is to force an implementation to suppress optimization that could otherwise occur. The value of the variable will get changed by some agency without the program assigning a value to it explicitly.

Example: volatile int date;

Variable passed as an argument to a clock function of the operating system: The value stored in the systems keep on changing in real time. The value of a variable declared as volatile can be modified by its own program also. If we wish that the value must not be modified by the program while it may be altered by some other process, we may declare the variable as both const and volatile.

EXPRESSIONS

An expression is a combination of variables, constants, and operators arranged according to the syntax of the language. Some examples of the expressions are:

$c = (m + n) * (a - b)$;

temp = $(a + b + c) / (d - c)$;

Such a statement is of the form

variable = Expression;

Evaluation of Expression

The expression on the R.H.S is evaluated first, then the value is assigned to the variable. But all the relevant variables must be assigned the values before evaluation of the expression.

By using the assignment statement we can evaluate an expression as

variable = expression;

Example: temp = ((*f* * cos (*x*) / sin (*y*)) + (*g* * sin (*x*) / cos (*y*)))

In the example, all the mathematical operations and the trigonometirc functions are evaluated first, and then the final value obtained is assigned to the variable temp.

Type Conversion in Expressions

Automatic Type Conversion If the operands are of different types, the lower type is automatically converted to the higher type before the operation proceeds. The result is of the higher type. Table 7.4 given below lists the sequence of rules that are applied while evaluating expressions.

Table 7.4 Type conversion during execution of experssions

Op-1	*Op-2*	*Result*
long double	any	long double
double	any	double
float	any	float
unsigned long int	any	unsigned long int
long int	any	longint
unsigned int	any	unsigned int

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning value to it.

However, the following changes are introduced during the final assignment:

- float to int causes truncation of the fractional part.
- double to float causes rounding of digits.
- long int to int causes dropping of the excess higher order bits.

Casting a Value Casting a value is forcing a type conversion in a way that is different from the automatic conversion. The process is called type cast. The general form of casting is

(type_desired) exp;

where type_desired is any of the standard 'C' data types and exp is constant, variable or an expression.

Example:

```
#include <stdio.h>
main( )
{
    int production, sale;
    float ratio;
    ratio = (float)production / sale;
    printf ("%f\n",ratio);
}
```

In the above program, the expression ratio = (float) production/sale; the value of variable production is converted to float, otherwise the decimal part of the result of division would be lost and the ratio would represent a wrong figure.

- Notice how a cast affects the value of the following example:

```
#include <stdio.h>
main( )
{
    int a, b ;
    float c, d ;
    a = 1 ;
    c = 3.1415 ;
    b = (int) c;   /* b will receive the value 3 */
    d = (float) a / (float) b ;   /* d = 0.333 */
    printf ("\n %d \n%f ", b, d);
}
```

Output:

The output of the above program will be

```
3
.333
```

TYPE DEFINITIONS USING typedef

'C' allows us to rename data types via the typedef statement. Once a type is defined, it can be used in the same manner as the standard 'C' types.

The format of the typedef statement is:

```
typedef type new_type_name ;
```

Example: typedef int units;

```
units bat1, bat2; /* this statement is equivalent to int bat1, bat2*/
```

Now units can be used to create variables similar to the type of int variables.

Summary

- Operator is a symbol which tells the computer to perform some operation.
- There are five arithmetic operators in 'C' : +, -, *, /, %.
- Relational operators are used to compare two operands.
- To combine two or more relational operators, logical operators &&, ¦¦, ! are used.
- To assign values to an expression, assignment operators are used.
- For increment or decrement of the value of a variable by 1, ++, -- operators are used. These are unary operators.
- If there is a need of a logical decision, If..then..else constructs, conditional operators are used.
- For operating on the bits, bitwise operators are used.

- 'C' also provides special operators like comma, sizeof.
- All these operators have different precedence over each other.
- Type modifiers assign special properties to the basic data type.
- Expression is a combination of variable, constants, and operators.
- Typedef is used to create new data types.

Review Exercise

Multiple-Choice Questions

1. Which one of the following is not a logical operator?
   (a) &&
   (b) ||
   (c) *
   (d) !
2. How many operands does a ternary operator contain?
   (a) 3
   (b) 4
   (c) 2
   (d) none of these
3. Renaming an existing data type is done by
   (a) rename
   (b) const
   (c) change
   (d) typedef
4. The statement *a*++; is equivalent to
   (a) *a* = *a* + *a*;
   (b) *a* = *a* + 1;
   (c) *a* = *a* + 2;
   (d) none of these
5. The sizeof operator returns the size of a data type expressed in
   (a) bits
   (b) nibbles
   (c) words
   (d) bytes

State whether True or False

1. Logical operators are used to manipulate data at bit level.
2. Relational operators have higher precedence than the logical operators.
3. Left Shift operator rotates the bits on the left and places them to the right.
4. The operators ++ & - - are binary operators.
5. Bitwise operators can manipulate individual bits of a float.

Fill in the Blanks

1. Integer arithmetic always yields an _____ value.
2. The assignment operator assigns the result of an expression to a _____ .
3. The bitwise operators are used to manipulate data at _____ level.
4. When one operand is of read type and other is an integer, the expression is called _____ arithmetic expression.
5. The prefix and postfix operators have _____ effect when they are used in expressions.

Descriptive Questions

1. What is meant by operator precedence? What are the relative precedences of the arithmetic operators?
2. How can the value of an expression be converted to a different data type? What is it called?
3. What is an expression? What are its components?
4. Describe two different ways to utilize the increment and decrement operators. What is the difference between them?
5. Describe the use of the conditional operator to form conditional expressions.
6. What will be the output of the following programs?

(a)
```
main( )
{
    int i = – 3;
    i = – i – i + i;
    printf ("%d", i);
}
```

(b)
```
main( )
{
    int i, j, k;
    i = 10;
    j = 5;
    k = i > j;
    printf ("%d", k);
}
```

(c)
```
main( )
{
    int p, q, r;
    p = 7, q = 3;
    r = (p > q) && (q < p);
    printf ("%d", r);
}
```

(d)
```
main( )
{
    unsigned char a;
    a = OxFF + 1;
    printf ("%d", a);
}
```

(e)
```
main( )
{
    int a, b, c, d, e;
    a = 5;
    b = 6;
    c = 12;
    d = 11;
    e = (a ! = b) ? (e < = (! d) ? a : b) : c;
    printf ("e = % d", e);
}
```

(f)
```
main( )
{
    float a = 1; b;
    int m = 3, n = 5;
    b = (a ? m : n) / 2.0;
    printf ("%f ", b);
}
```

(g)
```
main( )
{
    int a, b, c, d;
    a = 4;
    b = + + a;
    c = A - -;
    d = - - A;
    a + = 1 - 2 * 2;
    printf ("a = %d, b = %d, c = %d, d = %d ", a, b, c, d );
}
```

(h)
```
main( )
{
    int x, y, z;
    x = (y = 5, z = y + 2, y + z);
    x = y = 5; z = x = 9;
    printf ("x = % d, y = % d, z = %d", x, y, z);
}
```

(i)
```
main( )
{
    int i = -3, j = 2, k = 0, m;
    m = ++ i && ++k && ++k;
    printf ("%d, %d, %d", i, j, k);
}
```

(j)
```
main( )
{
    int i = -3, j = 2, k = 0, m;
    m = ++j && ++i | | ++ k;
    printf ("\n % d, % d, % d ", i, j, k, m);
}
```

CHAPTER

8

Basic Input/Output

Key Features

- Introduction to Input/Output
- Console I/O Functions
- Formatted Console I/O Functions
- Unformatted Console I/O Functions

Every 'C' program performs three main functions:

- It accepts data as input
- Processing of the data
- Produces output

In 'C' language there are several input-output functions. These functions collectively form a library called **io.h**.

In this chapter, input-output operations along with their types are discussed.

INTRODUCTION TO INPUT/OUTPUT

Input refers to accepting of data while **output** refers to the presentation of data. Normally the data is accepted from the keyboard and is outputted on to the screen.

'C' language has a series of standard input/output (I/O) functions. Such I/O functions together form a library named **stdio.h**. Irrespective of the version of 'C' language, the user will have access to all such library functions. These library functions are classified into three broad categories.

- Console I/O functions : functions which accept input from the keyboard and produce output on the screen.
- Disk I/O functions : functions which perform I/O operations on secondary storage devices like floppy disks or hard disks.
- Port I/O functions : functions which perform I/O operations on various ports like printer port, and mouse port.

CONSOLE I/O FUNCTIONS

Console I/O refers to the operations that occur on the keyboard and the screen of the computer. Console I/O functions can be further classified as:

- Formatted Console I/O
- Unformatted Console I/O

The basic difference between formatted and unformatted I/O functions is that the formatted I/O functions allow input and output to be arranged as per the requirement of the user.

The following Figure 8.1, describes the various I/O functions.

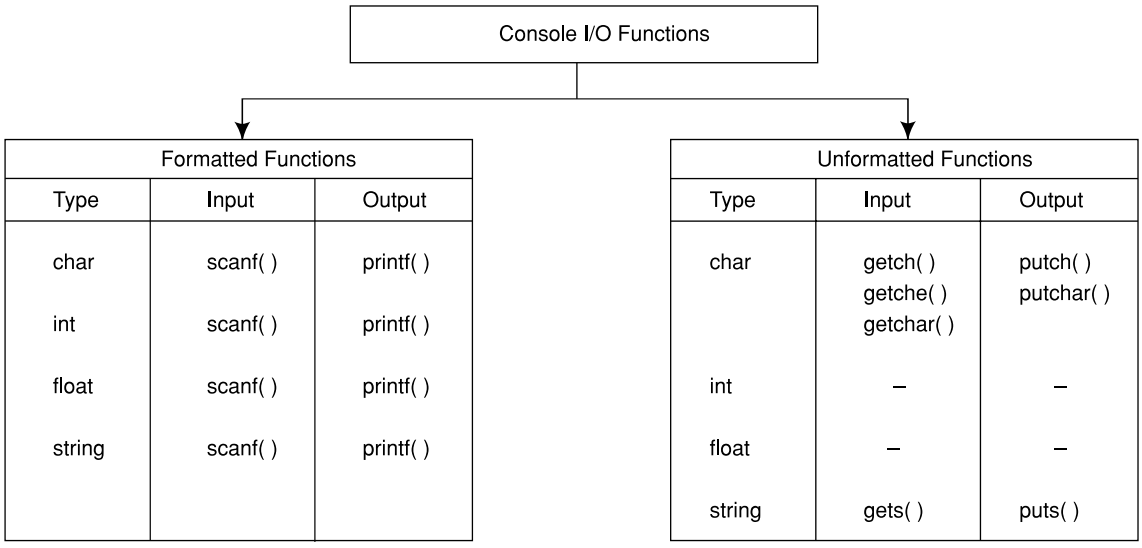


Fig. 8.1 *I/O functions*

FORMATTED CONSOLE I/O FUNCTIONS

Formatted Console I/O functions accept or present the data in a particular format. The standard 'C' library consists of two functions that perform output and input, namely, printf( ) and scanf( ). These functions can format the information under the user's direction.

Formatted Output

It is highly desirable that the outputs are presented in such a way that they are understandable and in an easy-to-use form.

The **printf( )** statement provides certain features through which the screen output is effectively controlled. The general form of printf( ) function is:

printf ("Control String", arg1, arg2. . .);

Control string may contain:

- characters that are simply printed as they are.
- conversion specifications that begin with a % sign.
- escape sequences that begin with \ sign.

The control string indicates how many arguments follow and what their types are. The arguments arg1, arg2. . . are the variables whose values are formatted and printed according to specifications of the control string. The arguments must match in number, order, and type with the format specifications.

Example:

```
main( )
{
     int arg = 346;
```

```
        float per = 69.2;
        printf ("Average = % d \n percentage = % f", arg, per);
    }
Output: Average = 346
        Percentage = 69.2
```

Conversion Specifications The conversion specifications are used to provide the type and size of the data. Each conversion specification must begin with %.

In the above example %d and %f are the conversion characters. The general form of the conversion specifier is

% fws fx

where fws = field width specifier
fx = format specifier

The field width specifier tells printf( ) how many columns on the screen should be used while printing a value.

Example: %7d tells to print the variable as a decimal integer in the field of 7 columns.

If we include a minus sign in conversion specification (% - 7*d*), it means that the left justification is desired and the value will be padded with blanks on the right.

Table 8.1 describes the conversion characters that can be used with the printf( ) function.

Table 8.1 Data types and their conversion characters

Data Type		*Conversion Chracter*
Integer	short signed	%d or % i
	short unsigned	% u
	long signed	% ld
	long unsigned	% lu
	unsigned hexadecimal	% x
	unsigned octal	% O
Real	float	% f
	double	% lf
Character	signed character	% c
	unsigned character	% c
String		% s

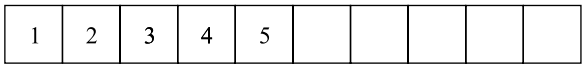
Escape Sequences The backslash symbol (\) is considered as an escape character because it causes an escape from the normal interpretation of a string, so that the next character is recognized as the one that has a special meaning. The various escape sequences are listed below in Table 8.2.

Table 8.2 Escape sequences in 'C' language

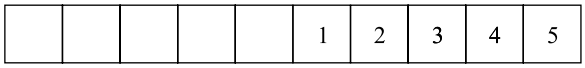
Escape Sequence	*Purpose*	*Escape Sequence*	*Purpose*
\n	New line	\t	Tab
\b	Backspace	\r	Carriage return
\f	Formfeed	\a	Alert
\'	Single quote	\"	Double quote
		\\	Backslash

Output of Integer Numbers The format specification for printing an integer number is %wd where 'w' specifies a minimum width for the output. The number is written right justified in the given field width.

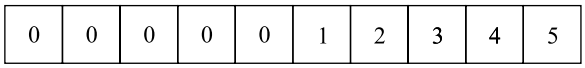
Example:



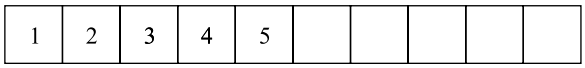
printf ("*%d*" , 12345);



printf ("%10*d*" , 12345);



printf ("% 010*d*" , 12345);



printf ("% -10*d*" , 12345);

Output of Real Numbers The real number is displayed in decimal notation with format specification % w.pf where 'w' is the integer which represents the minimum number of positions that are to be used and 'p' indicates that in the total width, how many numbers will be placed after the decimal point.

Example:

printf ("%7.4 *f*" , 98.7654);

printf ("%7.2" , 98.7654);

print ("% *f* " , 98.7654);

9	8	·	7	6	5	4

Printing of Single Character A single character can be displayed in a desired position using format %wc. By default, the character will be displayed right justified in a field of 'w' columns. To make it left justified, minus sign is placed in the format specification.

Printing of Strings The format specification for outputting strings is similar to that of real numbers. It is of the form %w.ps where 'w' specifies the field width for display and 'p' specifies the number of characters to be displayed. The display is right justified.

Example: Suppose the string to be printed is "NEW DELHI 110001"
The total length is 16 characters (including blanks).

specification	output																		
% s	N	E	W		D	E	L	H	I		1	1	0	0	0	1			
% 20S					N	E	W		D	E	L	H	I		1	1	0	0	0	1
% 20.75											N	E	W		D	E	L	H	I	

Formatted Input

Formatted input refers to an input data that has been arranged in a particular format. For the formatted input we use the function scanf( ).

scanf( ) function scanf( ) function allows us to read the formatted data and automatically convert numeric information into integers and float. The general form of scanf( ) is

scanf ("control string", arg1, arg2, ...);

Control string specifies the field format in which data is to be entered and the arguments arg1, arg2 - - - - - specify the ADDRESS OF LOCATION where value is to be stored. Control string and arguments are separated by commas.

Table 8.3 given below contains a list of format specifiers used to read the inputs of various data types.

Table 8.3 Format specifiers

Code	*Meaning*	*Code*	*Meaning*
% c	Read a single character	% d	Read a decimal integer
% ld	Read a long integer	% i	Read a decimal integer
% e	Read a floating point number	% f	Read a floating point number
% h	Read a short integer	% 0	Read an octal no.
% s	Read a string	% x	Read a hexadecimal number
% p	Read a pointer	% n	Read an integer value equal to the no. of character read so far

Input of Integer Numbers The format specification for reading an integer number is % wd where (%) sign indicates conversion specification, 'w' is the integer number for field width specification and 'd' indicates that the number is to be read in integer mode.

Example: scanf ("% 2*d* % 5*d*", &*n*1, &*n*2);

An input field may be skipped by specifying '&' in place of field width.

Example: scanf ("% 2*d* % * *d* % 6*d*", &*n*1, &*n*2);

Input of Character Strings % ws or % wc can be used as the specification for reading character strings. The specifier %s terminates reading a string at the encounter of a blank space. Some versions of scanf( ) support the following conversion specification for strings:

% [characters] and % [^characters]

The specification % [characters] means that only the characters specified within brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character.

The specification % [^character] does exactly the reverse, that is, characters specified after circumflex (^) are not permitted.

UNFORMATTED CONSOLE I/O FUNCTION

Unformatted console I/O functions cannot control the format of reading and writing the data. All the unformatted console I/O functions are defined in stdio.h header file.

Character Input Functions

getch( ) and getche( ) getch( ) function reads a character from the keyboard, does not echo the character on the screen and returns the character pressed. Where as the getche( ) function reads the character from the keyboard, echoes the character, and returns the character pressed.

It is required to add conio.h file on the program before using this function.

Example: ch = getch( ); or ch = getche( );

getchar( ) This is a character input function. It accepts the input until the carriage return is entered.

Example:

```
#include <conio.h>
#include <stdio.h>
main( )
{
    char c;
    c = getchar( );
    if (c = = 'y' || c = = 'Y')
    printf ("c is a character \n");
}
```

The above program will accept any character for the user and if it is *y* or *Y*, the printf statement will be executed.

Character Output Function

The *putchar( )* function is used for printing a character to a screen at the current cursor location.

syntax : putchar (variable_name);

Note: The variable must be of the character type.

Character String Input Function

The *gets( )* function is used to read a character entered on the keyboard and places it at the address pointed to by its character pointer argument. Characters are entered until the enter key is pressed.

syntax : char * gets (char *a);

Here *a* is the character array.

Character String Output Function

The *puts( )* function writes its string argument to the screen followed by the new line.

syntax : char * puts (const char * *a*);

The puts( ) function takes less space than printf( ). It is faster than printf( ), does not output numbers or do format conversions as puts( ) outputs the character string only.

Example:

```
#include <stdio.h>
#include <conio.h>
main( )
{
    char str [50]
    gets (str);
    puts (str);
}
```

The output will only display the string that has been supplied by the user before pressing the enter keys.

Example Programs

Program 1 : Write a program to exchange the values of two variables in 'C'.

Solution

```
/* program to exchange values of two variables */
#include <stdio.h> /* has prototypes of functions scanf() &
printf() */
main()
{
   int a,b,t ;
   printf ("Enter two values \n");
   scanf ("%d%d", &a,&b); /* reads in the values in variables
a and b */
```

```
            t = a ;
            a = b ; /* exchanges the values of a and b using a
temporary variable t*/
            b = t ;
            printf ("The values of a and b are \n");
            printf ("%d%d", a,b);
         }
```

Program 2 : Write a program in 'C' to check whether a given number is odd or even using the conditional operator.

Solution

```
         /* program to check whether a number is odd or even */
         #include <stdio.h>
         main ( )
         {
            int number;
            printf ("Enter the number");
            scanf ("%d", &number);
            number %2? printf ("Number is odd"):printf ("Number is
even");
         }
```

Summary

- All the I/O operations are done through function calls.
- I/O statements are of three broad types, namely, console I/O statements, disk I/O statements, and port I/O statements.
- Console I/O can be formatted or unformatted.
- Formatted I/O operations are done by printf( ), scanf( ) function.
- Unformatted I/O operations are performed by getch( ), getche( ), getchar( ), gets( ), puts( ), and putchar( ) functions.
- Conversion specifiers converts the form of input or output.

Review Exercise

Multiple-Choice Questions

1. Functions which accept the input from the keyboard and give the output on the screen are called
   (a) disk I/O functions
   (b) console I/O functions

(c) system I/O functions
(d) none of these

2. Which of the following is not an unformatted console I/O function?
   (a) gets( )
   (b) puts( )
   (c) printf( )
   (d) getch( )
3. The control character used to handle strings is
   (a) \t
   (b) \a
   (c) \st
   (d) \d
4. The getch( ) function does not
   (a) echo the character on the screen
   (b) terminate by itself
   (c) read a character
   (d) none of these
5. The putchar( ) function is used to
   (a) count the number of character in a string
   (b) print a single character on the screen
   (c) print the entire string on the screen
   (d) none of these

State whether True or False

1. Can we specify the variable field width in scanf( ) format string?
2. To tackle double in printf( ), % f is used while in scanf( ), % lf is used.
3. Messages and headings can be printed by using character strings directly in printf( ).
4. By placing the minus sign before the integer w in % wc we can make the display right justified.
5. When searching for a value, scanf( ) ignores line boundries.

Fill in the Blanks

1. _____ I/O functions can be formatted or unformatted.
2. _____ function is used to print a single character at the current cursor position.
3. _____ I/O operators are done by printf( ) and scanf( ) functions.
4. % 10d in the format specification will align the output towards _____ of the field width of 10 characters.
5. The basic I/O functions of 'C' language are defined in _____ header file.

Descriptive Questions

1. What are the commonly used input/output functions in 'C'?
2. What is the purpose of the getchar function? How is it used within a 'C' program?
3. How is the scanf function used within a 'C' program? Compare it with the getchar function.
4. How can the getchar function be used to read multicharacter strings?

5. What is the purpose of the control string in a scanf function?
6. What will be the output of the following programs:

```
(a) main( )
    {
       int a = 150;
       printf ("% 1d", a);
    }
(b) main( )
    {
       char ch;
       int i;
       scanf ("% c", & i);
       scanf ("% d", & ch);
       printf ("% c, % d", ch, i);
    }
(c) main( )
    {
       int n = 5;
       printf("\nn = % d", n, n);
    }
(d) main( )
    {
       printf("O\tLEVEL\CC");
    }
(e) main( )
    {
       puts ("Hello. . . ", "Goodbye");
       printf ("% s, % s", "Hello. . . ", "GoodBye");
       }
```

7. Assume that the string "INPUTOUTPUT" is stored in a variable. Write a program to read the string and display the output in the following formats:
   (a) INPUT OUTPUT
   (b) INPUT
       OUTPUT
   (c) I.O.
8. Write a program to read the given numbers from the keyboard, round them off to their nearest integer and print out the results in the integer format.

   37.5 50.12 −25.73 −46.75

CHAPTER

9 Control Constructs

Key Features

- Control Statements
- Conditional Statements
- Loops in C
- Infinite Loops
- Nested Loops
- The break Statement
- The continue Statement
- The exit( ) Function
- The goto Statement

We have seen that in a 'C' program, the statements are executed sequentially. There is no branching or repetition of the previous statement. But in a number of situations, it may occur that certain decisions need to be made and some statements might be executed repeatedly. To deal with these situations, control constructs are provided in 'C'.

In this chapter, we will discuss the various control constructs available in 'C'.

CONTROL STATEMENTS

Control statements enable us to specify the order in which the various instructions in a program are to be executed by the computer. They determine the flow of control in a program.

There are four types of control statements in 'C'. They are:

- *Sequence Control Statements*
- *Decision Control Statements or Conditional Statement*
- *Case Control Statement*
- *Repetition or Loop Control Statements*

Sequence Control Statements ensure that the instructions in the program are executed in the same order in which they appear in the program.

Decision and Case Control Statements allow the computer to take decisions as to which statement is to be executed next.

The Loop Control Statement helps the computer to execute a group of statements repeatedly.

CONDITIONAL STATEMENTS

C has two major decision-making statements.

- If_else Statement
- Switch Statement

if_else Statement

The if_else statement is a powerful decision-making tool. It allows the computer to evaluate the expression. Depending on whether the value of the expression is "True" or "False", certain groups of statements are executed.

The syntax of if_else statement is:

```
if (condition is true)
    statement 1;
else
    statement 2;
```

The condition following the keyword is always enclosed in parenthesis. If the *condition* is true, the statement in the *then* part is executed, that is, statement 1 otherwise statement 2 in **else** part is executed. There may be a number of statements in the **then** and **else** part.

Example:

```
/* magic number program * /
main( )
{
   int magic = 223;
   int guess;
   printf ("Enter your guess for the number: \n");
   scanf ("%d", &guess);
   if (guess = = magic)
   printf ("\n Congratulation ! Right guess");
   else
   printf ("\n wrong guess");
}
```

Output:

```
   Enter your guess for the number : 200
   Wrong guess
```

Nesting of if-else Statement When a series of decisions are involved in the statement, we may have to use more than one if_else statement in nested form. This can be described by following the flowchart as in Figure 9.1:

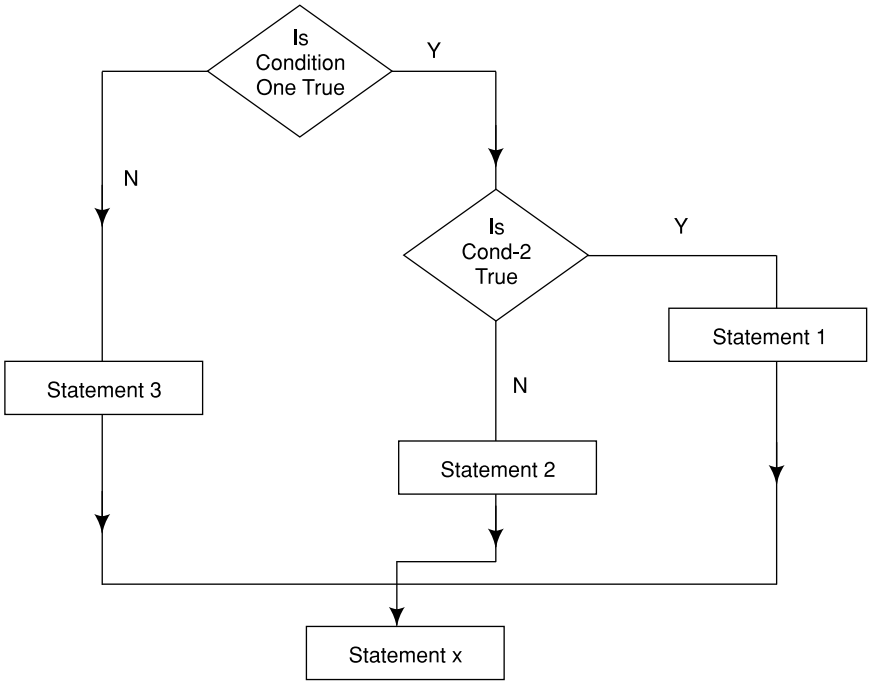


Fig. 9.1 *Flow chart of if_else statement in nested form*

Example:

```
/ * finding the largest of 3 numbers * /
main( )
{
int a = 5, b = 2, c = 7;
if (a > b)
{
   if (a > c)
   printf ("a is greatest");
   else
   printf ("c is greatest");
}
else
{
   if (b > c) printf ("b is greatest");
```

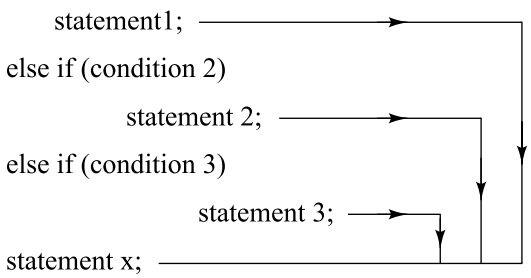
```
        else printf ("c is greatest");
        }
    }
```

Output:

```
    c is greatest.
```

else if Ladder There is another way of putting **ifs** together when multipath decisions are involved. A multipath decision is a chain of **ifs** in which the statement associated with each **else** is an **if**.

It takes the following general form:



The condition in **elseif** ladder is evaluated from the top of the ladder, downwards. As soon as the true condition is found, associated statement is executed and control is transferred to statement *x*.

Example:

```
main( )
{
   int unit, custnum;
   float charges;
   printf ("Enter Customer No. and Units Consumered : \n");
   scanf ("%d%d", & custnum, &unit);
   if (unit < = 200)
      charges = 0.5 *units;
   else if (units < = 400)
      charges = 100 + 0.65* (units - 200);
   else if (units < = 600)
      charges = 230 + 0.8 * (units - 600);
   printf ("\n \n Customer No : % charges : %0.2f \n"
Custnum, Charges);
}
```

The switch Statement

In a multiway decision construct, the complexity of the program increases with the increase in the number of alternatives. The program becomes difficult to read and follow. 'C' has a built-in multiway decision statement known as a **switch**. The switch statement tests the value of a given variable or expression against a list of case values and when a match is found, a block of statements associated with that case is executed.

The general form is:

```
switch (expression)
{
   case constant_1:
         statements;
   case constant_2:
         statements;
      default:
         statements;
   }
```

First, the integer expression following the keyword switch is evaluated. The value it gives is then matched, one by one, against the constant values that follow the case statements. Whenever a match is formed, the program executes the statements following the case, and all subsequent cases and default statements as well.

If no match is found with any of the case statements, only the statements following the default are executed. This default is optional and if not present, no action takes place if all the matches fail.

```
/* Find whether the number is even or odd */
main( )
{
   int n, ch;
   printf ("Enter the number: \n");
   scanf ("%d", &n);
   if (n%2 = = 0)
   ch = 1;
   else
   ch = 2;
   switch (ch)
   {
      case 1;
```

```
            printf ("number is even \n");
            break;
         case 2:
            printf ("number is odd \n");
      }
}
```

Output:

```
   Enter the number :  22
   Number is even
```

LOOPS IN 'C'

Loops in 'C' allow a set of instructions to be performed until a certain condition is reached. There are three types of loops in 'C'.

- **for** loop
- **while** loop
- **do while** loop

The for Loop

It is a very useful looping construct in 'C'. It has three expressions: counter initialization, condition, and modification of the counter. The flowchart of **for** loop is shown in Figure 9.2.

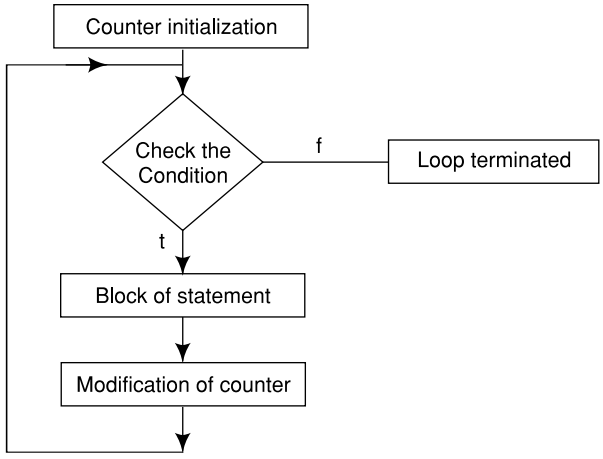


Fig. 9.2 *Working of for loop*

The general form of for loop is

```
for (initialization; condition; increment)
{
    statement 1;
        _____
        _____
    statement n;
}
```

The initialization is usually an assignment statement that is used to set the loop-control variable. The condition is a relational expression that determines when the loop will exit. The increment defines how the loop-control variable will change each time the loop is repeated. These three sections are separated by a semicolon. The **for** loop will be executed as long as the condition holds true. Once the condition becomes false, the program execution will resume on the statement following the block.

Example:

```
/* program to print a message 5 times */
main( )
{
int i;
for (i = 1; i < = 5; i ++)
{
   printf ("\n In the loop % d times", i);
}
}
```

More About 'for' Loop The **for** loop in 'C' has several capabilities that are not found in other loop constructs. More than one variable can be initialized at a time, in the **for** statement.

The statements:

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

can be rewritten as

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

The increment section may also have more than one part as given in the following example:

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

The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop-control variable. It can be seen in the following example.

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

The loop is executed as long as both the conditions $i < 20$ and sum < 100 are true.

It is also permissible to use expressions in the assignment statements of initialization and increment sections as given below.

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

Another unique aspect of the **for** loop is that one or more sections can be omitted, if necessary. An example program segment is given below.

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

Both the initialization and increment sections are omitted in the **for** statement, the initialization has been done before the **for** statement and the control variable is incremented inside the loop. We can set up time-delay loops using the null statement as follows:

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

This loop is executed 1,000 times without producing any output; it simply causes a time delay.

Example on 'for' loop

-
```
/* Find the sum of digits of the number */
main( )
{
    int num, count, sum, rem;
    sum = 0;
    printf ("\n Enter the number:");
    scanf ("%d", &num);
```

```
    for (count = num; count > 0; count / = 10)
    {
        rem = count % 10;
        sum + = rem;
    }
    printf ("sum of digits = %d", sum);
}
```

- Print the following pattern

```
*
*   *
*   *   *
*   *   *   *
*   *   *   *   *
main( )
{
    int row, i, j;
    printf ("\n Enter the number of rows:");
    scanf ("%d", &row);
    for (i = 1; i < = row; i ++)
    {
        for ( j = 1; j < = i; j ++)
        printf ("*");
        printf ("\n");
    }
}
```

The 'while' Loop

While is an entry-controlled loop statement. The basic format of the **while** statement is:

```
while (test condition)
{
    body of loop;
}
```

The test condition is evaluated and if the condition is true, the body of loop will be executed. That is why, the **while** loop is an entry-controlled loop statement. Figure 9.3 explains the portion of a flowchart containing a **while** loop.

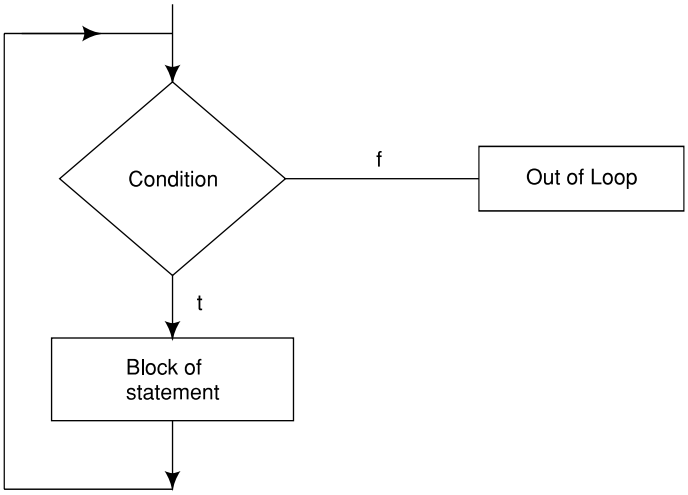


Fig. 9.3 *The while loop*

Example 1:

```
/* print the numbers 1 to 10 */
#include <stdio.h>
main( )
{
   int n = 1;
   while (n < = 10)
{
   printf ("%d \n", n);
   n++;
   }
}
```

Example 2:

```
/* Finding average of a set of numbers * /
#include <stdio.h>
main( )
{
```

```
    int n, count = 1;
    float x, average, sum = 0;
    /* initialize and read in a value of n */
    printf ("\n How many numbers:");
    scanf ("%d", &n);
    /* read the numbers and find sum */
    while (count < n)
    {
       printf ("\n %d number =" count + 1);
       scanf ("%f", &x);
       sum + = x;
       ++count ;
    }
    /* calculate the average and write answer */
    average = sum /n;
    printf ("\n Average is % f", average);
}
```

The 'do-while' Loop

The general form of **do-while** loop is:

```
do
{
    body of loop;
}
    while (test-condition);
```

It first executes the body of loop then evaluates the test condition. If the condition is true, the body of loop is executed again and again until the condition becomes false.

The following Figure 9.4 explains the flowchart of **do-while** loop.

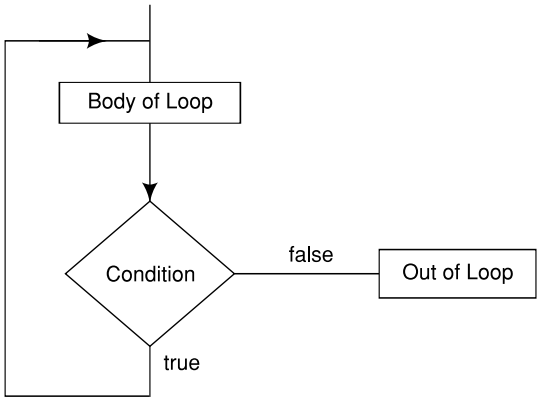


Fig. 9.4 *'do-while' loop*

Since the test condition is evaluated at the bottom of the loop, the **do-while** construct provides an exit-controlled loop.

Example:

```
/* Find the factorial of any number * /
#include <stdio.h>
main( )
{
   int n, no, fact = 1;
   printf ("Enter the number:");
   scanf ("% d", &n);
   no = n;
      if (n < 0)
      printf ("\n factorial of negative number not possible:");
      else
         if (n = = 0)
         printf ("Factorial of 0 is 1 \n");
      else
      do
      {
         fact * = n;
         n - -;
```

```
    }
    while (n > i);
    printf ("factorial of % d = % d", no, fact);
}
```

When to use 'while', 'do-while', and 'for' Loops

while **while** is an entry-controlled loop, that is, till the condition given is true, the loop will continue.
For example,

```
main( )
{
    int a = 10;
    while(a)
    {
        printf("% d \n", a);
        a--;
    }
}
```

Output : The loop will print 10 to 1 and as soon as the value of "a" becomes zero the loop will terminate.

do-while The **do-while** is an exit-controlled loop and used when statements inside it are required to be executed at least once.
For example,

```
main( )
{
    char ch;
    do
    {
        scanf("%c", &ch);
        a--;
    } while(ch!='s');
}
```

for The **for** loop is used when we know the number of iterations beforehand.
For example,

```
main()
{
    int i;
    for (i = 0; i < 10; i++)
    printf("%d", i);
}
```

INFINITE LOOPS

A looping process, in general, includes the following four steps:

- Setting of a counter
- Execution of the statements in the loop
- Testing of a condition for loop execution
- Incrementing the counter

The test condition eventually transfers the control out of the loop. In case, it does not do so, the control sets up an infinite loop and the loop body is executed over and over again. Such infinite loops should be avoided. The computer goes round and round until some special steps are taken to terminate the loop. Ctrl+C or Ctrl+Break are used to terminate the program caught in an infinite loop. Two examples of infinite loop are given below:

- ```
  main( )
  {
     main ( );
  }
  ```
- ```
  int i = 1 ;
  while (i < 10)
  {
     printf ("%d", i);
  }
  ```

This program will never terminate as variable *i* will always be less than 10. To get the loop terminated, an increment operation (*i*++) will be required in the loop.

NESTED LOOPS

Loops within loops are called nested loops. An overview of **nested while, for** and **do-while** loops is given below:

Nested while

It is required when multiple conditions are to be tested. The syntax of **nested while** is:

```
while (condition 1)
{   ----------
    ----------
    ----------
    while (condition 2)
    {   ----------
        ----------
        ----------
        while (condition n)
        {   ----------
            ----------
            ----------
        }
    }
}
```

For example,

```
int x = 5;
while (x)
{   int a = 0;
        while (a < 10)
        {   a++;
        } x—;
}
```

Nested for

It is used when multiple set of iterations are required. The syntax of **nested for** is:

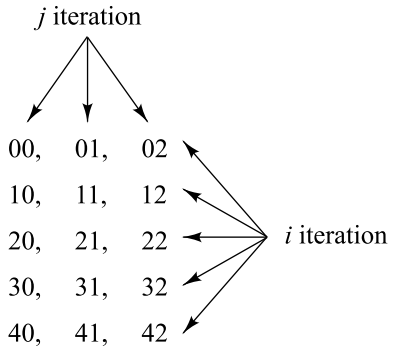
```
for( ; ; )
{   ----------
    ----------
    ----------
    for( ; ; )
    {   ----------
```

```
            ----------
            ----------
        }
    }
```

For example,

```
    for (i = 0; i < 5; i++)
    {
        for ( j = 0; j < 3; j++)
        {
            printf("%d %d", i, j);
        }
        printf("\n");
    }
```

Output is:



THE break STATEMENT

The break statement is used to terminate loops or to exit from a switch. When a break is encountered inside any 'C' loop, the control automatically passes to the first statement after the loop. It can be used within a 'while', a 'do-while', a 'for' or a 'switch' statement. The break statement is written simply as the *break*; without any embedded expression or statements.

Some illustrations of loop that contain break statements are given below. In each situation, the loop will continue to execute as long as the current value for the floating-point variable *x* does not exceed 100. However, the computation will break out of the loop if a negative value for *x* is detected.

'while Loop'

```
scanf ("%d", &x);
while (x < = 100)
{
   if (x < 0)
   {
      printf ("Enter negative value for x\n");
      break;
   }
   / * process the nonnegative value of x * /
   scanf ("%d", &x);
}
```

'do-while Loop'

```
do
{
   scanf ("%d", &x);
   if (x < 0)
   {
      printf ("Error-negative value for x");
      break;
   }
   / * process the nonnegative vlaue of x */
}
while (x < = 100);
```

'for Loop'

```
for (count = 1; x < = 100; ++ count)
{
   scanf ("% f ", &x);
   if (x < 0)
   {
      printf ("Error-negative value for x");
```

```
            break;
        }
        /* process the nonnegative value of x */
    }
```

In the event of several **nested while, do-while, for** or **switch** statements, a break statement will cause a transfer of control out of the immediate enclosing statement, but not out of the outer surrounding statements.

Consider the following outline of a '**while** loop' embedded with a '**for** loop'.

```
for (count = 0; count < = n; ++ count)
{
    while (( c = getchar( ))! = '\n')
    {
        if (c = '*') break;
    }
}
```

If the character variable *c* is assigned an asterisk (*) then the **while** loop will be terminated. However, the **for** loop will continue to execute. Thus, if the value of count is less than *n* when the break occurs, the computer will increment count and make another pass through the **for** loop.

THE continue STATEMENT

The **continue** statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. Rather, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. The continue statement tells the compiler to skip the following statements and continue with the next iteration. The continue statement can be included within a **while**, a **do-while** or a **for** statement. It is written simply as **continue**; without any embedded statements or expressions.

Some illustrations of loops that contain **continue** statements are given below. In each case, the processing of the current value of *x* will be bypassed if the value of *x* is negative. Execution of the loop will then continue with the next pass.

do-while Loop

```
do
{
    scanf ("%f", &x);
    if (x < 0)
    {
```

```
            printf ("Error-negative value for x");
            continue;
         };
      /* process the non negative value of x*/
      }
      while (x < = 100);
```

for Loop

```
      for (count = 1; x < = 100; ++ count)
      {
         scanf ("%f", &x);
         if (x < 0)
         {
            printf ("Error-negative value for x");
            continue;
         }
      /* process the non-negative value of x */
      }
```

Example:

```
      /* calculates the average of the non-negative numbers in
list of n numbers */
      #include <stdio.h>
      main( )
      {
         int n, count, navg;
         float x, average, sum;
         navg = 0;
         sum = 0.0;
         printf ("How many numbers ? \n"); /* initialize and read
in a value for n */
         scanf ("%d", &n);
         for (count = 1; count < = n; ++ count) /* read in the
numbers */
```

```
        {
            printf ("x =");
            scanf ("%f", &x);
            if (x < 0) continue;
            sum + = x;
            ++navg;
        }
        average = sum/navg; /* calculate the average and write
out the answer */
        printf ("\n The average is %f\n", average);
    }
```

THE exit( ) FUNCTION

The exit( ) function is used to terminate the execution of 'C' program. It is a standard library function and uses the header file stdlib.h.

The general form of the exit( ) function is

exit (int status);

The difference between break and exit( ) is that the former terminates the execution of a loop in which it is written, while exit( ) terminates the execution of the program itself.

The status (in the general form of exit( )) is a value returned to the operation system after the termination of the program.

The value zero indicates that the termination is normal, where as while value one indicates different types of errors.

THE goto STATEMENT

'C' supports the goto statement to branch unconditionally from one point to another in the program. A goto statement breaks the normal sequential execution of the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred.

The general form of goto and label statements are shown below:

goto label;	label:
. . . . .	statements;
label:	. . . . .
statement;	goto label;

The label: can be placed anywhere in the program, either before or after the goto label; statement.

During the execution of a program, when a statement like goto begin; is met, the flow of control will jump to the statement immediately following the label begin. The following program is written to evaluate the square root of numbers read from the terminal. Due to the unconditional goto statement at the end, the control is always transferred back to the input statement. It puts the computer in a permanent infinite loop.

```
main( )
{
    double x, y;
    read : scanf ("%f", &x);
    if (x < 0) goto read;
    y = sqrt (x);
    printf ("%f %f \n", x, y);
    goto read;
}
```

Another use of the goto statement is to transfer the control out of a loop (or nested loops) when certain peculiar conditions are encountered. We should try to avoid using goto as far as possible. It can, however, be used to enhance readability of the program or to improve the execution speed.

Example:

```
. . . . . .
while (. . . . . )
{
   for (. . . . . )
   {
   . . . . . .
   if (. . . . . .) goto end_of_program;
   . . . . . .
   }
   . . . . . . jumping out of loop
}
end_or_program:
```

Summary

- Control statements specify the order in which the various statements are to be executed.
- Conditional statements are the decision-making statements.
- **if-else** statement checks a condition. Depending upon the result, the set of statements are executed.
- **switch** statement is a multiway decision statement.
- Loops allow a set of instructions to be performed repeatedly.
- **for** loop is an open-ended loop.
- **while** is an entry-controlled loop.
- **do-while** is an exit-controlled loop.
- Loops within loops are called nested loops.
- A loop with no terminating condition is called an infinite loop.
- break statement is used to terminate the loop.
- exit( ) function is used to terminate the program.

Review Exercise

Multiple-Choice Questions

1. How many types of control statements are available in C language?
   (a) 3
   (b) 5
   (c) 2
   (d) 4
2. Which one of the following is also called entry controlled loop?
   (a) **if**
   (b) **while** loop
   (c) **do-while** loop
   (d) none of these
3. Loops present within other loops are called
   (a) linked loops
   (b) multi level loops
   (c) nested loops
   (d) none of these
4. The unconditional branch statement is:
   (a) if
   (b) switch
   (c) for
   (d) goto

5. The exit( ) function causes termination of
   (a) 'C' program in which it occurs
   (b) the loop in which it occurs
   (c) the last function being called
   (d) none of these

State whether True or False

1. exit( ) is a standard library function that comes ready-made with the 'C' compiler.
2. The initialization expression of the for loop can contain more than one statement separated by a semicolon.
3. When a break is encountered inside any 'C' loop, control automatically passes to the first statement after the loop.
4. When the keyword 'continue' is encountered inside any loop, control automatically passes to the beginning of the loop.
5. Nesting of **if-else** control statements is not permitted.

Fill in the Blanks

1. _____ statement is used to terminate the loop.
2. _____ is an exit controlled loop.
3. The switch statement is a _____ decision statement.
4. The function exit ( ) is defined in _____ header file.
5. The goto statement always requires a _____ .

Descriptive Questions

1. What is meant by looping?
2. How is the execution of a **while loop** terminated?
3. What is the purpose of the **do-while** statement? How does it differ from the while statement?
4. What is the purpose of the **for** statement? How does it differ from the **while** statement and the **do-while** statement?
5. What will happen when the break statement is executed if a break statement is included within the innermost of several nested control statements?
6. What will be the output of the following programs:

```
(a) void main( )
    {
       int a = 0, b = 0, c = 0, d = 0;
       if ((a == b) && (a * b < = b))
       {
          if (d == 1)
          c = 1;
          else
```

```
            if (a == 1)
            c = 2;
         }
         else
         c = 3;
      if ((b == 0) && (a == b) && (a == 1))
         a = 1;
      printf ("c = %d, d = %d", c, d);
         d = 1;
      }
(b) main( )
      {
         int i = 10;
         for (; i < = 10; if (i == 11);)
         printf ("%d", ++i);
      }
(c) main( )
      {
         int index = 3, x = 100;
         while (index)
         printf ("%d", x, index -- );
         x++;
         index --;
      }
(d) void main( )
      {
         int i = 5;
         while (i)
         {
            i - - ;
            if (i == 3)
            continue
```

```
            printf ("\n UPTEC");
         }
      }
(e) main( )
      {
         int i = 1;
         switch (i)
         {
            case 1:
               printf ("\n Case1");
               break;
            case 1*2+4:
               printf ("\n Case2");
               break;
            }
      }
(f) main( )
      {
         int i = 1;
         for (; ;)
         {
            printf ("%d", i ++)
            if (i > 0)
            break;
         }
      }
```

7. Print all prime numbers from 1 to 300.
8. Generate following pattern by using loops.

```
                1
            1   2   1
        1   2   3   2   1
    1   2   3   4   3   2   1
1   2   3   4   5   4   3   2   1
```

9. Write a program to produce following output:

```
A B C D E F G F E D C B A
A B C D E F . F E D C B A
A B C D E . . . E D C B A
A B C D . . . . . D C B A
A B C . . . . . . . C B A
A B . . . . . . . . . B A
A . . . . . . . . . . . A
```

10. Generate the following pattern using nested loop

```
z y x w u w x y z
  z y x w x y z
    z y x y z
      z y z
        z
```

11. Write a program to find the factorial of a given number using the **for** loop.
12. Write a program to calculate the percentage of a student's total marks in three subjects.

CHAPTER

10

Arrays

Key Features

- Introduction to Arrays
- One Dimensional Array
- Strings
- Two Dimensional Array
- Multi-Dimensional Array

Whenever there is a need to store a group of data of the same type in the memory, arrays are used. Arrays are the contiguous memory location used to store similar data in the memory. The ordinary variables are capable of holding only one value at a time. However, there are situations in which we want to store more than one value at a time in a single variable. For this purpose arrays are used.

In this chapter we will discuss about arrays in detail.

INTRODUCTION TO ARRAYS

Ordinary variables are capable of holding only one value at a time. If we want to store more than one value at a time in a singe variable, we use arrays.

Arrays are data types used to represent a large number of homogeneous values to a contiguous memory location. 'C' language allows both single dimensional as well as multidimensional arrays. All the elements of an array have the same data type and in a strictly contiguous fashion. Arrays may be storage class automatic, external or static, but not register.

The general format:

Data_type array-name [size]

A typical array declaration allocates the memory strarting from a *base address*. To store the elements of the array, the compiler assigns an appropriate amount of memory starting from a base address. The elements of the array are accessed using a *subscript*, also called an **index**. The value of the subscript must be in the range 0 to size -1. An array subscript value outside this range will cause a run-time-error.

ONE-DIMENSIONAL ARRAY

A list of items can be given one variable name using only one subscript and such a variable is called a *one-dimensional array*.

Example: If we want to store a set of five numbers by an array variable number, then it will be accomplished in following way:

```
int number [s];
```

This declaration will reserve five contiguous memory locations as shown below in Figure 10.1.

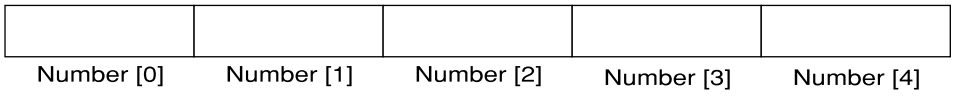


Fig. 10.1 *Structure of a single dimensional array*

As 'C' performs no bounds checking, therefore, care should be taken to ensure that the array indices are within the declared limits.

Array Declaration

Arrays are defined in the same manner as ordinary variables, except that each array name must be accompanied by a size specification.

The general form of an array declaration is:

```
data-type array-name [size];
```

Data-type specifies the type of array and size is the positive integer number or symbolic constant that indicates the maximum number of elements that can be stored in the array.

Example: float height [50];

This declaration declares an array named height containing 50 elements of type float.

> **Note:** The compiler will interpret first element as height [0] as in 'C' the array elements are indexed from 0 to [size-1].

Array Initialization

The elements of array can be initialized in the same way as the ordinary variables, when they are declared. Given below are some examples which show how the arrays are initialized.

```
static int num[6] = {2, 4, 5, 45, 12};
static int n[ ] = {2, 4, 5, 45, 12}
static float press[ ] = {12.5, 32.4, –23.7, –11.3};
```

In these examples note the following points:

- Till the array elements are not given any specific values, they contain garbage value.
- If the array is initialized where it is declared, its storage class must be either static or extern. If the storage class is static, all the elements are initialized by 0.
- If the array is initialized where it is declared, mentioning the dimension of the array is optional.

Accessing Elements of an Array

Once an array is declared, individual elements of the array are referred using a subscript or an index number. This number specifies the element's position in the array. All the elements of the array are numbered starting from 0. Thus number [5] is actually the sixth element of the array.

Entering Data into an Array

It can be explained by the following examples:

```
main( )
{    int num [6];
     int count;
     for (count = 0; count < = 5; count ++)
     {    printf ("\n Enter %d element:" count+1);
          scanf ("%d", num [count]);
     }
}
```

In this example, by using the **for** loop, the process of asking and receiving marks is accomplished. When count has the value zero, the scanf( ) statement will cause the value to be stored at num [0]. This process continues until count has value greater than 5.

Reading Data from an Array

Consider the program given above. It has entered six values in the array num. Now to read values from this array, we will again use the **for** loop to access each cell. The given program segment explains the retrieval of the values from the array.

```
for (count = 0; count < = 5; count ++)
{
     printf ("\n %d value =", num [count]);
}
```

Memory Representation of Array

Consider the following array declaration:

```
int arr[8];
```

16 bytes get immediately reserved in the memory because each of the 8 integers would be 2 bytes long and since the array is not being initialized, all eight values present in it would be garbage values.

Whatever be the initial values, all the array elements would always be present in contiguous memory location. This arrangement of array elements in the memory is shown in Figure 10.2.

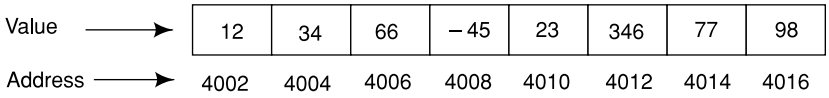


Fig 10.2 *Memory representation of an array*

Note: In 'C' there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in the memory outside the array. This will lead to unpredictable results and there will be no error message to warn you that you are going beyond the array size. So to see to it that we do not reach beyond the array size is entirely the programmer's task and not the compiler's.

Example:

- A program to find average marks obtained by a class of 30 students in a test.

```
main( )
{
   float avg, sum = 0; int i;
   int marks[30];                /* array declaration */
   for (i = 0; i < = 29; i++)
   {
      printf ("\n Enter marks:  \t");
      scanf ("%d", &marks[i]);  /* store data in array */
   }
   for (i = 0; i < = 29; i++) sum = sum + marks[i];   /* read data from
                                                        an array */
   avg = sum / 30;
   printf ("\nAverage marks: %f", avg);
}
```

- A program to read in a one-dimensional character array, convert all the elements to uppercase, and then write the converted array.

```
#include <stdio.h>
# define SIZE 80
main( )
{  char letter [SIZE]; int count;
   for (count = 0; count <SIZE; ++ count)  /* read in the line */
```

```
    letter [count] = getchar( );
    for (count = 0; count <SIZE; ++ count)
                          /* write out the line in upper-case */
    putchar (toupper (letter[count]));
}
```

It is sometimes convenient to define an array size in terms of a symbolic constant rather than a fixed integer quantity. This makes it easier to modify a program that utilizes an array, since all references to the maximum array size can be altered simply by changing the value of the symbolic constant.

STRINGS

Just as a group of integers can be stored in an integer array, a group of characters can be stored in a character array or"strings". The string constant is a one dimensional array of characters terminated by null character ('\0'). This null character'\0' (ASCII value 0) is different from'0' (ASCII value 48).

The terminating null character is important because it is the only way the function that works with string can know where the string ends.

Example: Static char name [ ] = {'K','R','I','S','H','N','A'\0'};

This example shows the declaration and initialization of a character array. The array elements of a character array are stored in contiguous locations with each element occupying one byte of memory.

K	R	I	S	H	N	A	'\0'
4001	4002	4003	4004	4005	4006	4007	4008

String Handling Function

A string can be represented as a one-dimensional character-type array. Each character within the string will be stored within one element of the array. Sometimes it is required that the characters within a string be processed individually while some other times it is required that the entire string be processed as a single entity. Such problems can be simplified considerably by using special, string-oriented library functions.

For example, most 'C' compilers include library functions that allow strings copied, concatenated or compared. Other library functions allow operations on individual characters within the string; for example, they allow individual characters to be found within strings and so on.

Some of the commonly used string functions are:

int strcmp (string1, string2)	compares string1 with string2, returns < 0 if string 1 <string2, 0 if string1 = =string2, or >0 if string1?string2
char*strcat (char*s1, char*s2)	concatenates string s2 to the end of string s1 and returns s1.
char*strcpy (char*s1, const char*s2	copies string s2 to s1, including '\0', return s1

Notes: 1. Contrary to the numeric array where a five digit number can be stored in a one array cell, in the character arrays only a single character can be stored in a single cell. So in order to store an array of strings, a two-dimensional array is required.
2. As scanf( ) function is not capable of receiving a multi-word string. Such strings should be entered using gets( ).

TWO-DIMENSIONAL ARRAY

It is possible to have an array of more than one-dimension. Two-dimensional arrays (2-D arrays) are arrays of a number of one-dimensional arrays.

A two-dimensional array is also called a matrix. Consider the following table:

	Item1	Item2	Item3
Sales 1	300	275	365
Sales 2	210	190	325
Sales 3	405	235	240
Sales 4	260	300	380

This is a table of four rows and three columns. Such a table of items can be defined using two dimensional arrays.

General form of declaring 2-D array is

```
data_type array_name [row_size] [colum_size];
```

example i) int marks [4][2];

It will declare an integer array with marks of four rows and two columns. An element of this array can be accessed by the manipulation of both the indices.

printf ("*%d*", marks [2] [1]) will print the element present in the third row and the second column.

Initialization of a Two-Dimensional Array

Two-dimensional arrays may be initialized by a list of initial values enclosed in braces following its declaration.

Example: static int table [2] [3] = {0, 0, 0, 1, 1, 1};

initializes the elements of the first row to 0 and the second row to 1. The initialization is done in rows.

The aforesaid statement can be equivalently written as

```
static int table [2] [3] = {{0, 0, 0}, {1, 1, 1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two dimensional array in the form of a matrix as shown below:

```
static int table [2] [3] = {{0, 0, 0},
                            {1, 1, 1}};
```

Note: In the syntax of the above statement, commas are required after each brace that closes off a row, except in the case of the last row.

If the values are missing in an initializer, they are automatically set to 0. For instance, the statement

```
static int table [2] [3] = {{1, 1},
                            {2}};
```

will initialize the first two elements of the first row to 1, the first element of the second row to 2, and all the other elements to 0.

When all the elements are to be initialized to 0, the following shortcut method may be used.

```
static int m [3] [5] = {{0}, {0}, {0}};
```

The first element of each row is explicitly initialized to 0 while other elements are automatically initialized to 0.

While initializing an array it is necessary to mention the second dimension (column), whereas the first dimension (row) is optional. Thus the following declarations are acceptable:

```
static int arr [2] [3] = {12, 34, 23, 45, 56, 45};
static int arr [ ] [3] = {12, 34, 23, 45, 56, 45 };
```

Memory Representation of Two Dimensional Array

All elements of a matrix get stored in the memory in a linear fashion. The two ways in which elements can be represented in computers memory are **- row major order** and **column major order**.

In **row major representation**, the first row of the array occupies the first set of memory locations, second occupies the next set and so on.

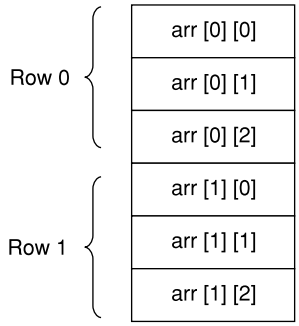


Fig. 10.3 *Representation of 2D-array in row major order*

The diagrammatic representation of two dimensional array arr [2] [3] in row major order is given in the figure. In the figure, in the memory locations are first occupied by the first row of the array. Now base (arr) has the address of first element of the array **arr [0] [0]**, the second has address of arr [0] [1].

The other method of representing a two-dimensional array in memory is the **column major order**. Under this representation, the first column of the array occupies the first set of memory locations reserved for the array, the second column occupies the next set, and so forth.

The diagrammatic representation of two dimensional array arr [2] [3] in column major order is given in Figure 10.4.

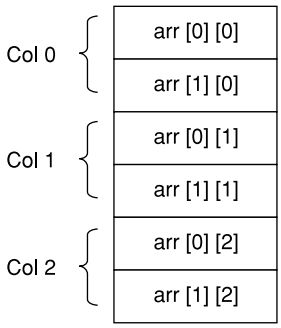


Fig. 10.4 *Representation of 2-D array in column major order*

In the figure the memory location is first occupied by the first column of the array. Now base (arr) has the address of the first element of the array arr [0] [0], the second has the address of arr [1] [0] and so forth.

Address Calculation Techniques

The formula to calculate the address of $(j, k)^{th}$ element of a 2-D array of mXn dimenssion is

$$A(j, k) = \text{base}(A) + W[N(j - 1) + (K - 1)$$

Consider a matrix declaration:

int matrix [3] [4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};

Let us assume that the base address of an array matrix is 100. Since $W = 2$ (as array is of integer whose size is 2), therefore, according to the formula, address of $(2, 3)^{th}$ element in the array matrix will be

$$\begin{aligned} \text{LOC}(2, 3) &= 100 + 2[4(2 - 1) + (3 - 1)] \\ &= 100 + (4 + 2) * 2 \\ &= 100 + 12 \\ &= 112 \end{aligned}$$

Note: Lower boundary of the array is assumed to be 1.

Thus, we see that in the above matrix, address of $(2, 3)^{th}$ element which is 7, is 112 (as depicted in the Figure 10.5).

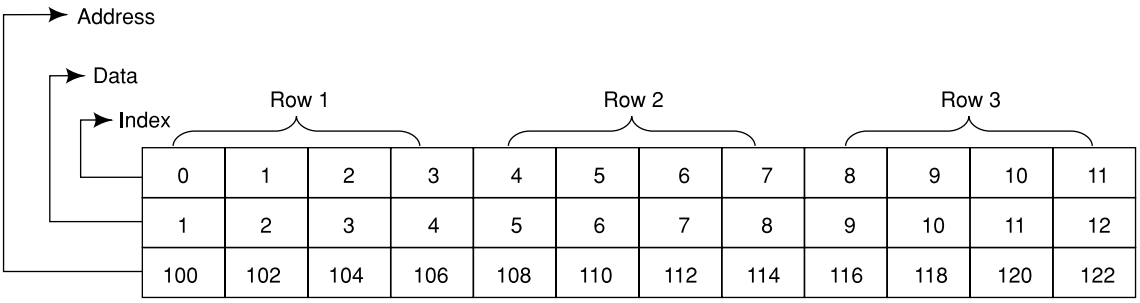


Fig. 10.5 *Row Major representation*

Similarly, for the column major order representation, let us consider the same matrix.

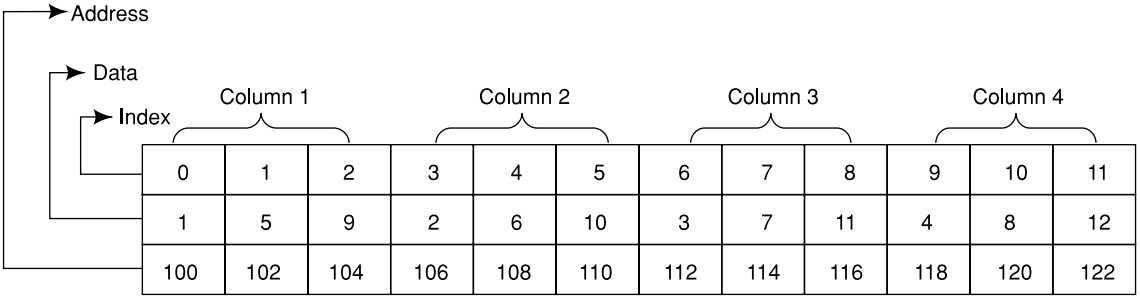


Fig. 10.6 *Column Major representation*

To find the address of $(j, k)^{th}$ element in an array of mXn dimension and W word size of each element, the following formula can be used:

$$A(j, k) = \text{Base}(A) + W[M(k-1) + (j-1)]$$

$$\therefore \quad \text{LOC}(2, 3) = 100 + 2[3(3-1) + (2-1)]$$

$$= 100 + 2(7)$$

$$= 114$$

Examples The arrangement of the array elements of a two dimensional array of students, which contains roll numbers in one column and the marks in the other (in memory) is shown below:

- Program that stores roll number and marks obtained by a student side-by-side in a matrix

Notation →	S[0][0]	S[0][1]	S[1][0]	S[1][1]	S[2][0]	S[2][1]	S[3][0]	S[3][1]
Value →	1234	1234	1234	1234	1234	1234	1234	1234
Address →	5002	5004	5006	5008	5010	5012	5014	5016

```
main( )
{
   int stud [4] [2];
   int i, j;
   for (i = 0; i < = 3; i++)
   {
         printf ("\n Enter rollno. and marks");
         scanf ("%d%d", &stud [i] [0], &stud[i] [1]);
   }
   for (i = 0; i < = 3; i++)
   printf ("%d%d\n", stud [i] [0], stud [i] [0];
}
```

There are two parts to the program, in the first part through a **for** loop, we read in the values of roll number and marks, whereas in second part through another **for** loop we print out these values.

- Program to print multiplication table, using two-dimensional array.

```
#define ROWS 5
#define COLUMNS 5
main( )
{
   int row, column, product [ROWS] [COLUMNS];
   int i, j;
   printf ("MULTIPLICATION TABLE \N");
   printf ("");
   for (j = 1; j < = COLUMNS; j++)
   printf ("%4d", j);
   printf ("\n");
   for (i = 0; i < ROWS; i++)
   {
      row = i + 1;
      printf ("%2d\n", row);
      for (j = 0; j < COLUMNS; j++)
      {
            column = j+1;
```

```
                product [i] [j] = row *column;
                printf ("%4d", product [i] [j]);
            }
            printf ("\n");
        }
    }
```

Output : Multiplication Table

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

MULTI-DIMENSIONAL ARRAY

'C' allows arrays of three or more dimensions. Multi-dimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript.

The general form of a multi-dimensional array is:

data_type array_name [s1] [s2] [s3] . . . [sm];

Example: int survey [3] [5] [12];
float table [5] [4] [5] [3];

Here, the survey is a three dimensional array declared to contain 180 integer_type elements. Similarly, a table is a four dimensional array containing 300 elements of floating point type.

An example of initializing a four dimensional array:

```
static int arr [3] [4] [2]  =  {{{2, 4}, {7, 8}, {3, 4}, {5, 6},},
                               {{7, 6}, {3, 4}, {5, 3}, {2, 3}, },
                               {{8, 9}, {7, 2}, {3, 4}, {6, 1}, }
                               };
```

In this example, the outer array has three elements, each of which is a two-dimensional array of four rows, each of which further, is a one-dimensional array of two elements.

Example:

- Sorting an integer array (bubble sort).

```
#include <stdio.h>
void main( )
```

```
{
      int arr [5];
      int i, j; temp;
      printf ("\n Enter the elements of the array:"};
      scanf ("%d", & arr [i]);
      for (i = 0; i < = 4; i ++)
      {
         for (j = 0; j < = 3; j ++)
         if (arr [j] > arr [j+1])
         {
            temp = arr [j];
            arg [j] = arr [j+1];
            arr [j+1] = temp;
         }
      }
      printf ("\ n The Sorted array is :");
      for (i = 0; i < 5; i++)
      printf ("\ t %d", arr [i]);
}
```

- To insert an element into an existing sorted array (insertion sort).

```
#include <stdio.h>
main( )
{
      int i, k, y, x [20], n;
      for (i = 0; i < 20; i++)
      x [i] = 0;
      printf ("\ Enter the number of items to be
inserted:\n");
      scanf ("%d", &n);
      printf ("\n Input %d values \n", n);
      for (k = 0; k < n; k++)
      {
```

```
            scanf ("%d", &x [k]);
            y = x [k];
            for (i = k-1; i > = 0 && y < x [i]; i --)
            x [i+1] = x[i];
            x [i+1] = y;
         }
         printf ("\n The sorted numbers are:");
         for (i = 0; i < n; i++)
         printf ("\n %d", x [i]);
      }
```

Example:

- Accept character string and find its length.

Note: We will solve this question by looping instead of using a library function strlen( ).

```
#include <stdio.h>
void main( )
{
   char name [20];
   int i, len;
   printf ("\n Enter the name:");
   scanf ("%s", name);
   for (i = 0; name [i] ! ='\0'; i++);
   len = i - 1;
   printf("\n Length of array is % d", len);
}
```

Example:

- Searching for a particular element in a list of numbers (using binary search)

```
#include <stdio.h>
void main( )
{
   int a[20],i,n,k,p,top,mid, bottom;
   printf ("Enter the total no. of elements is the list \n");
   scanf ("%d", &n);
```

```
            printf ("Enter the elements of the list\n");
            for (i=0; i<n; i++) /*storing a list of numbers in array a*/
            scanf ("%d", &a[i]);
            printf ("Enter the value to be searched");
            scanf ("%d", &p);
            top = 0; /*first element of list index */
            bottom = n-1; /*index for last element of the test */
            k = 0 ; /* to be used as a flag */
            while ((top < = bottom) && (k!=1))
            {
               mid = (top + bottom)/2; /*index for centre element of
the list*/
               if (a[mid] ==p) /* element found */
               k = 1
               else
               if (a[mid] >p)
               bottom = mid -1 ;
               else
               top = mid + 1;
            }
            if (k ==1)
            printf ("\n The element is at %dth position", mid);
            else
            printf ("not found");
         }
```

Example:

- In a square matrix check whether:
  (a) It is a sparse matrix
  (b) It is a upper triangular matrix.

```
#include <stdio.h>
#include <conio.h>
# define MAX 10
void main( void)
```

```
{
   int matrix [MAX][MAX], n, i, j ;
   char ch ;
   void sparse (int [ ][ ], int); /*prototype for sparse
function*/
   void upper (int [ ][ ], int); /* prototype for upper
triangular function*/
   printf ("Enter the actual dimension of the square matrix
? \n");
   scanf ("%d", &n);
   printf ("Enter row-wise elements of the matrix\n");
   for ( i = 0; i < n; i++)
   for (j = 0; j < n; j++)
   scanf ("%d", &matrix[i][j]);
   sparse (matrix, n) ;
   upper (matrix, n);
}
void sparse (int matrix [ ] [MAX], int n)
{
   int i, j, count1, count2;
   count1 = count2=0;
   for (i = 0; i < n; i ++)
   for (j = 0; j < n; j++)
   if (matrix [i][j] = = 0)
   count1++;
   else
   count2++;
   if (count1>=count2)
   printf ("\n Matrix is a sparse matrix") ;
   else
   printf ("\n Matrix is not a sparse matrix");
}
void upper (int matrix [ ] [MAX], int n)
```

```
{
   int i, j, k =1
   i = 0;
   while (i < n && k)
   {
      j = 0 ;
      while (j <i && k)
      {
         if (matrix [i][j]! =0)
         k=0;
         else
         j++;
      }
      i++;
   }
   if (k)
   printf ("\n It is a upper triangular matrix");
   else
   printf ("\n It is not an upper triangular matrix");
}
```

Summary

- An array is a collection of similar elements.
- An array is also known as a subscripted variable.
- Before use, the type and dimension of the array must be declared.
- The first element in the array is numbered 0, so the last element is one less than the size of the array.
- A character array is known as a string.
- A string is always terminated by a null character ('\0').
- A two dimensional array is also called a matrix.
- The elements of a two dimensional array are stored in a continuous chain.
- Arrays with three or more dimensions are multi-dimensional arrays.

Review Exercise

Multiple-Choice Questions

1. Arrays declared as static contain an initial value which is
   (a) 10
   (b) 1
   (c) 0
   (d) none of these
2. An integer array of 10 elements occupies
   (a) 10 bytes in memory
   (b) 20 bytes in memory
   (c) 40 bytes in memory
   (d) none of these
3. A string is a character array terminated by
   (a) z
   (b) zero
   (c) NULL character
   (d) none of these
4. The subscript of an array in 'C' language starts from
   (a) –1
   (b) 1
   (c) 0
   (d) user defined
5. Arrays, by default, contain
   (a) –1
   (b) garbage value
   (c) zero
   (d) none of these

State whether True or False

1. The subscript of the first element in the array is numbered 1.
2. A string constant is a one dimensional array of characters terminated by a NULL character ('\0').
3. Automatic arrays, unlike automatic variables, cannot be initialized.
4. Only external and static arrays can be initialized.
5. Size of an array can be supplied at execution time.

Fill in the Blanks

1. In an array int s[5], the letter S contains the _____ of the array.
2. In an array the subscript 4, actually contains the _____ element of the array.

3. A two dimensional array is stored in the form of _____ array in the memory.
4. Array elements can be accessed _____ .
5. A two dimensional array is also called a _____ .

Descriptive Questions

1. In what way does an array differ from an ordinary variable?
2. How are individual array elements identified?
3. How does an array definition differ from that of an ordinary variable?
4. What value is automatically assigned to those array elements not explicitly assigned?
5. What will be the output of the following programs:

```
(a) main( )
    {
       char a [5 * 2/2] = {'a','b','y','d','e'};
       printf ("%d \n", a [-3]);
    }
(b) main( )
    {
       char a [ ];
       a [0] ='A';
       printf ("%c", a [0]);
    }
(c) main( )
    {
       char p[ ];
       p = "%d \n";
       p[1] = 'c';
       printf (p, 65);
    }
(d) main( )
    {
       char a[4] = {'x','y','z'};
       printf ("%s \n", a);
    }
```

```
(e) main( )
    {
       static char city[20] ="Nagpur";
       int i = 0;
       while (i [city])
          printf ("%c", city [i++]);
    }
```

6. Read a list of 10 numbers and print it in reverse order.
7. Find the maximum number and its position in a list of N numbers.
8. Read a string and check if it is a palindrome or not.
9. Insert a value in an array at a particular location.
10. Sort the elements of a two-dimensional array in descending order.
11. Read 3×3 matrices and find their sum.
12. Calculate the sum of all non-zero elements of the array.
13. Find the transpose of a square matrix.

CHAPTER

11

Functions

Key Features

- Introduction to Functions
- Function Declaration and Prototypes
- Function Definition
- Storage Classes
- Scope and Lifetime of Declaration
- Passing Parameters to Functions
- Command Line Arguments
- Recursion in Function

A function is a self-contained block of programs that performs a coherent task of some kind. Every 'C' program can be thought of as a collection of these functions. Functions increase the reusability of the program. There is no need to write the same code of data repeatedly if we use functions.

Now let us discuss the functions in detail.

INTRODUCTION TO FUNCTIONS

Functions are the 'C' building blocks where every program activity occurs. It is a self contained program segment that carries out some specific, well-defined tasks. Every 'C'program must have a function. One of the functions must be main( ).

Program execution will always begin by carrying out the instructions in function main(). Additional functions will be subordinate to main( ) and also to one another.

'C' functions can be classified into two categories:

- Library functions : Predefined in the standard library of 'C'. It is just needed to be included in the library.
- User defined functions : It has to be developed by the user at the time of program writing.

Need of User-Defined Functions

If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. This approach clearly results in a number of advantages.

- It facilitates top down modular programming.
- The length of a program can be reduced by using a function.
- Debugging is easier.
- Reusability of function increases.

The general form of a function is:

```
type arg1, arg2, arg3;
function_name (arg1, arg2, arg3)
{
   statement 1;
      :
      :
   statement n;
}
```

This can also be written as:

```
function_name (type arg1, type arg2, type arg3)
{
   statement 1;
      :
      :
   statement n;
}
```

FUNCTION DECLARATION AND PROTOTYPES

Before defining the function, it is appropriate to declare the function along with its prototype. In function prototype, the return value of function, type, and number of arguments are specified. The declaration of all function statements should be first statements in main( ).

The general form of function declaration using ANSI Prototype is

data_type function_name (type1 arg1, type2 arg2 - - - -);

where arg1, arg2. . . are te list of arguments.

Function prototypes are desirable because they facilitate error checking between calls to a function and corresponding function definition. They also help the compiler to perform automatic type conversions on function parameters. When a function is called, actual arguments are automatically converted to the types in function definition using normal rules of assignment.

FUNCTION DEFINITION

The general form of a function definition is:

data_type function_name (formal argument list)

argument declarations;

```
{
    local variable declarations;
    executable statement 1;
    executable statement 2;
        ----------
    executable statement n;
    return (expression);
}
```

Where *data_type* represents the data type of the value which is returned. The type specification can be omitted if the function returns an integer or a character.

The formal argument list is a list of variables separated by commas that receive the values from the main program when the function is called. The identifiers used as formal arguments are "local" as they are not recognized outside of the function. They are also known as parameters or formal parameters.

Note:	An empty pair of parenthesis must follow the function name if the function definition does not include any arguments.

The argument declarations follow the first line. Each formal argument must have the same data type as its corresponding actual argument.

The remainder of the function definition is a compound statement that defines the action to be taken by the function. It is referred to as the body of the function.

The last statement in the body of the function is return (expression);. It is used to return the computed result, if any, to the calling program.

Calling a Function

A function can be called by specifying its name followed by a list of arguments enclosed in parenthesis and separated by commas. If a function call does not require any arguments, an empty pair of parenthesis must follow the function name.

The arguments appearing in the function call are referred to as actual agruments, in contrast to the formal arguments that appear in the first line of function definition.

Example:

- /* Program to find square of given number */

```
main( )
{
   float square (float);  /* function prototype declaration*/
   float a, b;
   printf ("\n Enter the number:");
   scanf ("%f ", &a);
```

```
   b = square(a);    /* calling of function with */
                     / * actual arguments */
   printf ("Square of entered no. is = %f" , b);
}
float square(x)  / * function definition with formal argument * /
float x;         /* formal argument declaration * /
{
   float y;      /* Local variable decleration
   y = x * x;
   return(y);
}
```

Output : Enter the number : 2
Square of entered no. is = 4

The Return Statement

Information is returned from the function to the calling portion of the program via the return statement. It causes the control to be returned to the point from where the function was accessed. The return statement can take one of the following forms:

return;

or

return (expression);

In the return (expression); statement, the value of the expression is returned to the calling of the program.

A function can have multiple return statements, each containing a different expression.

Example:

```
/* Program to convert lowercase character to uppercase */
#include <stdio.h>
main( )
{
   char lower, upper;
   char Lower_to_upper (char Lower);
   printf ("\n Enter the lowercase character:");
   scanf ("%c", & lower);
   upper = Lower_to_upper (lower);
   printf ("\n The upper case Equivalent is % c", upper);
```

```
}
char lower_to_upper (Char ch)
{
   char c2;
   c2 = (c1 > = 'a' && c1 < = 'z') ? ('A' + c1 - 'a') : c1;
   return (c2);
}
```

Example:

```
int square (x)
int x;
{
   if (x < = 0)
   return 0;
   else
   return (x * x);
}
```

The absence of return statement indicates that no value is being returned.

STORAGE CLASSES

There are two different ways to characterize variables:

- by data types
- by storage class

Data types refer to the type of information while storage class refers to the lifetime of a variable and its scope within the program.

A variable in 'C' can have any one of the following four storage classes.

- Automatic Variable
- External Variable
- Static Variable
- Register Variable

Automatic Variable

The scope of automatic variable is confined to the function in which they are declared. They are created when the function is called and destroyed automatically when the function is exited, hence the name *Automatic*.

By default, a variable declared inside a function with storage class specification is an automatic variable. The value of automatic variables cannot be changed accidently by what happens in some other function in the program.

Example:

```
main( )
{
   int m = 1000;
   function2( );
   printf ("%d \n", m);
}
function1( )
{
   int m = 10;
   printf ("%d \n", m);
}
function2( )
{
   int m = 100;
   function1( );
   printf ("%d \n", m);
}
```

Output :
10
100
1000

External Variable

They are also known as *global variables*. They are not confined to a single function. Their scope extends from the point of definition through the remainder of the program.

They can be accessed from any function that falls within their scope. External variables are declared outside a function. If a local variable and global variable have the same name, the local variable will have precedence over global in the function where it is declared.

Example:

```
int count;
main
{
   count = 10;
   --------
   --------
   --------
```

```
}
function ( )
{
   int count = 0;
   --------
   --------
   count ++;
}
```

When the function references the variable count, it will be referencing only its local variable, not the global one. The value of the count in main( ) will not be affected.

Example:

```
/* illustration of working of global variable
int x;
main( )
{
   x = 10;
   printf ("x = %d \n", x);
   printf ("x = %d \n", fun1( ));
   printf ("x = %d \n", fun2( ));
   printf ("x = % d \n", fun3( ));
}
fun1( )
{
   x = x + 10;
   return x;
}
fun2( )
{
   int x = 1;
   return x;
}
fun3( )
{
```

```
    x = x+10;
    return (x);
}
```

Output :
$x = 10$
$x = 20$
$x = 1$
$x = 30$

External Declaration In the program segment discussed previously, the *main* cannot access the variable *y* as it has been declared after the main function. This problem can be solved by declaring the variable with the storage class **extern**.

Example:

```
main( )
{
   extern int y; /* external declaration */
   ---------
}
fun1()
{
   extern int y; /* external declaration */
   --------
}
int y ;          /*definition */
```

The external declaration of *y* inside the functions informs the compiler that *y* is an integer type defined somewhere else in the program.

Note: The external declaration does not allocate storage space for a variable.

Static Variable

Static variables are defined within a function in the same manner as automatic variables, except that the variable declaration must begin with the static storage class designation.

Example: static int *x*; or static float *y*;

A static variable is initialized only once, when the program is compiled. It is never initialized again.

A static variable may be either an internal type or an external type, depending on the place of declaration. Internal static variables are those which are declared inside a function. The scope of internal static variables extend upto the end of the function in which they are defined. Therefore, internal static

variables are similar to auto variables, except that they remain in existence (alive) throughout the remaining program. Therefore, internal static variables can be used to retain values between function calls.

Example:

```
/* Illustration of static variable */
main( )
{
   int i;
   for(i = 1; i < = 3; i++)  stat( );
}
stat( )
{
   static int x = 0;
   x = x + 1;
   printf("x = %d;\t", x);
}
```

Output : `x = 1;  x = 2;  x = 3;`

An external static variable is declared outside of all functions and is available to all the functions in that program. The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while a simple external variable can be accessed by other files also.

Register Variable

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of being kept in the memory (where, normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of programs. For example, loop control variables.

This is done as given below:

```
register int count;
```

Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, 'C' will automatically convert register variables into non-register variables once the limit is reached.

SCOPE AND LIFETIME OF DECLARATION

The following Table 11.1 gives a summarized detail of all the storage classes along with their visibility and lifetime, in any program.

Table 11.1 Storage classes and their details

Storage Class	*Where declared*	*Visibility (Active)*	*Lifetime (Alive)*
None	Before all functions in a file (may be initialized	Entire file plus other files where variable is declared with extern	Entire Program (Global)
Extern	Before all functions in a file (cannot be initialized)	Entire file plus ohter files where variable is declared with extern and the files where variables originally declared as global	Global
Static	Before all functions in a file	Only in that file	Global
Auto	Inside a function or a block	Only in that function or block	Unit end of fuction or block
Register	Inside a function or block	Only in that function or block	Unit end of function or block
Static	Inside a function	Only in that function	Global

PASSING PARAMETERS TO FUNCTIONS

Call by Value

Call by value means sending the values of the arguments to functions. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure to pass the value of an argument to a function is known as passing by value or **call by value.**

Example:

```
            /*  A simple 'C' program containing a function that alters
the value of its argument. */
            #include <stdio.h>
            main()
            {
               int a = 2;
               printf("\na = %d (from main, before calling the
function)",a);
               modify(a);
               printf("\na = %d (from main, after calling the
function)",a);
            }
```

```
modify (int a)
{
   a * = 3;
   printf("\na = %d (from the function, after being
modified)",a);
   return;
}
```

Output : *a* = 2 (from main, before calling the function)
a = 6 (from the function, after being modified)
a = 2 (from main, after calling the function)

The original value of *a* (that is,=2) is displayed when main is executed. This value is then passed to the function 'modify', where it is multiplied by three and the new value of the formal argument is displayed within the function. Finally, the value of *a* within main (the actual argument) is again displayed, after control is transferred back to function main from function 'modify'.

These results show that *a* is not altered within main, even though the corresponding value of *a* is changed within 'modify'.

Passing an argument by value has advantages and disadvantages.

On the positive side, it allows a single valued actual argument to be written as an expression rather than being restricted to a single variable. Moreover, if the actual argument is expressed as a single variable, it protects the value of this variable from alterations within the function.

On the negative side, it prevents information from being transferred back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information.

Call by Reference

Call by reference means sending the *addresses* of the arguments to a called function. In this method, the addresses of actual arguments in the calling function are copied into formal arguments of the called functions. Thus using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. Using a call by reference constructively it is possible to make a function return more than one value at a time, which involves the study of pointer which will be dealt-with in later chapters.

Passing Arrays to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In call by value, we pass values of array elements to the function while in call by reference we pass the name of the array, without any subscript, and the size of array elements, to the function.

Example:

-
```
/* passing by value * /
main( )
{
```

```
  int i;
  void display (int [ ], int);
  static int num [ ] = {50, 55, 60, 65, 60};
  for (i = 0; i < = 4: i++)
  display (num [i]);
}
void display (int m)
{
  printf ("%d \n", m);
}
```

Example:

- ```
  /* passing by reference * /
  #include <stdio.h>
  void main (void)
  {
    int i ;
    void display (int [ ], int); /*prototype for display function*/
    static int num [ ] = {50,55,60,65,60};
    display (num, 5);
  }
  void display (int num [ ], int n)
  {
    int i;
    for (i = 0; i < n - 1; i++)
    printf ("%d\n", num [i]);
  }
  ```

Functions Returning Values

The result of the function can be returned to the calling program. This defines the type-specifier in the function header. By default, in C, each function has a return type of "int". A function can return only one value. Arguments are passed by reference to overcome this limitation.

If you want the function to return a value of any other data type than the default, you need to mention it explicitly in the function header. The following example illustrates this:
  ```

Example:

```
/* demonstration of call by reference * /
main( )
{
  float largest( float [ ], int);
  static float vlaue [4] = {2.5, -4.75, 1.2, 3.67};
  printf ("%f \n", largest (value, 4));
}
float largest (float arr[ ], int n)
{
  int i; float max;
  max = arr [0];
  for (i = 1; i < n; i++)
  if (max < arr [i])
  max = arr [i];
  return (max);
}
```

COMMAND LINE ARGUMENTS

Every program must have a function main( ). Till now, we know that main( ) function takes no arguments. But the empty parenthesis in the main( ) may contain special arguments that allow parameters to be passed to the main( ) from the operating system.

Two arguments are passed to main( ) function

- argc : integer type argument
  indicates the number of parameters passed
- argv : array of strings.
  Each string in this array will represent a parameter that is passed to main( ).

Execution of the program is normally initiated by specifying the name of program at the operating system level. The program name is interpreted as an operating system command. Hence the line in which it appears is referred to as command line.

In order to pass one or more parameters to the program when the program execution is initiated, the parameters must follow the program name on command line.

Example: program_name parameter1 parameter2 - - - - parametern.

All the parameters including program name are stored in the array argv, and the number of elements are stored in argc.

Example:

```
main (argc, argv)
int argc;
char * argv [ ];
{
   int i;
   for (i = 0; i < argc; i++)
   printf ("%s \n", argv [i]);
}
```

On command line the following command is given:

C:\>prg first second third.

RECURSION IN FUNCTION

Any function in a 'C' program that can call itself is called a **recursive function**. The number of recursive calls is limited to the size of the stack. Each time the function is called, new storage is allocated for the parameters and for the **auto and register** variables so that their values in previous unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Note that variables declared with static storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computation in which each action is stated in terms of the previous result.

In order to solve a problem recursively, two conditions must be satisfied:

- The problem must be written in recursive form.
- The problem statement must include a stopping condition.

Example:

```
/* To calculate the factorial of an integer recursively * /
#include <stdio.h>
main( )
{
   int n;
   long int fact (int);
   printf ("\n n = ");
   scanf ("%d", &n);
```

```
    printf ("\n n! = % ld" fact (n));
}
long int fact (int n)
{
    if (n < = 1)
    return 1;
    else
    return (n * fact (n - 1));
}
```

The Tower of Hanoi

The Towers of Hanoi is a well-known children's game, played with three poles and a number of different sized disks. Each disk has a hole in the centre, allowing it to be stacked on any of the poles. Initially, the disks are stacked on the leftmost pole in the order of decreasing size, the largest at the bottom and the smallest on the top.

The object of the game is to transfer the disks from the leftmost pole to the rightmost pole, without ever placing a larger disk on top of a smaller disk. Only one disk may be moved at a time, and each disk must always be placed around one of the poles.

The general strategy is to consider one of the poles to be the origin, and another to be the destination. The third pole will be used for intermediate storage, thus allowing the disks to be moved without placing a larger disk over a smaller one. Assume there are n disks, numbered from smallest to largest.

If the disks are initially stacked on the left pole, the problem of moving all n disks to the right pole can be stated in the following recursive manner:

- Move the top $n - 1$ disks from the left pole to the centre pole.
- Move the n^{th} disk (the largest disk) to the right pole.
- Move the $n - 1$ disks from the centre pole to the right pole.

The problem can be solved in this manner for any value of n greater than 0 ($n = 0$ represents a stopping condition).

The program consists of *main( )*, which merely reads in a value for n and then initiates the computation by calling a function *transfer*. In this first function call, the actual parameters will be specified as character constants, that is,

transfer (n, 'L', 'R', 'C');

where
- n number of disks
- L represents the left pole
- R represents the right pole
- C represents the centre pole

This function call specifies the transfer of all n disks from the leftmost pole (the origin) to the rightmost pole (the destination), using the centre pole for intermediate storage.

Example:

```
            /* Program to solve the Towers of Hanoi  problem using
recursion. */
            #include <stdio.h>
            main( )
            {
               void transfer(int, char, char, char);
               int n;
               printf("\nWelcome to the Towers of Hanoi");
               printf("\nHow many disks?");
               scanf("%d", &n);
               printf("\n");
               transfer(n, 'L', 'R', 'C');
            }
            void transfer(int n, char from, char to, char temp)
            /* transfer n disks from one pole to another number of
disks; from = origin; to = destination; temp = temporary storage    */
            {
               if (n > 0)
               {  /* move n - 1 disks from origin to temporary  */
                  transfer(n - 1, from, temp, to);
                  /* move nth disk from origin to destination */
                  printf("\nMove disk %d from %c to %c\n", n, from, to);
                  /* move n - 1 disks from temporary to destination */
                  transfer(n - 1,temp,to,from);
               }
               return;
            }
```

The function *transfer* receives a different set of values for its arguments each time the function is called. These sets of values will be pushed onto the stack so that each set is independent of the others. They are then popped from the stack at the proper time during the execution of the program.

It is this ability to store and retrieve these independent sets of values that allows the recursion to work.

Sample programs

- To generate positive prime numbers:

```
#include <stdio.h>
main( )
{
   int no, x min, x max, J;
   printf ("kEnter the lower and upper limit of the numbers :");
   scanf ("%d %d", & xmin, &xmax);
   printf ("\n prime numbers are :");
   for (J = x min; J < = x max; J++)
   prime (J);
}
prime (int no)
{
   int i; flag = 0
   if (no < = 3) printf ("\n % d", no);
   else
   {
      for (i = 2; i < = (no/2); i++)
      {
         if (no % i = = 0)
         {
            flag = 0; break;
         }
         else flag = 1;
      }
      if (flag = = 1) printf ("\n % d", no);
   }
}
```

- Sorting a list of integers (using quick sort)

```
#include <stdio.h>
# define MAX 10
void main (void)
```

```
{
  int i,n, a[MAX]
  void quicksort (int [ ], int, int) ; /*prototype for quicksort*/
  void partition (int [ ], int, int) ; /*prototype for partition*/
  printf ("Enter the total number of integers in the list\n");
  scanf ("%d", &n);
  printf ("Enter the elements of the list \n");
  for (i = 0; i < n; i++);
  scanf ("%d", &a[i]);
  quicksort (a,o,n);
  printf ("The sorted elements are \n");
  for (i = 0; i < n; i++);
  printf ("\n%d", a[i]);
}
/* function quicksort */
void quicksort (int a[ ], int lower, int upper)
{
  int i ;
  if (upper >lower) /* array contains more than one element */
  {
        i = partition (A, lower, upper);
        quicksort (A, lower, i - 1);
        quicksort (A, i + 1, upper);
  }
}
/* function partition */
int partition (int a[], int lower, int upper)
{
  int i,p,q,t;
  p=lower;
  q=upper + 1;
  i=A[lower];
  while (q>=p)
```

```
{
        while (A[++ p] < i);
        while (A [- -q] > i);
        if (q  > p)
        {
                t = A [p];
                A[p]=A[q];
                a[q] = t;
        }
        t = A [lower];
        a[lower] = a[q];
        a[q] = t ;
        return q ;
}
```

Summary

- 'C' program is a collection of one or more functions.
- A function is defined when the function name is followed by a pair of braces in which one or more statements may be present.
- A function is called when function name is followed by a semicolon.
- A function may be placed either before or after main( ).
- A function returns integer value by default.
- A functions may have more than one exit points.
- A return statement can occur anywhere within the body of the function.
- A function is treated as an external variable.
- A variable declared inside a function is known only to that function.
- A global variable is visible from the point of declaration to the end of the program.
- Static variable retains its value even when one function is exited.
- When a function calls itself repeatedly, it is called recursion.
- Values are passed to the function in two ways—call by value and call by reference.

Review Exercise

Multiple-Choice Questions

1. The function main( ) is a/an:
   (a) user defined function
   (b) system defined function
   (c) external function
   (d) none of these
2. The default return type of any function in 'C' language is
   (a) void
   (b) int
   (c) long
   (d) NULL
3. Recursive functions are those functions which
   (a) are written once and not modified again
   (b) call more than one functions
   (c) call themselves
   (d) are system defined
4. How many storage classes for variables are allowed in 'C' language?
   (a) 2
   (b) 4
   (c) 5
   (d) 10
5. For passing arguments to the main( ) function how may argreements are required?
   (a) One
   (b) Two
   (c) Three
   (d) None of these

State Whether True or False

1. A function can return more than one value to the calling portion of the program via return.
2. The arguments appearing in the function call are referred to as formal arguments.
3. Any 'C' function by default returns int value.
4. Every program must have a function main( ).
5. During call by reference, the original data values of the parameters do not change.

Fill in the Blanks

1. A function is treated as an _____ variable.
2. Main( ) is the first _____ function.

3. _____ variables maintain their values during multiple function calls.
4. A function can have _____ return statements, each containing different expressions.
5. Functions help in implementing the concept of _____ of code.

Descriptive Questions

1. State several advantages of the use of functions.
2. What is meant by a function call? From what parts of a program can a function be called?
3. What is the relationship between formal arguments and actual arguments?
4. Explain the difference between a function declaration and a function definition.
5. What is the purpose of the return statement?
6. What will be the output of the following programs:

(a)
```
/ * file name my.c */
void main (int arg c, char * argv[ ])
{
   int count;
   printf ("argc = %d", argc);
   fot (count = 0; count <arg c; ++ count)
   {
      printf ("argv [%] = %s", count, argv [count]);
   }
}
on command line the command is
C:\ my A B C
```

(b)
```
main( )
{
   int k = 35, z;
   z = func (k);
   printf ("z = %d", z);
}
func (int m)
{
   ++m;
   return (m = func1(++m));
}
```

```
    func1 (int m)
    {
       m++;
       return m;
    }
(c) main( )
    {
       int i = 3; k, l;
       k = plus (++i);
       l = plus (i++);
       printf ("i = %d, k = %d, l = %d", i, k, l);
    }
    plus (int J)
    {
       ++J;
       return (J);
    }
(d) main( )
    {
       void message( );
       int m;
       printf ("m before call = %d \n", m);
       m = message( );
       printf ("m after call = %d \n", m);
    }
    void message ( )
    {
       printf ("\n Inside the function");
    }
(e) main( )
    {
       int i;
```

```
printf ("UPTEC \n");
for (i = 1; i < = 10; i++);
main( );
```

```
        3                               4
    3   2   3                       4   3   4
3   2   1   2   3               4   3   2   3   4
    3   2   3               4   3   2   1   2   3   4
        3                       4   3   2   3   4
                                    4   3   4
                                        4
```

```
}
```

7. Write a recursive function to calculate the Fibonacci series.
8. Write a function for string concatenation.
9. Write a function to determine whether the number is prime or not.
10. Write a function to accept integer N and display the following pattern:
    if $N = 3$ if $N = 4$
11. Write a function that copies one string into another.
12. Write a recursive function to add each digit of a number.
13. Write a function that returns the total number of words of a given string.
14. Write a function that compares two strings A and B and returns:
    -1 if $A < B$
    0 if A equals B
    1 if $A > B$

CHAPTER

12

Pointers

Key Features

- Introduction to Pointers
- Pointer Notation
- Pointer Declaration and Initialization
- Accessing Variable through Pointer
- Difference between Array and Pointer
- Pointer Expressions
- Pointers and One Dimensional Arrays
- Malloc Library Function
- Calloc Library Function
- Pointers and Multi-dimensional Arrays
- Arrays of Pointers
- Pointer to Pointers
- Pointers and Functions
- Functions with a Variable Number of Arguments

Pointer is an important feature of 'C' language. It is a powerful and handy tool of 'C' Language. Pointer is a variable which contains the location of a data item rather than the value.

As the pointers are closely associated with the arrays, they provide an alternate way to access the individual element of the array.

In this chapter we are going to discuss pointers in detail.

INTRODUCTION TO POINTERS

Computers use their memory to store instructions of the programs and the values of the variables. The memory is a sequential collection of storage cells. Each cell has an address associated with it. Whenever we declare a variable, the system allocates, somewhere in the memory, a memory location and a unique address is assigned to this location.

Pointer is a variable which can hold the address of a memory location rather than the value at the location. Consider the following statement:

```
int num = 84;
```

This statement instructs the system to reserve a 2-byte memory location and put the value 84 in that location. Assume that system allocates memory location 1001 for num. Diagramatically, it can be shown as in Figure 12.1.

As the memory addresses are whole numbers, they can be assigned to some other variable.

Let ptr be a variable which holds the address of the variable num.

Thus, we can access the value of num by the variable ptr. We can say "ptr points to num". Diagrammatically, it can be shown as in Figure 12.2.

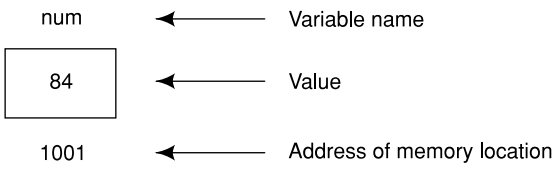


Fig. 12.1 *Memory representation of a variable*

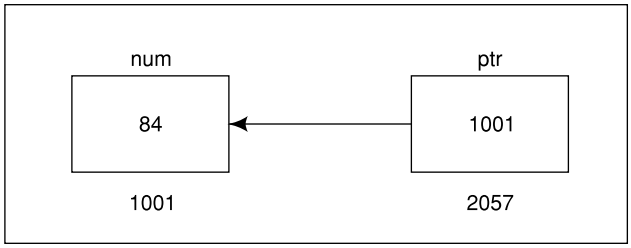


Fig. 12.2 *Pointer and variable in the memory*

POINTER NOTATION

The actual address of a variable is not known immediately. We can determine the address of a variable using "address of" operator (&). We have already seen the use of "address of" operator in the scanf( ) function.

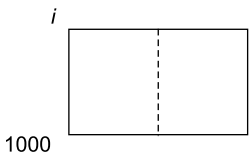
Another pointer operator available in 'C' is "*" called "value at" operator. It gives the value stored at a particular address. This operator is also known as "indirection operator".

Example:

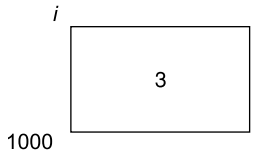
```
main( )
{
    int i = 3;
    printf ("\n Address of i : = %u", &i);  /* returns the address * /
    printf ("\t value i = %d", *(&i));  /* returns the value at address of i */
}
```

Explanation

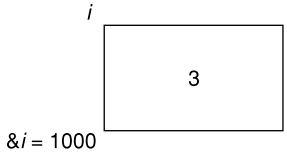
Let *i* refer to a 2 byte memory location 1000.



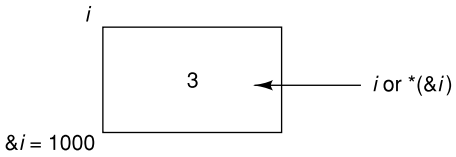
The statement $i = 3$ will assign value at this location.



&*i* will return the address, which *i* is refering



and *(&*i*) returns the value at this address i.e.3.



POINTER DECLARATION AND INITIALIZATION

Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them.

The declaration of a pointer variable takes the following form:

data_type *pt_name;

This tells the compiler three things about the variable *pt_name*.

- The *asterisk (*)* tells that the variable *pt_name* is a pointer variable.
- *pt_name* needs a memory location.
- *pt_name* points to a variable of type *data_type*.

Example: int *p;

declares the variable *p* as a pointer variable that points to an *integer* data type. The type *int* refers to the data type of the variable being pointed to by *p* and not the type of the value of the pointer.

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as *p = &quantity;* which causes *p* to point to *quantity.* That is, *p* now contains the address of *quantity*. This is known as *pointer initialization*. Before a pointer is initialized, it should not be used. A pointer variable can be initialized in its declaration itself.

Example: int *x*, **p*=&*x*;

statement declares *x* as an integer variable and *p* as a pointer variable and then initializes *p* to the address of *x*. This is an initialization of *p,* not **p*. On the contrary, the statement *int *p = &x, x;* is *invalid* because the target variable *x* is declared first.

ACCESSING A VARIABLE THROUGH A POINTER

Consider the following statements:

int *q*, * *i*, *n*;

q = 35;

i = &*q*;

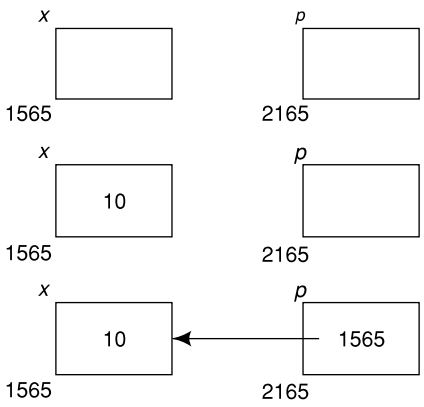
n = **i*;

i is a pointer to an integer containing the address of *q*. In the fourth statement we have assigned the value at address contained in *i* to another variable *n*. This way, indirectly we have accessed the variable *q*, through *n*, using pointer variable *i*.

```
main( )
{
   int x, *p;

   x = 10;

   p = &x;
   /* displays address of x */
   printf ("\n%u%u", &x,p);
   /* displays value at x */
   printf ("\n%d%d", x,*p);
   /* displays address of p */
   printf ("\n%u", &p);
}
```



Output:

```
1565        1565
10          10
2165
```

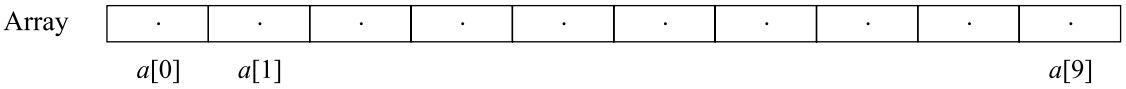
DIFFERENCE BETWEEN ARRAY AND POINTER

Pointers and arrays are strongly inter related in 'C'. Both of them are capable of performing the same functions.

When we declare an array

int *a* [10];

ten consequtive memory blocks of integer type are allocated and named *a*[0], *a*[1], . . . *a*[9]



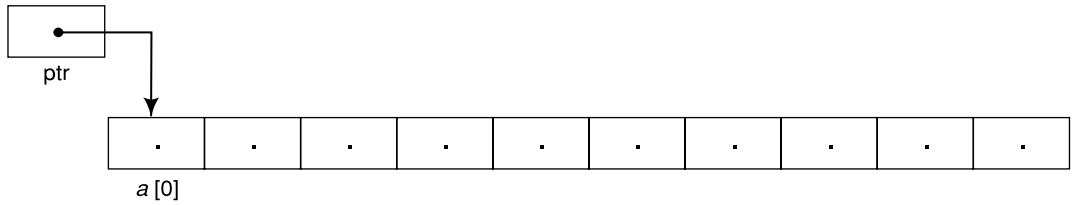
The i^{th} element of the array is represented by the notation *a*[*i*]. Let us declare an integer type pointer as

int *ptr

then the assignment

ptr = &*a*[0];

sets ptr to print to the element zero of *a*, i.e., the address of *a*[0] is stored in ptr.



Suppose we have defined an integer type variable *y* and make an assignment

y = *ptr

this will copy the contents of a[0] into *y*.

If ptr points a particular element of an array then ptr + 1 points to the next element, ptr + *i* points *i* elements after ptr, and ptr – *i* points *i* elements before. Thus if ptr points to *a*[0]

*(ptr + 1)

refers to the contents of *a*[*i*], ptr+ *i* is the address of *a*[*i*], and * (ptr + *i*) is the contents of *a*[*i*].

There is one more difference between a pointer and an array that should be kept in mind. A pointer is a variable, so ptr = *a* and ptr ++ are legal. But an array name is not a variable, hence constructions like *a* = ptr and *a*++ are illegal.

POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers.

Example: If *p*1 and *p*2 are properly declared and initialized pointers, then following statements are valid.

```
y = *p1 * *p2;
sum = sum + * p1;
```

```
main ( )
```

```
{
   int a,b, sum1, sum2, *p1,*p2;
   a = 15;
   b = 25;
   p1 = &a;
   p2 = &b;
   sum1 = a + b;
   sum2 = *p1 + *p2;
   printf ("sum1=%d sum2=%d", sum1, sum2)
}
```

Output:

sum1 = 30
sum2 = 30

1

'C' allows us to add integers to pointers as well as subtract one pointer from another. Shorthand operators like sum + = *p2; can also be used with the pointers for arithmetic expressions. The following operations can be performed on a pointer:

- Addition of a number to a pointer.

  Example: int *i* = 4, **J*, **k*;
  J = &*i*;
  J = *J*+1;
  J = *J*+9;
  k = *J*+3;

- Subtraction of a number from a pointer.

  Example: int *i* = 4, **J*, **k*;
  J = &*i*;
  J = *J* - 2;
  J = *J* - 5;
  k = *J* - 6;

- Subtraction of one pointer from another provided that both variables point to elements of same array. The resulting value indicates the number of bytes separating corresponding array elements.

However, the following operations are not allowed on pointers:

- Multiplication of pointer with a constant.
- Addition of two pointers.
- Division of pointer with a constant.

For example:

```
main ( )
{
    int i, *j, *k;
    i = 4;
    j = &i;
    k=j+1;
    printf ("%u %u %u", &i, j, k);
}
```

Output:

If address of *i* is 1000 then the output will be

1000 1001 1002

Explanation:

Addition of 1 to the pointer *j* does not mean an arithmetic operation, but it refers to the next block from the given base address.

i is an integer variable and occupies 2 bytes addressed 1000 and 1001, where 1000 is the base or starting location. Now 1001 is reserved for the variable *i*, therefore, 1000 and 1001 form a block.

Addition of 1 to the base address will return the base address of next block, that is, address of the byte not occupied by current variable, which is 1002.

Pointer Increment/Decrement and Scale Factor

The pointers can be incremented as

$p1 = p2 + 2;$

Similarily, pointers can be decremented as

$p1 = p2 - 1;$

Short hand operators can also be used to increment or decrement the pointers.

$p1 ++ ;$

$++ p2 ;$

$p2 -- ;$

$-- p2 ;$

An expression like

$p1 ++;$

will cause the pointer $p1$ to point to the next memory location of its type.

For example, if $p1$ is an integer with an initial value 1000, then after the operation $p1 = p1 + 1$, the value of $p1$ will be 1002. That is, when we increment a pointer, its value is increased by the length of the data type that it points to.

This length is called the *scale factor.*

Pointer Comparison

Pointers can be compared using relational operators. Expressions such as $p1 > p2$, $p1 == p2$ and $p1 != p2$ are allowed. Such comparisons are useful when both pointer variables point to elements of the same array.

Any comparison of a pointer that refers to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.

POINTERS AND ONE-DIMENSIONAL ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

The array declared as

static int *x*[5] = {1, 2, 3, 4, 5};

is stored in the memory as shown in Figure 12.3.

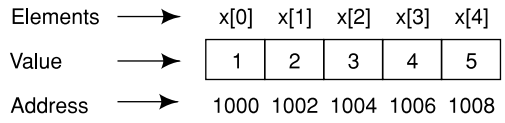


Fig. 12.3 *Array representation in the memory*

The name *x* is defined as a constant pointer pointing to the first element, *x*[0] and, therefore, the value of *x* is 1000, the location where *x*[0] is stored. That is,

x, &*x*[0] and 1000 are same.

If we declare *p* as an *integer pointer*, then we can make the pointer *p* to point to the *array x* by the assignment statement

p = *x* ; which is equivalent to *p* = & *x* [0];

Now we can access every value of *x* using *p*++ to move from one element to another. The relationship between *p* and *x* is shown below:

p = & *x* [0] (=1000)

p+1 = & *x* [1] (=1002)

p+2 = & *x* [2] (=1004)

p+3 = & *x* [3] (=1006)

The address of an element is calculated using its *index* and the *scale factor* of the data type.

Example: Address of *x*[3] = Base Address + (3 × Scale Factor of *int*)

= 1000 + (3*2)

= 1006

When handling arrays, instead of using array indexing, we can use pointers to access array elements, as *(*p*+3) gives the value of *x*[3]. The pointer accessing method is much faster than array indexing. *&x*[*i*] and (*x*+*i*) both represent the address of the i^{th} element of *x*. *x*[*i*] and *(*x*+*i*) both represent the contents of that address, that is, the value of the i^{th} element of *x*. The two terms are interchangeable.

When assigning a *value* to an array element such as *x*[*i*], the left side of the assigned statement may be written as either *x*[*i*] or as *(*x*+*i*). Thus, a value may be assigned to an array element, or it may be assigned to the memory area whose address is that of the array element. While assigning an *address* to an identifier, a pointer variable must appear on the left side of the assignment statement. Expressions such as *x,* (*x*+1) and *&x[i]* cannot appear on the left side of an assignment statement because it is not possible to assign an arbitrary address to an array name or an array element. The address of an array cannot arbitrarily be altered, so expressions such as ++*x* are not permitted.

```
main ( )
{
   int brr [5];
   int arr [5]; /* will assign 10 bytes of contiguous memory */
   int i;
   for (i = 0; i < 5; i++)
   {
      printf ("Enter 1st array element");
      scanf ("%d", &arr[i]);
      printf ("Enter 2nd array element");
      scanf ("%d", (brr+i));
   }
   for (i=0;i<5;i++)
   printf ("\n%d element of arr=%d, brr=%d", arr[i], *(brr+i));
}
```

Input:

```
   arr : 10, 20, 30, 40, 50
   brr : 50, 40, 30, 20, 10
```

Output:

```
   0 element of arr = 10, brr = 50
   1 element of arr = 20, brr = 40
   and so on.
```

MALLOC LIBRARY FUNCTION

Function	Allocates main memory
Syntax	void *malloc (size_t *size);*
Prototype in	stdlib.h, alloc.h
Remarks	*malloc* allocates a block of *size* bytes from the 'C' memory heap. It allows a program to allocate memory explicitly, as it is needed and in the exact amount needed.
Return value	On success, *malloc* returns a pointer to the newly allocated block of memory. If enough space does not exist for the new block, it returns null. The contents of the block are left unchanged. If the argument *size = = 0*, *malloc* returns null.
Portability	*malloc* is available on UNIX systems and is compatible with ANSI C.

The use of a pointer variable to represent an array requires some sort of initial memory assignment before the array elements are processed. Generally, such initial memory allocations are accomplished by using *malloc* library function.

Numerical array elements cannot be assigned initial values if the array is defined as a pointer variable. Therefore, a conventional array definition is required if initial values will be assigned to the elements of a numerical array.

It is possible to define *x*, which is a one-dimensional, 10-element array of integers, as a pointer variable rather than as an array. We can write *int *x*; instead of *int x*[0];.

However, *x* is not automatically assigned a memory block when it is defined as a pointer variable, though a block of memory large enough to store 10 integer quantities will be reserved in advance when *x* is defined as an array.

To assign sufficient memory for *x*, we can make use of the library function *malloc*, as follows:

x = malloc (10 * sizeof(int));

This function reserves a block of memory whose size (in bytes) is equivalent to the size of an integer quantity. The function returns a pointer to a character. To be consistent with the definition of *x*, we really want a pointer to an integer, however, characters and integers are equivalent in 'C'. Therefore, the statement is acceptable as defined previously, though we could include a type cast to be on the safe side, that is, *x=(int*) malloc (*10**size of (int));*

The allocation of memory in this manner, as it is required, is known as *dynamic memory allocation.*

If the declaration is to include the assignment of initial values, then *x* must be defined as an array rather than as a pointer variable.

However, character-type pointer variable can be assigned an entire string as a part of the variable declaration. Thus, a string can conveniently be represented by either a one-dimensional character array or by a character pointer.

```
main( )
{
   int size, *x, i=0, len;
   clrscr( );
```

```
    printf ("\n enter the no of elements");
    scanf("%d", &size);
    x = (int*) malloc (size*sizeof(int));
    for (;i < size; i++)
    scanf ("%u", (x + i));
    for(i=0; i < size; i++)
    scanf("%d", * (x + i));
    for (i=0; i < size; i++)
    printf ("%d", *(x + i));
    getch ( );
}
```

CALLOC LIBRARY FUNCTION

Function	Allocates main memory
Syntax	void *calloc(size_t *nitems,* size_t *size*);
Prototype in	stdlib.h, alloc.h
Remarks	*calloc* provides access to the 'C' memory heap. It allocates a block of size *n* items×*size*. The block is cleared to 0.
Return value	*calloc* returns a pointer to the newly allocated block. If not enough space exists for the new block, or *n* items or *size* is 0, *calloc* returns null.
Portability	*calloc* is available on UNIX systems and is compatible with ANSI C. It is also defined in K&R C.

The *calloc ( )* function works exactly similar to *malloc ( )* except for the fact that it needs two arguments as against the one required by *malloc ( )*.

Example: int *p;

p = (int *) calloc (10, 2);

Another minor difference between *malloc ( )* and *calloc ( )* is that by default the memory allocated by *malloc ( )* contains garbage values whereas those allocated by *calloc ( )* contain all zeroes.

Realloc library function

Function	reallocates main memory, changing the size of the storage allocated to an object
Syntax	void * realloc (void *p, size * size)
Prototype in	stdlib.h
Remarks	The function realloc changes the size of the object pointed to by "*p*" to "size".
Return value	it returns a pointer to the new storage and null if it is not possible to resize the object

Free library function

Function	deallocates storage space
Syntax	void free (void *p)
Prototype in	stdlib.h
Remarks	The function **free** deallocates the part of the main memory pointed to by "p" where "p" is a pointer to a space allocated by *malloc*, *calloc*, or *realloc*. If "p" is a null pointer 'free' does nothing.

POINTERS AND MULTI-DIMENSIONAL ARRAYS

A two-dimensional array can be defined as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array declaration can be written as:

data_type (*ptvar)[expression2];

rather than *data_type array [expression1] [expression2];*

This can be generalized to higher dimensional arrays, that is,

data_type (*ptvar) [expression2] [expression3] ... [expressionN];

replaces *data_type array [expression1] [expression2] ... [expressionN];*

In these declarations *data_type* refers to the data type of the array, *ptvar* is the name of the pointer variable, *array* is the corresponding array name, and *expression*1, *expression*2 ------ *expressionN* are positive valued integer expressions that indicate the maximum number of array elements associated with each subscript.

Example:

- Suppose *x* is a two-dimensional integer array that has 10 rows and 20 columns. We can declare *x* as *int (*x) [20];* rather than *int x[10] [20];*

In the first declaration, *x* is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays. Thus, *x* points to the first 20-element array, which is actually the first row (row 0) of the original two-dimensional array. Similarly, (*x* + 1) points to the second 20-element array, which is the second row (row 1) of the original two-dimensional array, and so on, as illustrated below in Figure 12.4.

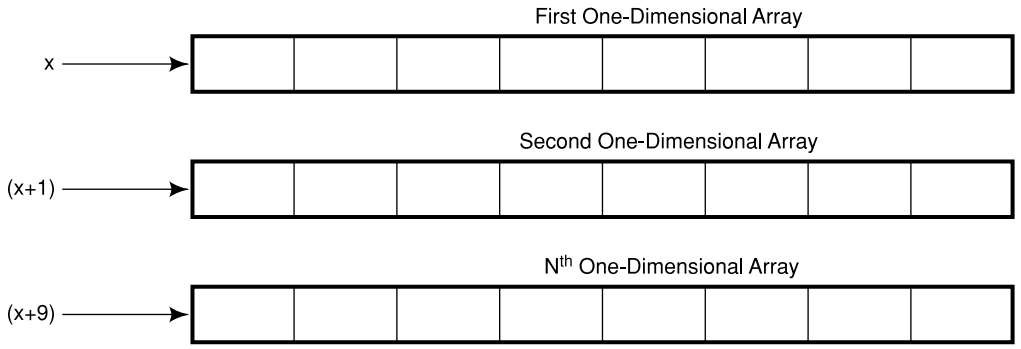


Fig. 12.4 *Two-dimensional array in the memory*

Now consider a three-dimensional floating-point array *t*.
This array can be defined as *float* (*t) [20] [30]; rather than
float t [10] [20] [30];

In the first declaration, *t* is defined as a pointer to a group of contiguous, two-dimensional, 20 × 30 floating-point arrays. Hence, *t* points to the first 20 × 30 arrays, (*t* + 1) points to the second 20 × 30 array, and so on.

- An individual array element within a multi-dimensional array can be accessed by repeatedly using the *indirection* operator. Usually, however, this procedure is more awkward than the conventional method to access an array element. The following example illustrates the use of the indirection operator.
  Suppose *x* is a two-dimensional integer array that has 10 rows and 20 columns, as declared in the previous example, the item in row 2, column 5 can be accessed by writing either *x*[2][5] or *(*(*x* + 2) + 5)
  The second form requires some explanation. First, note that *x* is a pointer to row 0 so (*x* + 2) is a pointer to row 2. Therefore, the object of this pointer, *(*x* + 2), refers to the entire row. Since row 2 is a one-dimensional array, *(*x* + 2) is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence, (*(*x* + 2) + 5) is a pointer to element 5 (the sixth element) in row 2. The object of this pointer, *(*(*x* + 2) + 5), therefore, refers to the item in column 5 of row 2, which is *x*[2][5], as illustrated in Figure 12.5.

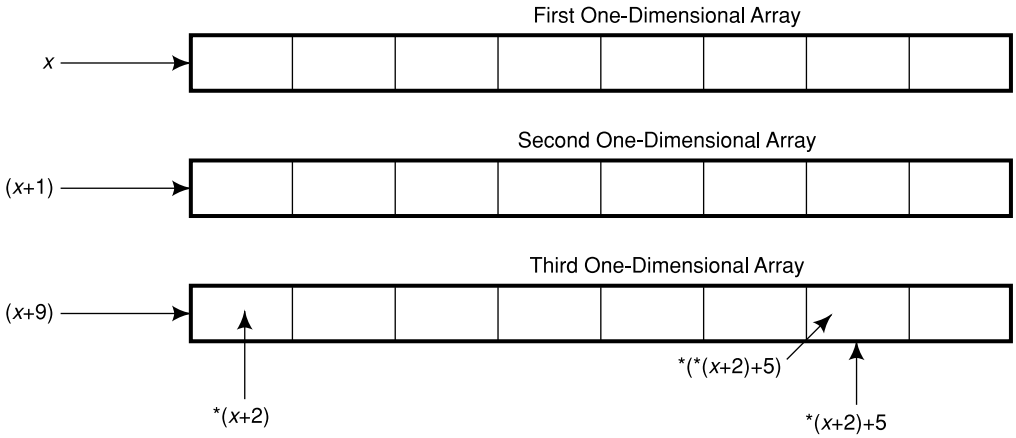


Fig. 12.5 *Pointer representation of a 2-D array*

ARRAYS OF POINTERS

A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such situations, the newly defined array will have one less dimension than the original multi-dimensional array. Each pointer will indicate the beginning of a separate (*n* – 1) dimensional array.

In general terms, a two-dimensional array can be defined as one dimensional array of pointers by writing.

*data_type *array[expression1];*

rather than the conventional array definition *data_type array[expression1] [expression2];*

Similarly, an *n*-dimensional array can be defined as an $(n - 1)$ dimensional array of pointers by writing

data_type *array[expression1][expression2]...[expression$N$ – 1];

rather than the conventional array definition *data_type array[expression1] [expression2]... [expressionN];*

In these declarations *data_type* refers to the data type of the original *n*-dimensional array, *array* is the array name, and *expression1, expression2, . . ., expressionN* are positive-valued integer expressions that indicate the maximum number of elements associated with each subscript.

The array name and its preceding asterisk are not enclosed in parenthesis in this type of declaration. Thus, a right-to-left rule first associates the pairs of square brackets with the *array,* defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

Moreover, note that the last (the right most) expression is omitted when defining an array of pointers, whereas the first (the left most) expression is omitted when defining a pointer to a group of arrays.

When an *n*-dimensional array is expressed in this manner, an individual array element within the *n*-dimensional array can be accessed by a single use of the indirection operator. The following example illustrates how this is done:

Suppose that *x* is a two-dimensional integer array having 10 rows and 20 columns, we can define *x* as a one dimensional array of pointers by writing

int *x[10];

Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row, and so on. The number of elements within each row is not explicitly specified.

An individual array element, such as x[2][5], can be accessed by writing *$(x$[2] + 5*)*. In this expression, x[2] is a pointer to the first element in row 2, so that (x[2] + 5) points to element 5 (actually, the sixth element) within row 2. The object of this pointer, *(x[2] + 5), therefore refers to x[2] [5].

These relationships are illustrated below in Figure 12.6.

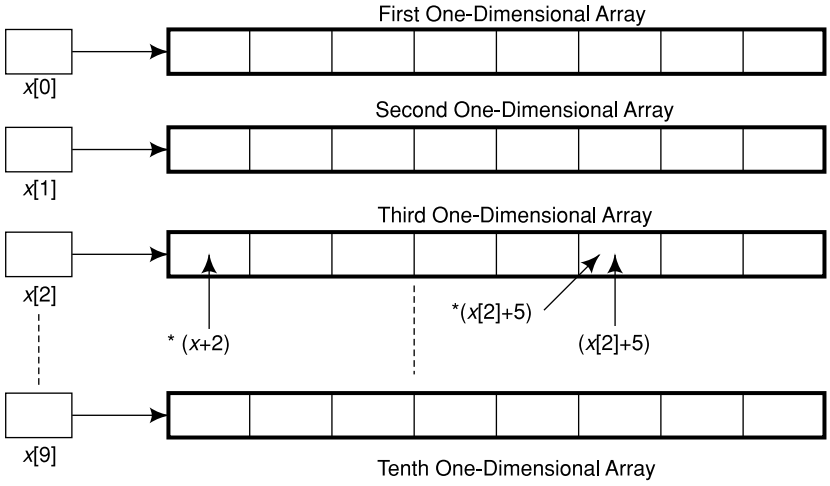


Fig. 12.6 *Array of pointers*

POINTER TO POINTERS

A pointer is a variable which contains the address of a variable. This variable itself could be another pointer. Declaration of such variables requires increase in number of "*" (the indirection operator) as prefix to the variable name. For example:

```
main( )
{
   int i = 3; int *j, **k;
   j = &i; k = &j;
   printf ("Address of i = %d\n", &i);
   printf ("Address of i = %d\n", j);
   printf ("Address of i = %d\n", *k);
   printf ("Address of j = %d\n", &j);
   printf ("Address of j = %d\n", *k);
   printf ("Address of k = %d\n\n", &k);
   printf ("Value of j = %d\n", j);
   printf ("Value of k = %d\n", k);
   printf ("Value of i = %d\n", i);
   printf ("Value of i = %d\n", *(&i));
   printf ("Value of i = %d\n", *j);
   printf ("Value of i = %d\n", **k);
}
```

The following Figure 12.7 would help you in tracing out how the program prints the output that follows.

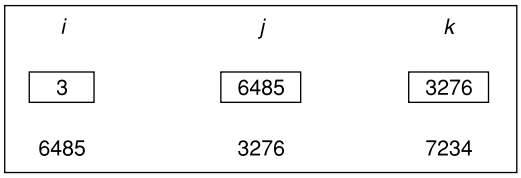


Fig. 12.7 *Pointer to pointer*

Output : Address of *i* = 6485
Address of *i* = 6485
Address of *i* = 6485

Address of j = 3276
Address of j = 3276
Address of k = 7234
Value of j = 6485
Value of k = 3276
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3

POINTERS AND FUNCTIONS

All 'C' functions have an identical template, composed of a unique name, a return type, argument(s), and a body. Function names are only identifiers that are treated as pointers without explicitly being declared. The return type of a function is understood to represent its value. A function body contains expressions, keywords, and statements that control the actions that are to be performed. Functions may optionally be declared to accept arguments. They can be passed as arguments "by value" and "by reference".

When an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. If x is an array, when we call *sort(x),* the address of x[0] is passed to the function *sort*. The function uses this address to manipulate the array elements. The address of a variable can be passed as an argument to a function in the normal fashion. When an address is passed to a function, the parameters receiving the address should be pointers. The process of calling a function using pointers to pass the address of variables is known as "call by reference". The function which is called by "reference" can change the value of the variable used in the call.

Consider the following example:

```
main ( )
{
   int x;
   x = 20;
   change (&x);
   printf ("%d \n", x);
}
change (p)
int *p;
{
   *p = *p + 10;
}
```

When the function *change( )* is called, the address of the variable *x*, not its value is passed into the function *change ( )*. Inside *change( ),* the variable *p* is declared as a pointer and therefore *p* is the address of the variable *x*.

Using "call by reference" intelligently we can make a function return more than one value at a time, which is not possible ordinarily.

```
main ( )
{
   int radius;
   float area, perimeter;
   printf ("\nEnter radius of a circle :");
   scanf ("%d", &radius);
   areaperi (radius, &area, &perimeter);
   printf ("\nArea = %f", area);
   printf("\nPerimeter = %f", perimeter);
}
areaperi (int r, float *a, float *p)
{
   * a = 3.14 *r * r;
   * p = 2 * 3.14 * r;
}
```

In this program there is a mixed call as the value of *radius* is being passed, but for the *area* and *perimeter,* addresses are passed.

Since the addresses are passed, any change made in values stored at the addresses contained in the variables *a* and *p* would make the change effective even in main ( ).

Thus the limitation of the *return* statement, which can return only one value from a function at a time has been overcome, and it is now possible to return two values from the called function *areaperi ( ),* as in the example program.

Functions Returning Pointers

A function can return a pointer just as it returns an *int,* a *float*, a *double,* or any other data type. To make a function return a pointer' it has to be explicitly mentioned in the calling function as well as in the function declaration. The following program illustrates this.

```
main( )
{
   int *p; int *fun( );
   p = fun( );
```

```
   printf ("\n%u", p);
}
int *fun( )
{
   int i = 20;
   return (&i);
}
```

This program just indicates how an integer pointer can be returned from a function.

The following program copies one string into another and returns the pointer to the target string:

```
main( )
{
   char *str;
   char *copy( );
   static char source[ ] = "UPTEC";
   char target[5] ;
   str = copy (target, source);
   printf ("%s", str);
}
char *copy (char *t, char *s)
{
   char *r;
   r = t;
   while (*s!='\0')
   {
      *t = *s;
      t++;
      s++;
   }
   *t = '\0';
   return (r);
}
```

The base address of *source* and *target* strings are sent to *copy( )*. In the *copy( )* function the ***while*** loop copies the characters in the source string into the target string. Since during copying *t* is continuously incremented, before entering into the loop the initial value of *t* is safely stored in the character pointer *r*. Once the copying is over, this character pointer is returned to *main ( )*.

Pointers to Functions

A function, like a variable, occupies memory and has a unique address to that location (the starting address). Therefore, a pointer to a function can be called the address of the code that will execute when the function is called.

Declaring and Initializing a Function Pointer A function pointer is declared as:

```
data type (*pointername) (<parameters>);
```

Example:
```
int(*ptr) ( );
```
or
```
int (*p) (int, int);
```

Parenthesis around the pointername is necessary to differentiate the function pointer from the function returning pointer, that is, int *ptr( );

A function pointer can be initialized by assigning the base address of the function code.

Example:
```
int (*ptr) ( ), func1( );
ptr = func1;
```

Since the function name always refers to the base address of the code, func1 refers to the base address from where the code is written and can be easily assigned to another pointer ptr. So, ptr is now pointing to the func1( ).

Calling Function through Pointer To call the function pointed to by a pointer variable, simply de-reference it like other pointers, and include within the parenthesis the arguments to be passed to the function.

For example,

```
int (*fp) (int, int), test (int, int);
fp = text;
(*fp) (10, 20);    /* same as test (10, 20) */
```

Advantages of a Function Pointer A function pointer can be used to define a common template that can be used by many other functions.

For example, **printf, fprintf,** and **sprintf** operate in the same way by calling a common print function. They vary just while sending output to different destinations, the standard output device, file and string respectively.

A function pointer can be passed to another function by merely passing the name of the function (without parenthesis).

FUNCTIONS WITH A VARIABLE NUMBER OF ARGUMENTS

It is often required to pass a variable-length argument list to a function. The parameter list of such a function contains one or more fixed parameters and is terminated by an ellipsis (,...) to represent the variable part. Such functions are declared as:

datatype func_name(type parfix1, type parfix2, ...);

For example,

double min (double oldmin, int *n*, ...);

In the file *stdarg.h* there are three macros available called *va_start, va_arg* and *va_list*. These macros provide a method to access the arguments of the function when a function takes a fixed number of arguments followed by a variable number of arguments.

The fixed number of arguments are accessed in the normal way, where as the optional arguments are accessed using the macros *va_start* and *va_arg*.

va_start is used to initialize a pointer to the beginning of the list of optional arguments.

va_arg is used to advance the pointer to the next argument.

The program given below calls a function *display ( )* which prints any number of arguments of any type.

```
#include <stdarg.h>
main( )
{
   printf ("\n");
   display (1, 2, 5, 6);
   printf ("\n");
   display (2, 4, 'A', 'a', 'b', 'c');
   printf ("\n");
   display (3, 3, 2.5, 299.3, -1.0);
}
display (type, num)
int type, num;
{
   int i, j;
   char c;
   float f;
   va_list ptr;
   va_start (ptr, num);
```

```
switch (type)
{
   case 1 :
         for (j = 1; j < = num; j++)
         {
            i = va_arg (ptr, int);
            printf ("%d", i);
         }
         break;
   case 2 :
         for (j = 1; j < = num; j++)
         {
            c = va_arg (ptr, char);
            printf ("%c", c);
         }
         break;
   case 3 :
         for (j = 1; j < = num; j++)
         {
            f = (float) va_arg (ptr, double);
            printf ("%f", f);
         }
   }
}
```

Here we pass two fixed arguments to the function *display( )*. The first one indicates the data type of the arguments to be printed. The second indicates the number of such arguments to be printed.

Once again through the statement *va_start (ptr, num)* we set up *ptr* such that it points to the first argument in the variable list of arguments. Then depending upon whether the value of the type is 1, 2 or 3, we print out the arguments as *int*s, *char*s, or *float*s.

Summary

- Pointer variable represents the address of a data item.
- Using the pointer, the memory location can directly be accessed.
- There are two operators for pointer related operations & (address of) and * (value at address).
- Pointer variables can be used in expressions.
- Addition of two pointers, multiplication, and division of the pointer with a number is not allowed.
- Array elements can be accessed through the pointers quickly.
- malloc( ) and calloc( ) functions are used to allocate memory dynamically.
- Pointers can be passed to the functions as an argument.
- Like any other return type, functions can also return the pointers.

Review Exercise

Multiple-Choice Questions

1. A pointer is a variable that holds
   (a) value of any variable
   (b) address of some other variable
   (c) its own address
   (d) none of these
2. The memory allocated by malloc( ) contains
   (a) garbage values
   (b) all zeroes
   (c) all ones
   (d) none of these
3. Which one of the following is not a valid operation on pointers?
   (a) Adding integer to a pointer
   (b) Addition of two pointers
   (c) Subtraction of one pointer from another
   (d) None of these
4. The base address of an array is stored
   (a) in the array itself
   (b) in the registers of CPU
   (c) in the name of the array itself
   (d) none of these
5. The increment operation when applied to a pointer
   (a) increases its value by 1
   (b) increases its value by the length of the data type of this pointer

(c) increases by 2 only
(d) none of these

State whether True or False

1. The address operator is a unary operator.
2. The indirection operator (*) can only act upon operands that are pointers.
3. Pointer variables can only point to numeric or character variables.
4. A pointer variable cannot be assigned a null (zero) value.
5. One pointer variable can be subtracted from another provided both variables point to elements of the same array.

Fill in the Blanks

1. Malloc( ) is used to allocate memory _____ .
2. & is also known as _____ operator.
3. A pointer to pointer holds the address of a _____ .
4. Pointers are used for _____ accessing the memory locations.
5. The address of any variable is always an _____ integer.

Descriptive Questions

1. What kind of information does a pointer variable represent?
2. How is a pointer variable declared? What is the purpose of the data type included in the declaration?
3. What is the relationship between an array name and a pointer? How is an array name interpreted when it appears as an argument to a function?
4. Why is it sometimes desirable to pass a pointer to a function as an argument?
5. Suppose an integer quantity is added to or subtracted from a pointer variable. How will the sum or difference be interpreted?
6. Write the output of the following programs:

(a)
```
main( )
{int x[ ] = {1, 2, 3, 4, 5};
int m, *ptr;
ptr = x;
*ptr + 2 = 10;
for (m = 0; m < 5; m++)
{
   printf ("%d-", x [m]);}
   printf ("\n");
}
```

```
(b) main( )
    {
       int x [4] = {4, 5, 9, 10};
       int *ptr = x;
       ptr + = 2;
       printf ("%d\n", * ptr);
    }
(c) #include <stdio.h>
    int * func( )
    {
       static int x = 0;
       x++;
       return &x;
    }
    int main( )
    {
       int *y = func( );
       printf ("%d", (y)++);
       func( ); printf ("%d \n", *y); return 0;
    }
(d) char *ptr;
    char str [ ] = "UPTEC COMPUTER";
    ptr = str;
    ptr + = 5;
    What string does ptr points to ?
(e) #include <stdio.h>
    int y;
    void main( )
    {
       int x, *px, **ppx;
       x = 10;
       y = 1000;
```

```
        px = &x;
        ppx = &px;
        f2 (px);
        printf ("%d", *px);
    }
    void f2 (int *p)
    {
        p = &y;
        printf ("%d", *p);
    }
    What will be the output ?
(f) void main( )
    {
        char a[ ] = "sunshine";
        char *p = "sunstroke";
        a = "sunstroke";
        p = "sunshine";
        printf ("\n %s", a, p);
    }
```

7. Write a program to exchange the position of strings stored in an array using array of pointers.
8. Write a program to create two arrays to store roll numbers and marks of students whose number would be known at run time.
9. Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.
10. Write a program using pointers to read in an array of integers and print its elements in reverse order.

CHAPTER

13

Structures

Key Features

- Structure Definition
- Bit Fields
- Giving Values to Members
- Structure Initialization
- Comparison of Structure Variables
- Arrays of Structures
- Arrays within Structures
- Structures within Structures
- Passing Structures to Functions
- Structure Pointers

The 'C' Language allows you to create custom data types. The structure is a custom data type which combines different data types to form a new user-defined data type. A structure gathers together different atoms of information that comprise of a given entity.

In this chapter, we are going to discuss in detail, various aspects of structures.

STRUCTURE DEFINITION

A structures is a collection of related variables, also called aggregates, under one name. The variables may be of different data types. This is in contrast to arrays, in which the elements should be of the same data types.

Structures are commonly used to define records which will be stored in files. Pointers and structures facilitate the formation of more complex data structures such as limited lists, queues, stacks and trees.

Structures are a derived data type as they are constructed using objects of other types. Consider the following example:

```
struct student
{
    int rollno;
    float marks;
};
```

The keyword *struct* introduces the structure definition. The identifier *student* is the *structure tag*. The structure tag names the structure definition and is used with the keyword *struct* to declare variables of the *structure type*. In the above example, struct student is the structure type. Variables which are defined within the structure definition are the structure's members. Members of the same structure must have

unique names, but two different structures may contain members with the same name without conflict. Each structure definition must end with a semicolon.

The structure tag can be used later in definitions of instances of the structure. The declaration struct student s, stu [4]

declares *s* to be a variable of type *struct student* and *stu* to be an array with four elements of type *struct student.*

The preceding declaration could also have been done as follows:

```
struct student
{
    int rollno;
    float marks;
} s, stu [4];
```

The structure tag name is optional. If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition and not in a seperate declaration.

Creating Structure Variables

The structure declaration does not actually create variables, instead, it defines a data type. For actual use, a structure variable needs to be created. This can be done in two ways:

- Declaration using a tag name anywhere in the program.
  Example:
  ```
  struct book
  {
     char name [30];
     char author [25];
     float price;
  }
  struct book book1, book2;
  ```
- It is allowed to combine the structure declaration and the variable declaration in one statement. This declaration is given below :
  ```
  struct person
  {
     char * name;
     int age;
     char *address;
  }
  p1, p2, p3;
  ```

While declaring structure variables along with their definitions, the use of a tag name is optional.

```
struct
{
   char *name;
   int age;
   char *address;
}
p1, p2, p3;
```

BIT FIELDS

'C' provides us the ability to specify the number of bits in which an unsigned or *int* member of a structure is stored - referred to as a **bit field.** By using bit fields, we can store data in the minimum number of bits thereby increasing memory utilization. **Bit fields** members must be declared as *int* or unsigned.

A **bit field** is declared by following an unsigned or *int* member name with a colon (:) and an integer constant representing the width of the field (i.e., the number of bits in which the member is stored). The constant representing the width must be an integer between 0 and the total number of bits used to store an *int* on your system. Consider the decleration:

```
struct student
{
    unsigned rollno: 2;
    unsigned makes: 4;
};
```

indicates that a member roll number occupies 2 bits and member marks occupy 4 bits.

GIVING VALUES TO MEMBERS

As the members are not themselves variables, they should be linked to the structure variables. The link between a member and a variable is established using member operator '.' which is also known as dot operator.

This can be explained using the following example:

```
/ * Program to define a structure and assign value to
members * /
struct book
{
   char * name;
```

```
    int pages;
    char *author;
};
main( )
{
    struct book b1;
    printf ("\n Enter Values :");
    scanf ("%s %d %s", b1.name, &b1.page, b1.author);
    printf ("%s, %d, %s, b1.name, b1.page, b1.author);
}
```

STRUCTURE INITIALIZATION

A structure variable can be initialized as any other data type.

```
main( )
{
    static struct
{
    int weight;
    float height;
}
student = {60, 180.75};
```

This assigns the value 60 to the student-weight and 180.75 to the student-height. There is a one-to-one correspondence between the members and their initializing values.

A structure must be declared as static if it is to be initialized inside a function (similar to arrays). The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main( )
{
    struct st_record
    {
        int weight;
        float height;
    }
          static struct st_record student1 = {60, 180.75};
```

```
        static struct st_record student2 = {53, 170.60};
        - - - - - -
        - - - - - -
    }
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record / * No static word */
{  int weight;
   int height;
}
student1 = {60, 180.75};
main( )
{  static struct st_record student2 = {53, 170.60}
   - - - - - -
   - - - - - -
}
```

The initialization of individual structure members within the template is permitted. The initialization must be done only in the declaration of the actual variables.

COMPARISON OF STRUCTURE VARIABLES

Two variables of the same structure type can be compared the same way as ordinary variables.

If person1 and person2 belong to the same structure, then the following operations are valid as listed in the Table 13.1.

Table 13.1 Operations on structure variables

Operation	*Meaning*
person 1 = person2	Assign person2 to person1.
person1 = = person2	Compare all members of person1 and person2 and return 1 if they are equal, 0 if otherwise

Example:

```
struct class
{
   int number;
   char name [20];
```

```
        float marks;
    }
    main( )
    {
        int x;
        static struct class student1 = {111, "Rao", 72.50};
        static struct class student2 = {222, "Reddy", 67.00};
        static class student 3;
        student 3 = student 2;
        x = ((student3.number = = student.number) &&
(student3.marks = = student2.marks)) ? 1:0;
        if (x = = 1)
        {
            printf ("\nStudent2 and Student3 are same \n");
            printf ("%d %s %f\n", student3.number, student3.name,
student3.marks);
        }
        else
        printf ("\nStudent2 and Student3 are different\n");
    }
```

Output : Student2 and Student3 are same.
222 Reddy 67.000000

ARRAYS OF STRUCTURES

The most common use of structures is in the arrays of the structure. To declare an array of structures, first the structure is defined, then an array variable of that structure is declared. In such a declaration, each element of the array represents a structure variable.

Example: struct class student [100];

It defines an array called *student* which consists of 100 elements of structure named *class.*
An array of structures is stored inside the memory in the same way as a multi-dimensional array.

Example Program:

```
    / * Program to implement an array of structure * /
    struct marks
```

```
            {
               int sub1;
               int sub2;
               int sub3;
               int total;
            };
            main( )
            {
               int i;
               static struct marks student [4] = {{45, 67, 81, 0}, {75,
53, 69, 0}, {75, 53, 69, 0}, {57, 36, 71, 0}};
               static struct marks total;
               for (i = 0; i < = 2; i++)
               {
                  student [i].total = student [i].sub1 + student
[i].sub2+student[i] sub3;
                  total.sub1 = total.sub1 + student [i].sub1;
                  total.sub2 + = student [i].sub2;
                  total.sub3 + = student [i].sub3;
                  total.total = total.total + student [i].total;
               }
               printf ("STUDENT \t\t TOTAL \n");
               for (i = 0; i < = 3; i++)
               printf ("Student [%d] \t \t %d \n", i+1, student
[i].total);
               printf ("\n SUBJECT\t\t TOTAL \n %s \t\t %d \n %s \t\t %d
\n %s\t\t %d", "subject1", total.sub1,
               "Subject2", total.sub2. "Subject3", total.sub3);
               printf ("\n GRAND TOTAL = \t\t %d \n", total.total.);
            }
```

ARRAY WITHIN STRUCTURES

Single or multi-dimensional arrays of type *int* or *float* can be defined as structure members.

Example:

```
struct marks
{
    int number;
    float subject [3];
}
student [2];
```

Here the member subject contains three elements, subject [0], subject [1], and subject [2]. These elements can be accessed using appropriate subscripts. For instance, the name student [1],subject [2];would refer to the marks obtained in the third subject by the second student.

Example Program:

```
/* Program to implement arrays within a structure. * /
main( )
{
   struct marks
   {
      int sub [3];
      int total;
   };
static struct marks student [3] = {45, 76, 81, 0, 75, 53,
69, 0, 57, 36, 71, 0};
static struct marks total;
int i, j;
for (i = 0; i < = 2; i++)
      {
         student[i].total + = student[i].sub[j]
         total.sub[j] + = student [i].sub [j];
      }
      total.total + = student [i].sub[j];
   }
   total.total + = student [i].total;
```

```
}
printf ("STUDENT\t\tTOTAL \n");
}
for (i = 0; i < = 2; i++)
printf ("Student [%d]\t\t%d\n", i + 1, student [i].total);
printf("SUBJECT\ttd\n")
for (j = 0; j < = 2; j++)
printf ("Subject %d\t\t   %d\n", j + 1, total.sub[j]);
printf("GRAND TOTAL = \t\t%d\n", total.total);
}
```

Output:

Student	**Total**
Stduent[1]	193
Student[2]	197
Student[3]	164
Subject	**Total**
Subject1	177
Subject2	156
Subject3	221
GRAND TOTAL	554

STRUCTURES WITHIN STRUCTURES

Structures within a structure means nesting of structures. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
    char name [20];
    char department [10];
    int basic_pay;
    int dearness_allowance;
    int house_rent_allowance;
    int city_allowance;
}
employee;
```

This structure defines name, department, basic pay, and three kinds of allowances. All the items related to allowance can be grouped together and declared under a sub-structure as shown below:

```
struct salary
{
    char name [20];
    char department [10];
        struct
        {
            int dearness;
            int house_rent;
            int city;
        }
        allowance;
    }
    employee;
```

The salary structure contains a member named *allowance* which itself is a structure with three members. The members contained in the inner structure namely dearness, house_rent, and city can be referred to as

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outermost to inner-most) with the member using the dot operator.

The following statements are invalid:

```
employee.allowance          (actual member is missing)
employee.house_rent         (inner structure variable is missing).
```

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
    struct
    {
        int dearness;
        - - - - -
    }
    allowance, arrears;
```

```
}
employee [100];
```

The inner structure has two variables, *allowance*, and *arrears*. This implies that both of them have the same structure template.

A base member can be accessed as follows:

```
employee[1].allowance.dearness
employee[1].arrears.dearness
```

Tag names can also be used to define inner structures.

Example:

```
struct pay
{
   int dearness;
   int house_rent;
   int city;
};
struct salary
{
   char name [20];
   char department[10];
   struct pay allowance;
   struct pay arrears;
};
struct salary employee[100];
```

The pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

It is also permissible to nest more than one type of structure

```
struct personal_record
{
    struct name_part name;
    struct date date_of_birth;
    ------
    ------
};
struct personal_record person1;
```

The first member of the structure is the name which is of the type *struct name_part.* Similarly other members have their structure types.

PASSING STRUCTURES TO FUNCTIONS

These are three methods by which the values of a structure can be transferred from one function to another.

- The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables.
- The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the entire structure to the called function, changes are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function.
- The third approach employs a concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it.

The general format of sending a copy of a structure to the called function is:

```
function_name (structure_variable_name)
```

The called function takes the following form :

```
data_type function_name (st_name)
struct_type st_name;
{
    - - - - - -
    - - - - - -
    return (expression);
}
```

Notes:
1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as struct with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type.
3. The return statement is necessary only when the function is returning some data. The expression may be any simple variable or structure or, an expression, using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in the calling function for its type, if it is placed after the calling function.

Example Program

```
            /* Program showing passing of structure member as function
parameters * /
            struct stores
            {
               char name [20];
               float price;
               int quantity;
            };
            main( )
            {
               struct stores update( );
               float mul( ), p_increment, value;
               int q_increment;
               static struct stores item = {"XYZ", 25.75, 12};
               printf ("\nInput Increment Values:");
               printf ("\nPrice Increment and Quantity Increment\n");
               scanf ("%f %d", &p_increment, &q_increment);
               item = update item, pincrement, qincrement);
               printf ("\nUpdated values of item");
               printf ("\nName   :   %s\n", item.name);
               printf ("\nPrice    : %f\n", item.price);
               printf ("\nQuantity : %d\n", item.quantity);
               value = mul (item);
               printf ("\nValue of the item : %d\n", value);
            }
            struct stores update (struct stores product, float p, int q)
            {
               product.price + = p;
               product.quantity + = q;
               return (product);
            }
```

```
float mul (struct stores stock)
{
   return (stock.price *stock.quantity);
}
```

Output:
Input Increment Values : Price Increment and Quantity Increment
10 12
Updated values of item
Name : *XYZ*
Price : 35.750000
Quantity : 24
Value of the item : 858.000000

In case of structures that have numerous structure elements, passing these individual elements would be tedious. In such cases, an entire structure can be passed to a function.

Example Program:

```
/* Program passing entire structure as function parameter. */
struct emp
   {
      char empname [25];
      char company [25];
      int empno;
   };
   main( )
   {
      static struct emp emp1 = {"Prashant", UPTEC", 101};
      display (emp1);
   }
   display (e)
   struct emp e;
   {
      printf ("%s\n%s\n%d", emp.empname, emp.company,
                  emp.empno);
   }
```

Output:
Prashant
UPTEC
101

STRUCTURE POINTERS

A complete structure can be transferred to a function by passing a structure-type pointer as an argument. In principle, this is similar to the procedure used to transfer an array to a function. However, we must use an explicit pointer notation to represent a structure that is passed as an agrument. A structure passed in this manner will be passed by reference rather than by value. Hence, if any of the structure members are altered within the function, the alterations will be recognized outside the function.

Example Program:

```
#include <stdio.h>
typedef struct
{
   char *name;
   int acct_no;
   char accttype;
   float balance;
}
record;
/* transfer a structure-type pointer to a function */
main( )
{
   void adjust (record *pt);    /* function declaration * /
   static record customer = {"Smith", 3333, 'C', 33.33};
   printf ("%s %d %c %.2f\n", customer.name,
customer.acct_no, customer.acct_type, customer.balance);
   adjust (&customer);
   printf ("%s %d %c %.2f\n", customer.name, customer.acct_no,
      customer.acct_type, customer.balance);
}
void adjust (record *pt)
{
   pt->name = "Jones";
   pt->acct_not = 9999;
   pt->acct_type = 'R';
   pt->balance = 99.99;
   return;
}
```

- This program illustrates the transfer of a structure to a function by passing the structure's address (a pointer) to the function. In particular, customer is a static structure of type record, whose members are assigned an initial set of values. These initial values are displayed when the program begins to execute. The structure's address is then passed to the function *adjust* where different values are assigned to the member of the structure.
- Within *adjust*, the formal argument declaration defines *pt* as a pointer to a structure of type record. Also, nothing is explicitly returned from *adjust* to *main*.
- Within main, the current values assigned to the members of customer are again displayed after *adjust* has been accessed. Thus, the program illustrates whether or not the changes made in *adjust* carry over to the calling portion of the program.
- Executing the program results in the following output:
  Smith 3333 C 33.33
  Jones 9999 R 99.99
- The value assigned to the members of customer within *adjust* are recognized within *main*.
- A pointer to a structure can be returned from a function to the calling portion of the program. This feature may be useful when several structures are passed to the function, but only one structure is returned.

As we define a pointer pointing to **int**, or a pointer pointing to a **char**, similarly we can have a pointer pointing to a **struct**. Such pointers are known as **structure pointers**. The program given below demonstrates the usage of **structure pointer.**

```
main( )
{
    struct emp
    {
        char empname [25];
        char company [25];
        int empno;
    };
    static struct emp emp1 = {"Prashant", "UPTEC', 101};
    struct emp *ptr;
    ptr = &emp1;
    printf ("%s %s %d\n", emp1.empname,emp1.company,emp1.enpno);
    printf ("%s %s %d\n", ptr->company, ptr->empno);
}
```

In the above program, two types of operators are used to the refer to structure elements :

(i) Dot Operator

(ii) Arrow Operator

When the structure is referred to, by its name, the structure elements are addressed using dot operators.

Example: b1.name

When the structure is referred to by the pointer to the structure, the structure elements are addressed using arrow operators.

Example: ptr->name

> **Note:** On the left hand side of '.' structure operator, there must always be a structure variable, whereas on the left handside of the '->' operator there must always be a pointer to a structure.

The following program demonstrates the passing of the address of a structure variable to a function.

```
struct emp
{
   char empname [25];
   int empno;
}
main( )
{
   static struct emp emp1 = {Prashant",uptec", 101};
   display (&emp1);
}
display (e)
struct emp *e; /*pointer to a structure */
{
   printf ("%s \n%s\n%d", e->empname, e->empno);
}
```

Output :
Prashant
UPTEC
101

In the above example, -> operator is used to access the structure elements using the pointer to the structure.

Summary

- Structure is a derived data type used to store the instances of variables of different data types.
- Structure definition creates a format that may be used to declare structure variables in the program later on.
- The structure operators like dot operator "." are used to assign values to structure members.

- Structure variable can be initialized as any other data type.
- An array of the structure can be declared as any other array. In such an array, each element is a structure.
- Structures may contain arrays as well as structures.
- Structures can be passed to the functions in three ways:
  1. Pass each member individually.
  2. Pass a copy of the entire structure.
  3. Use a pointer to the structure.

Review Exercise

Multiple-Choice Questions

1. Structure is a
   (a) primitive data type
   (b) desired data type
   (c) basic data type
   (d) none of these
2. For defining a structure which keyword is used?
   (a) stru
   (b) structure
   (c) struct
   (d) none of these
3. A structure variable accesses its members by using
   (a) comma operator
   (b) dot operator
   (c) asterisk
   (d) none of these
4. In how many ways can a structure be passed to a function?
   (a) 4
   (b) 3
   (c) 2
   (d) none of these
5. An array of structures contains
   (a) each element is a structure
   (b) only one element is a structure
   (c) first and last elements are structures
   (d) none of these

State whether True or False

1. We can combine the declaration of the structure type and the structure variables in one statement.
2. The elements of a structure are always stored in non-contiguous memory locations.
3. The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.
4. One structure cannot be nested within another structure.
5. A structure variable can be initialized only if its storage class is either external or static.

Fill in the Blanks

1. A structure is a _____ of dissimilar data types.
2. During a structure declaration _____ memory space is reserved.
3. The total size of a structure is the _____of memory requirements of its members.
4. A structure pointer uses _____ operator to refer to its members.
5. Structures within a structure are called _____ structures.

Descriptive Questions

1. What is a structure? How does a structure differ from an array?
2. How can structure variables be declared? How do structure variable declarations differ from structure type declarations?
3. How are the members of a structure variable assigned initial values?
4. How is a structure member accessed? How can a structure member be processed?
5. Can the period operator (.) be used with an array of structures? Explain.
6. What will be the output of the following programs:

(a)
```
main( )
{
   struct
   {
      int i;
   }
   xyz;
   (&xyz) -> i = 10;
   printf ("%d", xyz.i);
}
```

(b)
```
main( )
{
   struct
   {
```

```
            int i;
         }
         *xyz;
         (&*xyz)->i = 10;
         printf ("%d", xyz ->i);
      }
(c) main( )
      {
         struct xyz
         {
            int i;
         }
         struct xyz *p;
         struct xyz a;
         p = &a;
         p -> i = 10;
         printf ("%d", xyz.i);
      }
(d) main( )
      {
         struct xyz
         {
            int xyz;
         };
         struct xyz xyz;
         xyz.xyz = 10;
         printf ("%d", xyz.xyz);   }
(e) Is there any error (Yes/No)?
      main( )
      {
         struct xyz
         {
```

```
        int i;
      }
      *pqr;
  }
(f) main( )
  {
      struct xxx
      {
         int i;
         char j;
      };
      struct xxx zzz = {I, 'a'];
      abc (zzz);
  }
  abc (struct xxx aaa)
  {
      printf ("%d. . . %d", aaa.i, aaa.j);
  }
```

7. Write a program to accept the names of students, date of birth, date of enrollment and print them.
8. Write a program to search a telephone number in a record of five telephone subscribers and print the name, telephone number, bill number, and amount of the searched record.

CHAPTER

14

Unions

Key Features

- Union—Definiton and Declaration
- Accessing a Union Member
- Union of Structures
- Initialization of a Union Variable
- Uses of Union
- Use of User-Defined Type Declarations

Unions are derived data types, the way structures are. Though unions and structures look alike, there is a fundamental difference. While structures enable you to create a number of different variables stored in different places in the memory, unions enable you to treat the same space as a number of different variables.

In this chapter, we are going to discuss the unions in detail.

UNION—DEFINITION AND DECLARATION

Unions follow the same syntax as structures. Unions and structures differ in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time.

Like structures, a union can be declared using the keyword union as follows:

```
union item
{
    int m;
    float x;
    char c;
} code;
```

This declaration declares a variable code of type *union item.* The union contains three members, each with a different data type. However, only one can be used at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

The following Figure 14.1 describes the memory arrangement of the union discussed above.

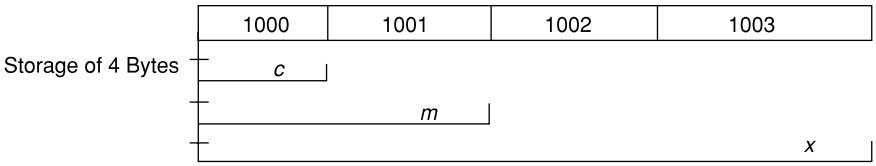


Fig. 14.1 *Memory storage of a union*

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. As shown in the example declaration, the member *x* requires 4 bytes which is the largest among the members. It is assumed that a *float* variable requires 4 bytes of storage and the figure above shows how all the three variables share the same address.

ACCESSING A UNION MEMBER

To access a union member, we can use the same syntax that we use for structure members.

For example: *code.m, code.x, code.c* are all valid member variables.

During accessing, we should make sure that we are accessing the member whose value is currently stored.

For example, the statements such as

```
code.m = 150;
code.x = 785;
printf ("%d", code.m);
```

would produce an erroneous output (which is machine dependent). The user must keep track of what type of information is stored at any given time.

Thus a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

UNION OF STRUCTURES

Just as one structure can be nested within another, a union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union. Here is an example of structures nested in a union:

```
main( )
{
   struct a
   {
      int i;
```

```
        char c[2];
    };
    struct b
    {
        int j;
        char d[2];
    };
    union z
    {
        struct a key;
        struct b data;
    } strange;
    strange.key.i = 512;
    strange.data.d[0] = 0;
    strange.data.d[1] = 32;
    printf("%d\n", strange.key.i);
    printf("%d\n", strange.data.j);
    printf("%d\n", strange.key.c[0]);
    printf("%d\n", strange.data.d[0]);
    printf("%d\n", strange.key.c[1]);
    printf("%d\n", strange.data.d[1]);
}
```

Output:

```
512
512
0
0
32
32
```

Structures and unions may be freely mixed with arrays.

Example:

```
union id
{
    char color[12];
    int size;
```

```
};
struct clothes
{
   char manufacturer[20];
   float cost;
   union id description;
} shirt, trouser;
```

Now *shirt* and *trouser* are structure variables of type *clothes*. Each variable will contain the following members: a string (*manufacturer*), a floating-point quantity (*cost*), and a union (*description*). The union may represent either a string (*colour*), or an integer quantity (*size*). Another way to declare the structure variable *shirt* and *trouser* is to combine the above two declarations as follows:

```
struct clothes
{
    char manufacturer[20];
    float cost;
    union
    {
        char color[12];
        int size;
     } description;
} shirt, trouser;
```

This declaration is more concise, though perhaps less straight-forward than the original declarations.

An individual union member can be accessed in the same manner as an individual structure member, using the operators " . " and " -> ". Thus, if the *variable* is a union variable, then *variable.member* refers to a member of the union. Similarly, if *ptvar* is a pointer variable that points to a union, then *ptvar->member* refers to a member of that union.

Example Program:

```
#include <stdio.h>
main( )
{
   union id
   {
      char color;
      int size;
```

```
            };
            struct
            {
               char manufacturer[20];
               float cost;
               union id description;
            } shirt, trouser;
            printf("%d\n", sizeof(union id));
            shirt.description.color = ' w ' ;
                                 /* assigns a value to color */
            printf("%c  %d\n",  shirt.description.color,
shirt.description.size);
            shirt.description.size = 12;
                                 /* assigns a value to size   */
            printf("%c  %d\n",  shirt.description.color,
shirt.description.size);
         }
```

INITIALIZATION OF A UNION VARIABLE

A union variable can be initialized, provided its storage class is either *external* or *static*. Only one member of a union can be assigned a value at any one time. The initialization value is assigned to the first member within the union.

Example Program:

```
/* Program to demonstrate initialization of union variables. */
#include <stdio.h>
main()
{
   union id
   {
      char color[12];
      int size;
   };
   struct clothes
   {
```

```
            char manufacturer[20];
            float cost;
            union id description;
        };
        static struct clothes shirt = {"American", "25.00", "White"};
        printf("%d\n", sizeof(union id));
        printf("%s %5.2f", shirt.manufacturer, shirt.cost);
        printf("%s    %d\n",    shirt.description.color,
shirt.description.size);
        shirt.description.size = 12;
        printf("%s %5.2f", shirt.manufacturer, shirt.cost);
        printf("%s    %d\n",    shirt.description.color,
shirt.description.size);
    }
```

Output:

```
12
American    25.00    White    26743
American    25.00    ~        12.
```

USES OF UNION

Unions, like structures, contain members whose individual data types may differ from one another. However, the members that compose a union share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus unions are used to *conserve memory.*

They are useful for applications involving multiple members, where values need not be assigned to all the members at any one time. Unions are also used wherever the requirement is to access the same memory locations in more than one way. This is often required while calling *Basic Input/Output System functions* (often simply called *BIOS routines*) present in the *read only memory* (ROM) of the computer.

Many DOS-based application softwares need to access DOS's internal data structures.

The break-up of these internal data structures, however, is not consistent and often changes from one version of DOS to another. Therefore, to make the application programs compatible with different versions of DOS, these programs create *unions* which take into account the variations in the break-up of these DOS data structures. These programs when executed, first test the version member of DOS being used on the machine and then access the appropriate part of the union.

USE OF USER-DEFINED TYPE DECLARATIONS

'C' supports a feature known as *type definition* that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables.

typedef

It takes the general form:

```
typedef type identifier;
```

where *type* refers to an existing data type and *identifier* refers to the new name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. The new type is 'new' only in name, but not the data type. *typedef* cannot create a new type.

Some examples of the type definitions are:

```
typedef int units;              /* units symbolizes int */
typedef float marks;            /* marks symbolizes float */
```

units and *marks* can be later used to declare variables as follows :

```
units batch1, batch2; /* batch1 and batch2 are declared as int variables .*/
marks name1 [50], name2 [50]; /* name1 [50] and name2 [50] are declared as
                                 50 element floating point array variables. */
```

The main advantage of *typedef* is that we can create meaningful data type names for increasing the readability of the program.

Example:

```
struct employee
{
    char name[30];
    int age;
    float bs;
};
struct employee e;
```

This structure declaration can be made more handy after renaming using *typedef* as shown below:

```
struct employee
{
    char name[30];
    int age;
    float bs;
};
typedef struct employee EMP;
EMP e1, e2;
```

```
e1.age = 40;
printf("%d", e1.age);
```

In this example, by using *typedef*, a long data type name is replaced by a short and suggestive data type name. Thus, by reducing the length and apparent complexity of data types, *typedef* can help to clarify source listing and save time and energy spent in understanding a program.

enum (Enumerated Data Type)

It is defined as *enum identifier {value 1, value 2, ... value n};*

The *identifier* is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants).

After this definition, we can declare the variable to be of this 'new' type as below :

enum identifier *v*1, *v*2, ...*vn*;

The enumerated variables *v1, v2, --- vn* can only have one of the values *value1, value2, ----valuen.*

The assignments *v1 = value3; v5 = value1;* are valid.

Example:

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

Note: The values, that are in original declaration, can only be used.

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant *value*1, is assigned 0, *value*2 is assigned 1, and so on.

The automatic assignments can be overridden by assigning values explicitly to the enumeration constants.

Example: *enum day {Monday =*1, *Tuesday, ---, Sunday};*

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

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

Example: *enum day{Monday,—Sunday}week_st, week_end;*

Like structures, this declaration has two parts:

(a) The first part declares the data type and specifies its possible values. These values are called *enumerators.*

(b) The second part declares variables of this data type.

Summary

- Union is a memory location that is shared by two or more variables.
- When a union variable is declared, the compiler automatically allocates enough storage to hold the largest member of the union.
- Only the unions with storage class external or static can be initialized.
- Unions are useful for applications involving multiple members. They are also used in many DOS-based application softwares.
- *typedef* and *enum* are two user-defined data types.

Review Exercise

Multiple-Choice Questions

1. Which one of the following is a user defined data type?
   (a) typedef
   (b) enum
   (c) Both (a) and (b)
   (d) none of these
2. In the union, the memory location is shared by
   (a) the first member of the union
   (b) any two members of the union
   (c) all the members of the union
   (d) none of these
3. The default value for the first enumeration constant is
   (a) 2
   (b) 4
   (c) 0
   (d) 1
4. How many members of a union can be initialized at a time?
   (a) 2
   (b) 4
   (c) 0
   (d) 1
5. A union may contain
   (a) another union
   (b) another structure
   (c) only unions
   (d) both (a) and (b)

State whether True or False

1. A union may be a member of a structure, and a structure may be a member of a union.
2. The elements of a structure use different memory locations whereas that of a union use same memory locations.
3. Only one member of a union can be assigned a value at any one time.
4. A union variable can be initialized, provided its storage class is either external or static.
5. The total size occupied by a union is equal to the size of its smallest member.

Fill in the Blanks

1. _____ datatypes can be created by enum.
2. Unions are used in _____ application softwares.
3. Unions having storage class external or static can be _____ .
4. Only _____ member of a union can be assigned a value at one time.
5. Unions are _____ from structures in terms of memory requirement.

Descriptive Questions

1. What is a union? How does a union differ from a structure?
2. For what kinds of applications are unions useful?
3. How is a union member accessed?
4. How is a member of a union variable assigned an initial value?
5. What will be the output of the following programs:

```
(a) main( )
    {
       union a
       {
             int i;
          char ch[2];
       };
       union a z1 = {512};
       union a z2 = {0, 2};
    }
(b) main( )
    {
       union
       {
          int i;
          char j;
```

```
        }
        xyz;
        xyz. i = 300;
        printf("%d", xyz.j);
    }
(c) main( )
    {
        union
        {
            union
            {
                char a;
                char b;
                char c;
                char d;
            }
            car;
        union
        {
            char i;
            char j;
        }
        int;
        char z;
    }
    pqr;     printf ("%d", sizeof (pqr));}
```

CHAPTER

15

Linked List

Key Features

- Data Structure
- Dynamic Memory Allocation
- Linked List
- Basic List Operations
- Doubly Connected Linked List
- Circular Linked List
- Self Referential Structure

In programming, there are many situations in which the data is dynamic in nature, that is, the number of data items are not fixed. They keep changing during the execution of the program. Arrays are not capable of handling such situations. These situations can be handled more effectively using dynamic data structures like linked list. Dynamic data structures provide flexibility in adding, deleting, or rearranging data items at run time.

In this chapter we will discuss linked list and various operations performed on the linked lists.

DATA STRUCTURE

Data structure is the branch of computer science that unleashes the knowledge of how the data should be organized, how the flow of data should be controlled and how a data structure should be designed and implemented to reduce the complexity and increase the efficiency of the algorithm.

The term data structure refers to a set of computer variables that are connected in some logical or mathematical manner. More precisely, a data structure can be defined as the structural relationship present within the data set and thus should be viewed as 2 tuple. [*N*, *R*] where '*N*' is the finite set of nodes representing the data structure and '*R*' is the set of relationship among those nodes.

For example, in a tree data structure, each node is related to each other in a 'parent-child' relation-

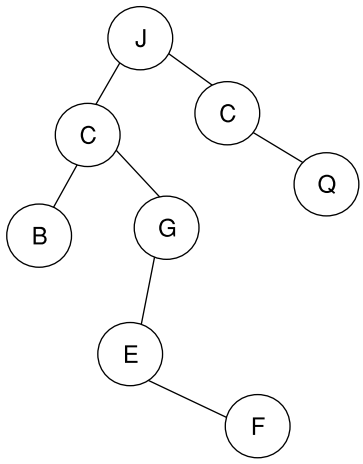


Fig. 15.1 *A sample tree structure*

ship. Thus, a large volume of data can be represented using a tree data structure and relationship between each data can also be shown as in Figure 15.1.

Linear Data Structure and Non-Linear Data Structure

In a linear data structure, member elements form a sequence. Such linear structures can be represented in the memory by using one of the two strategies.

- By having the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called *arrays*.
- By having a relationship between the elements represented by pointers, these structures are called linked lists.

There are various non-linear structures, such as, trees and graphs. In these data structures, insertion or deletion is not possible in a linear fashion.

- In a tree, the data frequently contains a hierarchical relationship between the various elements.
- A graph is a data structure that sometimes contains a relationship between pairs of elements, which is not necessarily hierarchical in nature.

Basic Operations on Data Structure

Various operations can be performed on data structures such as:

- **Traversal** : One of the most important operations which involves processing each element in the list.
- **Searching** : Searching or finding any element with a given value or record with a given key.
- **Insertion** : Adding a new element to the list
- **Deletion** : Removing an element from the list
- **Sorting** : Arranging the elements in some order
- **Merging** : Combining two lists into a single list.

DYNAMIC MEMORY ALLOCATION

'C' Language requires the number of elements in an array to be specified at compile time but with arrays it is not possible. In arrays, we allocate the memory first and then start using it. This may result in failure of the program or wastage of memory space.

Table 15.1 Dynamic memory management functions

Function	*Task*
malloc	allocates memory and return a pointer to the first byte of allocated space. Example: ptr = (cast.type*) malloc (byte_size);
calloc	allocate the memory spaces, initialize them to zero and returns pointer to first byte. Example: ptr = (cast_type*) calloc (n.elem_size):
free	frees previously allocated space. example: free (ptr);
realloc	modifies the size of previously assigned space. example: ptr = realloc (ptr, newsize);

The concept of dynamic memory location can be used to eradicate this problem. In this technique, the allocation of memory is done at run time. 'C' language provides four library functions known as memory management functions that can be used to allocate and free memory during program execution. These functions help us to build complex application programs that use the available memory intelligently.

LINKED LIST

Linked lists are self-referential structures. The basic idea of the linked list is that each component within the structure includes a pointer indicating where the next component can be found. The relative order of the components can easily be changed by altering the pointers.

A linked list does not need contiguous memory allocation and they can be dynamically allocated. A sample linked list is shown in Figure 15.2.

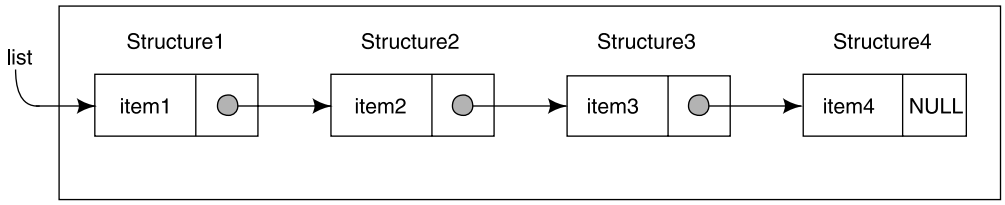


Fig. 15.2 *Linked list*

Each structure of the linked list is called a node and consists of two fields, one containing the item and the other containing the address of the next item in the list. Such a structure is given below.

```
struct node
{
    int num;
    struct node *next;
};
```

Advantanges of Linked Lists

- They can grow or shrink in size during the execution of a program.
- They do not waste memory space.
- They provide flexibility in allowing the items to be rearranged efficiently.

BASIC LIST OPERATIONS

We can perform the following basic operations on the linked lists:

- Creating of List
- Traversing the List
- Printing the List

- Deleting a node
- Concatenating two lists

CREATION OF LINKED LIST

Consider the following structure:

```
struct node
{
    int item;
    struct node * next;
};
typedef struct node link;
```

The *struct* declaration only describes the format of the node and does not allocate storage. For storage, we have to use *malloc* function and contain the address of the node in a pointer variable of type struct node. Initially this variable will contain null value.

If the pointer contains null value it indicates that the link has no node, and that we have to first create the node and assign null value to the next, otherwise we will traverse the entire list until the last node is reached. On obtaining null value in the next portion, create a new node using *malloc* function and then assign values to the next node. 'C' implementation of *create* function is given below. In this function we will use a previously defined structure.

```
void create (link ** ptr, int data)
{
   link *temp;
   temp = * ptr;
   if (temp = = NULL)        /*creation of first Node */
   {
      * ptr = malloc (sizeof (link));
      * ptr --> item = data;
      * ptr --> next = NULL;
   }
   else
   {
      while (temp --> next ! = NULL)  /*traversal of Linked
List * /
      temp = temp --> next;
```

```
            (temp --> next) = (Link *) malloc (sizeof (link));
                                        /* insert in the end */
            (temp --> next) --> item = data;
            (temp --> next) next = NULL;
         }
      }
```

Note: In this function we have used the double pointer because we have to make a change in the memory.

```
      /* creating and inserting in an ordered manner */
      link *create ( ) /*function for creating a node and invoking
insert */
      {
         int num;
         link *q, *p1;
         p1 = (link *) malloc (sizeof (link));
         printf ("\n enter the number, terminating condition is
0\n");
         scanf ("%d", &num);
         p1 → item = num ;
         p1 → next = null ;
         p1 = insert (p1);
         return (p1);
      }
      link * insert (link * p1)
      {
         link *q, *r;
         int num;
         printf ("enter the number \n");
         scanf ("%d",&num);
         while (num!=0)
         {
            q = p1;
```

```
                if (q → item > num)
                {
                   r = (link*) malloc (sizeof (link));
                   r → item = num ;
                   r → next = q ;
                   p1= r ;
                }
                else
                {
                   while (( q → next → item < = num) &&( q → next!=
                   null))
                   {
                      q = q → next ;
                   }
                   r = (link *) malloc (sizeof (link));
                   r → item = num ;
                   r → next = q next ;
                   q → next = r ;
                }
                printf ("\n enter the next no. >> (terminating condition
is 0)\n");
                scanf ("%d", &num);
             }
```

Printing of the linked list is very simple. Just traverse the list node by node and display the content of the node on the screen until the last node is reached. 'C' implementation of the print function is given below:

```
void print (link * ptr)
{
    link * temp;
    temp = ptr;
    while (temp ! = NULL)
    {
```

```
        printf ("%d->", temp --> item);
        temp = temp --> next;
    }
    printf ("NULL");
}
```

Deleting a Node

Consider the following linked list:

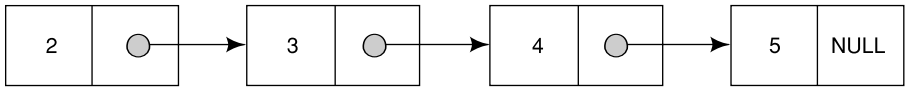


Fig. 15.3 *Linked list*

It is required to remove a node from this list.

In case of deletion, these cases may arise.

A. node to be removed is the first node of the list.
B. node is any other node.

Case A: Node is the first node of the List While deleting the first node of the list, first of all store the address of the first node in a pointer variable, make the second node as the first node and finally free the first node. Necessary coding is:

```
q = temp;
temp = q --> next;
i = q --> item;
free (q);
return (i);
```

Case B: Deletion of any other node Traverse the list until the desired node is obtained. While traversing, always retain the address of the previous node. Then assign the address contained in the next portion of the desired node to the next of the previous node and delete the node.

'C' implementation of deletion is given below. In this function, we have searched the node on the basis of data and performed the deletion.

```
int delete (link ** ptr, int data)
{
   link *temp, * q;
   int i;
   temp = * ptr;
   if (temp --> item == data)
```

```
{
   q = temp;
   temp = q --> next;
   i = q --> item;
   *free (q);
   *top = temp;
   return (i);
}
else
{
   while (( temp --> item) ! = data)
   {
      q = temp;
      temp = temp --> next;
   }
   q --> next = temp --> next;
   i = temp --> item;
   free (temp);
}}
```

Example Program:

- Concatenating two previously created lists:

Note: In this program we have referred the create( ) and print( ) functions that have been discussed previously.

```
struct node
{
  int item;
  struct node * next;
};
typedef struct node link;
void main( )
{
```

```
    link *l1, *l2, temp;
    int data;
    char * ch = 'y';
    l1 = l2 = NULL;
    clrscr( );
    while (ch = = 'y')
    {
          printf ("\n Enter data :");
          scanf ("%d", &data);
          create (&l1, data);
          printf ("\n \n do you want to continue y/n");
          scanf ("%c", &ch);
    }
    ch = 'y';
    while (ch = = 'y')
    {
          printf ("n Enter data for second list :");
          scanf ("%d", &data);
          create (&l2, data);
          printf ("\n \n do you want to continue y/n");
          scanf ("%c", &ch);
    }
    temp = l1;
    while (temp --> next ! = NULL)
    temp = temp --> next;
    temp --> next = l2;
    print (l1);
}
```

- ```
  /* program to sort a link list */
  #include <stdio.h>
  #include <conio.h>
  struct node
  ```
  ```

```
{
  int item;
  struct node*next ;
};
typedef struct node link;
void main ( )
{
  link * list = NULL, *k;
  int num ;
  void esort (link **);
  clrscr ( );
  printf ("Enter the no, enter 0 for termination >") ;
  scanf ("%d", &num) ;
  list = (link*) malloc (sizeof (link));
  list → item = num;
  list → next = NULL;
  printf ("\n Enter next no, enter 0 for termination >");
  scanf ("%d", &num);
  while (num!=0)
  {
        k = (link*) malloc (sizeof (link*));
        k → item = num;
        k → next = list ;
        list = k;
        printf ("\n Enter next no, enter 0 for termination >");
        scanf ("%d", &num);
  }
  k = list ;
  printf ("\n \t\t The list is \n");
  while (k!= NULL)
  {
        printf ("\t%d\n", k → item);
```

```
        k=k → next ;
    }
    esort (&list) ;
    k = list ;
    printf ("\n\t\t The sorted list is : \n");
    while (k! = NULL)
    {
        printf ("\t%d\n", k → item);
        k = k → next ;
    }
}
void    esort (link ** list)
{
    link *min, *p, *q, *r, *j;
    int k = 0;
    p = min = * list;
    while (p → next ! = NULL)
    {
        if (p → next → item < min → item)
        {
            r = p ;
            min = p → next;
        }
        k = 1;
        p = p → next ;
    }
    if (k)
    {
        r → next = *list ;
        j = (*list) → next;
        (*list) → next = min → next ;
        min → next = j ;
```

```
            * list = min ;
      }
      q = * list;
      p = q → next;
      while (q!= NULL)
      {
            min = p;
            k = 0;
            while (p → next! = NULL)
            {
                  if (p → next → item < min → item)
                  {
                        r = p;
                        min = p → next ;
                        k = 1;
                  }
                  else
                  p = p → next ;
            }
            if (k)
            {
                  r → next = q → next;
                  q → next = min ;
                  j = min → next ;
                  min → next = r → next → next ;
                  r → next → next = j;
                  q = q → next ;
            }
            p = q → next;
      }
}
```

DOUBLY CONNECTED LINKED LIST

Linked lists that we have considered so far have only one pointer that points to the next node. These are thus called **singly linked lists**. Singly linked lists are restrictive for operations where a node is known and some operation needs to be performed on its previous node. To overcome this restriction we have **doubly linked lists** in which each node stores two pointers—one to its previous node and one to the successive node. The list can thus be traversed from two sides.

Such a list will have each node of the following structure:

```
typedef struct node
{
    int item ;
    struct node * next, * back;
} link ;
```

Pictorially, each node may be represented as in Figure 15.4 (a).

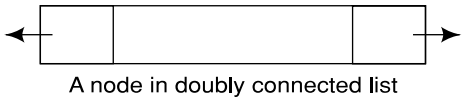


Fig. 15.4(a) *Node of doubly connected linked list*

The back link of the first node and the next link of the last node are set to null.

> **Note:** For a doubly circular linked list, the back of first node will point to last node and the next of last node will point to the first node.

All operations that can be performed on a singly linked list can be also performed on a doubly linked list.

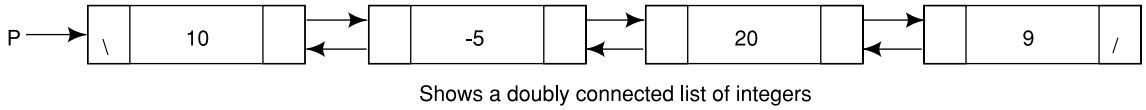


Fig. 15.4(b) *Node of doubly connected linked list*

CIRCULAR LINKED LIST

For a few applications, it is more convenient if the link field of the last node in a singly linked list points back to the first node of the list. Such a list is called a circular linked list. The structure of each node would be the same as for a singly linked list.

Pictorially a circular linked list integers can be displayed as shown in Figure 15.5.

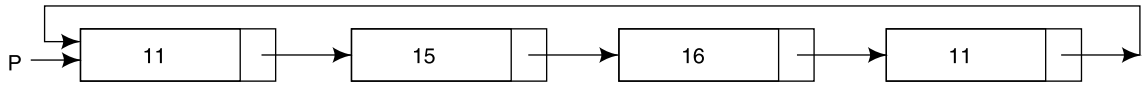


Fig. 15.5 *Circular linked list*

SELF-REFERENTIAL STRUCTURE

Sometimes it is desirable to include within a structure, one member that is a pointer to the parent structure type. In general terms, this can be expressed as:

```
struct tag
{
    member 1;
    member 2;
    . . .
    struct tag *ptr;
};
```

where *ptr* refers to the name of a pointer variable. Therefore structures of type *tag* will contain a member that points to another structure of type *tag*. Such type of structures are known as **self-referential structures**.

For example the following code

```
struct emp
{
    char name [40];
    struct emp *next;
};
```

is a structure of type *emp*. The structure contains two members: a 40-element character array, called *name*, and a pointer to another structure of the same type (i.e., a pointer to another structure of type *emp*) called *next*. Thus, this is a self-referential structure.

Self-referential structures are very useful in applications involving linked data structure, such as lists and trees.

Summary

- Arrays provide static memory allocation, while through linked lists we can achieve dynamic memory allocation.
- List structure contains nodes which have a data part and an address part which contains the address of the next node.
- Linked lists do not waste memory space.
- They provide flexibility in memory management.
- The operations performed on the list are creation, deletion, display, and insertion.

Review Exercise

Multiple-Choice Questions

1. Which one of the following is not a memory management function?
   (a) malloc( )
   (b) calloc( )
   (c) sqrt( )
   (d) none of these
2. One of the following operation cannot be performed on a single linked list. Identify.
   (a) Traversal
   (b) Insertion
   (c) Deletion
   (d) Concatenation
3. Linked lists are:
   (a) static data structure
   (b) dynamic data structure
   (c) not flexible for memory management
   (d) none of these
4. Doubly connected linked list contains _____ pointers in each node.
   (a) two
   (b) one
   (c) three
   (d) none of these
5. In a circular linked list the last node contains
   (a) null
   (b) address of its previous node
   (c) address of first node
   (d) none of these

State whether True or False

1. Self-referential structures are very useful in applications that involve linked data structures, such as lists and trees.
2. A linked list needs contiguous memory allocation.
3. Singly linked lists are restrictive for operations where a node is known and some operation needs to be performed on its previous node.
4. Linked lists provide flexibility in memory management.
5. A linked list uses self-referential structure.

Fill in the Blanks

1. Linked lists are based on the concept of _____ memory allocation.
2. _____ structures are used to create linked lists.
3. _____ linked list can be traversed in both directions.
4. *malloc( )* function is defined in the _____ header file.
5. As compared to an array, _____ operation is easier in a linked list.

Descriptive Questions

1. What is a self-referential structure? For what kinds of applications are self-referential structures useful?
2. What advantages are there in the use of linked data structures?
3. What is the basic idea behind a linked data structure?
4. Summarize several types of commonly used linked data structures.
5. Split one list into two based on the following criteria:
   (a) Evens in one list and odds in the other
   (b) Alternate elements in separate lists
   (c) Write a program to sort a linked list
   (d) Write a program to implement a linked list using recursion.
   (e) Write a program to traverse a linked list and display the contents in reverse order.
   (f) Write a program to insert a node at the beginning of a linked list.
   (g) Write a program to create, insert and display a doubly linked list of integers.
   (h) Write a program to create nodes, insert node, delete nodes, and traverse a circular linked list.

CHAPTER

16

File Handling in C

Key Features

- What is a File?
- Defining and Opening a File
- Closing a File
- Input /Output Operations on Files
- Functions for Random Access to Files
- Example Programs

In the 'C' programming studied so far, we have used memory to store data. But this storage was not permanent. For permanent storage it is required to use files to store data and read it from there. Like other languages, 'C' language supports a number of functions that have the ability to perform the basic file operations.

In this chapter, we are going to discuss various operations that can be performed on files.

WHAT IS A FILE?

Wherever there is a need to handle large volumes of data, it is advantageous to store data on disks and read whenever necessary. This method employs the concept of files to store data. A file is a place on the disk where a group of related data is stored. 'C' supports a number of functions that have the ability to perform basic file operations, which include:

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file
- Closing a file

There are two distinct ways to perform file operations in 'C':

- Low-level I/O Operation (It uses UNIX system calls)
- High-level I/O Operation (It uses functions in 'C's Standard I/O library)

The list of high level I/O functions is given below in Table 16.1.

Table 16.1 File I/O functions

Function Name	*Operation*
fopen( )	Creates a new file for use or opens an existing file for use.
fclose( )	Closes a file which has been opened for use.
getc( )	Reads a character from a file.
putc( )	Writes a character to a file.
fprint( )	Writes a set of data values to a file.
getw( )	Reads an integer from a file.
putw( )	Writes an integer to a file.
fgets( )	Reads the next input line from a file to a character array.
fputs( )	Writes a new line to the file.

Binary Mode versus Text Mode

High level I/O functions can be categorized as text and binary. This classification arises out of the mode in which the file is opened. The difference between text and binary mode file lies in their way of:

(a) **Handling of a new line—**In case of text mode, while writing to disk, a new line character is converted into the carriage return-linefeed combination. Similarly, while reading from the disk a linefeed combination is converted back into a new line. This conversion process does not take place if the file is opened in binary mode.

(b) **Representation of the end of a file**—In text mode, the end of a file is marked by a special character whose ASCII value is 26. There are no such special characters to mark the end file in binary mode. In binary mode, the end of a file is determined by keeping track of the number of characters present in the directory entry of the file.

(c) **Storage of Numbers**—If the file is open in text mode then the numbers are actually stored in string format. Therefore, even if 3291 occupies two bytes in the memory, when it is transferred to the disk using *printf( ),* it would occupy four bytes, one byte per character. If the amount of numeric data to be stored in a disk is large, then making use of text mode may turn out to be inefficient. Under such circumstances, it is better to open the file in binary mode and use the functions which store the numbers in binary format.

DEFINING AND OPENING A FILE

Before storing data in a file in the secondary memory, certain things about the file must be specified to the operating system. They include:

- Filename
- Data Structure
- Purpose

Filename is a string of characters that make up a valid filename for an operating system. It may contain two parts, a primary name and an optional period with an extension.

Example: Input.dat
Store
PROG.C
Student.C
Text.out

Data structure of a file is defined as FILE in the library of standard I/O function definitions. All files should be declared as of type FILE before they are used.

Purpose : When we open a file, we must specify what we want to do with the file.

Following is the general format for declaring and opening a file :

```
FILE        *fp;
fp = fopen ( "filename", "mode");
```

The first statement declares the variable *fp* as a "pointer to the data type FILE".

The second statement opens the file named *filename* and assigns an identifier to the FILE type pointer *fp*. This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job.

Mode can be one of the following :

r	Opens the file for reading only
w	Opens the file for writing only
a	Opens the file for appending (or adding) data to it.

Both the file name and mode are specified as string. They should be enclosed in double quotation marks.

Depending on the mode specified, one of the following actions may be performed.

1. When the mode is "writing", a file with the specified name is created, if the file does not exist. The contents are deleted if the file already exists.
2. When the purpose is "appending", the file is opened with the current contents safely. A file with the specified name is created if the file does not exist.
3. If the purpose is "reading", and if it exists, then the file is opened with the current contents safely, otherwise an error occurs.

Other additional modes of operation are:

r+	The existing file is opened from the beginning for both reading and writing.
w+	Same as *w* except both for reading and writing.
a+	Same as *a* except both for reading and writing.

Whenever a file is opened using *fopen* function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a null pointer.

This facility can be used to test whether a file has been opened or not.

Example:

```
if (fp == NULL)
printf ("File could not be opened.\n");
```

CLOSING A FILE

Once all the operations on a file have been completed, the file is closed. This is done to clear the buffers and flush all the information associated with the file. It also prevents any accidental misuse of the file. In case there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. When there is a need to use a file in a different mode, the file has to be first closed and then reopened in a different mode.

The I/O library supports a function for this of the following form:

```
fclose (file_pointer);
```

This would close the file associated with the FILE pointer file_pointer.

Example:

```
.....
FILE *p1, *p2;
p1 = fopen (“INPUT”, “w”);
p2 = fopen (“OUTPUT”, “r”);
    .....
fclose(p1);
fclose(p2);
```

This program opens two files and closes them after all operations on them are completed.

Once a file is closed, its file pointer can be reused for another file. All files are closed automatically whenever a program terminates. However, closing a file as soon as all operations related to it have been completed is a good programming habit.

INPUT/OUTPUT OPERATIONS ON FILES

getc and putc Functions

These are analogous to *getchar* and *putchar* functions and handle one character at a time.

putc can be used to write a character in a file opened in write mode.

A statement like putc (ch, fp1); writes the character contained in the character variable ch to the file associated with file pointer fp1.

Similarly, *getc* is used to read a character from a file that has been opened in read mode.

The statement c = getc(fp2); would read a character from the file whose file pointer is *fp*2.

The file pointer moves by one character position for every operation of *getc* or *putc*. The *getc* will return an end-of-file marker EOF, when end of the file has been reached. The reading should be terminated when EOF is encountered. Testing for the end-of-file condition is important. Any attempt to read past the end of the file might either cause the program to terminate with an error or result in an infinite loop situation.

getw and putw Functions

The *getw* and *putw* are integer-oriented functions. They are similar to the *getc* and *putc* functions and are used to read and write integer values on UNIX systems.

The general forms of getw and putw are :

putw (integer, fp); & getw (fp);

fprintf and fscanf Functions

The functions *fprintf* and *fscanf* perform I/O operations that are identical to the familiar *printf* and *scanf* functions.

The general syntax of fprintf is

fprintf (fp, "control string", list);

where *fp* is a file pointer associated with a file that has been opened for writing. The control string contains output specifications for items in the list. The list may include variables, constants and strings.

Example: fprintf (*f*1,"*%s %d %f*", name, age, 7.5); here name is an array variable of type *char* and age in an *int* variable.

The general syntax of *fscanf* is

fscanf (fp, "control string", list);

This statement would cause the reading of the items in the list from the file specified by *fp*, according to the specifications contained in the control string.

Example: fscanf (*f* 2, "*%s %d*", item, &quantity);

fscanf also returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF.

feof() Function

The *feof* function can be used to test for an end-of-file condition. It takes a FILE pointer as its only argument and returns a non zero integer value if all of the data from the specified file has been read, and returns zero otherwise. If *fp* is a pointer to the file that has just been opened for reading, then the statement

```
if (feof (fp))
    printf ("End of data.\n");
```

would display the message "End of data." on reaching the end of file condition.

Errors During Input/Output It is possible that an error may occur during input/output operations on a file. Typical error situations include:

- Trying to read beyond the end-of-file mark
- Device overflow
- Trying to use a file that has not been opened
- Trying to perform an operation on a file, when the file is opened for another type of operation
- Opening a file with an invalid filename
- Attempting to write to a write-protected file

The *ferror* function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a non zero integer if an error has been detected upto that point, during processing. It returns zero otherwise.

The statement

```
if (ferror (fp)! = 0)
printf ("An error has occurred. \n");
```

would print the error message, if the reading is not successful.

FUNCTIONS FOR RANDOM ACCESS TO FILES

To randomly access only a particular part of a file, the following functions are provided in 'C'.

- ftell
- rewind
- fseek

ftell( ) Function

ftell takes a file pointer and returns a number of type long that corresponds to the current position.

This function is useful in saving the current position of a file, which can be used later in the program. It is used as follows:

```
n = ftell (fp);
```

n would give the relative offset (in bytes) of the current position. This means that *n* bytes have already been read (or written).

rewind( ) Function

rewind takes a file pointer and resets the position to the start of the file.

For example, the statements

```
rewind (fp);
n = ftell (fp);
```

would assign 0 to *n* because the file position has been set to the start of the file by rewind.

This function helps us in reading a file more than once, without having to close and open the file.

> **Note:** The first byte in the file is numbered as 0, second as 1, and so on. Whenever a file is opened for reading or writing, a rewind is done implicitly.

fseek( ) Function

fseek function is used to move the file position to a desired location within the file. Its syntax is:

```
fseek (fileptr, offset, position).
```

- *fileptr* is a pointer to the file concerned.
- *offset* is a number or variable of type long. It specifies the number of positions (bytes) to be moved from the location specified by position.
- *position* is an integer number. It can take one of the following three values:

Value	Meaning
0	Beginning of file
1	Current position
2	End of file

The offset may be positive to move forward, or negative to move backwards.

The following examples illustrate the operation of the *fseek* function:

Statement	Meaning
fseek (fp, OL, 0);	Go to the beginning, (Similar to rewind)
fseek (fp, OL, 1);	Stay at the current position. (Rarely used)
fseek (fp, OL, 2);	Go to the end of the file, past the last character of the file.
fseek (fp, *x*, 0);	Move to $(x+1)^{th}$ byte in the file.
fseek (fp, *x*, 1);	Go forward by *x* bytes,
fseek (fp, -*x*, 1);	Go backwards by *x* bytes from the current position.

Unformatted data files

Some applications need to access records or blocks of data.

This can be implemented through the case of *fread* and *fwrite* functions.

These functions are often referred to as unformatted read and write functions.

fread (&buf, sizeof (buffer), number of records, file pointer) fwrite (&buf, sizeof (buffer), number of records, file pointer)

```
/* example to use fread() and fwrite() */
#include <stdio.h>
{
   unsigned int reg_no ;
   char name [50] ;
   unsigned int mask ;
} st_rec ;
void main (void)
{
```

```
        st_rec rec ;
        FILE *fp ;
        fp =fopen ("class.rec", "wb") ;
        printf ("Enter the registration number, name and marks \n");
        printf ("Type ctrl+z to stop \n");
        /* Read from Keyboard and write to file */
        while (scanf ("%u %s %u",&rec.reg_no, rec.name, &rec.mark)! = EOF)
        fwrite (&rec, size of (rec),1, fp);
        fclose (fp);
        printf ("\n");
        /* read from file and write on screen */
        fp = fopen ("class.rec", "rb");
        /* while loop terminates when fread returns 0 */
        while (fread (&rec, sizeof (rec), 1, fp))
        printf ("%5u %10s%3u", rec.reg_no, rec.name, rec.marks);
        fclose (fp);
    }
```

EXAMPLE PROGRAMS

- In this example, the text is read into the computer character-by-character using the *getchar* function and then written out to a data file character-by-character using *putc*.

```
        #include <ctype.h>
        #include <stdio.h>
        /* read in a line of lower-case text and store its upper-
case equivalent within a data file */
        main()
        {
           FILE *fpt;  /* define a pointer to pre-defined structure
type FILE */
           char c;
           /* open a new data file for writing only */
           fpt = fopen("sample.dat", "w");
           /* read each character and writes its upper-case equivalent
to the data file */
```

```
            do
            putc (toupper ( c = getchar() ), fpt);
               while (c != '\n');
            fclose (fpt);          /* close the data file */
         }
```

- A data file that has been created in this manner can be viewed in several different ways.
  Example: The data file can be viewed directly, using an operating system command such as *print* or *type*.
- Another approach is to write a program that will read the data file and display its contents. Such a program will, in a sense, be a mirror image of the one described above, that is, the library function *getc* will read the individual characters from the data file and *putchar* will display them on the screen.
- The following program will read a line of text from a data file character-by-character and display the text on the screen. The program makes use of the library functions *getc* and *putchar* to read and display the data. It complements the program presented in previous example.

```
         #include <stdio.h>
         #define NULL 0
         /* read a line of text from a data file and display it on
the screen */
         main()
         {
            FILE *fpt;  /* define a pointer to pre-defined structure
type FILE */
            char c;
            /* open the data file for reading only */
            if ((fpt = fopen("sample.dat", "r")) == NULL)
            printf("\nERROR - Cannot open the designated file\n");
            else    /* read and display each character from the
data file */
            do
            putchar(c = getc(fpt));
            while (c != '\n');
            /* close the data file */
            fclose(fpt);
         }
```

- The logic is directly analogous to that of the program as shown in the previous example, however, this program opens the data file sample.dat as a read-only file. An error message is generated if sample.dat cannot be opened.

  getc requires that the stream pointer fpt be specified as an argument.

- Data files consisting entirely of strings can often be created and read more easily with programs that utilize special string-oriented library functions. Some commonly used functions of this type are *gets*, *puts*, *fgets* and *fputs*. The functions *gets* and *puts* read or write strings to or from the standard output devices, whereas *fgets* and *fputs* exchange strings with data files.

Example:

/*This program uses more than one file and copies contents of one file into another. Both files are opened at a time, one in read mode and another in write mode */

```
main( )
{
   FILE  *fopen( ), /* holds pointer to the "source" file */
      *first_ptr, /* holds pointer to "destination" file */
      *sec_ptr;
   int c; /* receives one character of input*/
   /*opening files*/
      first_ptr = fopen ("firstfile", "r");
      sec_ptr = fopen ("secondfile", "w");
   /* copying contents */
   while ((c = getc (first_ptr)) ! = EOF)
   putc (c, sec_ptr);
   /* close pointers */
      fclose (first_ptr);
      fclose (sec_ptr);
   /* open second file for displaying */
      sec_ptr = fopen ("secondfile", "r");
      printf ("contents are:\n");
      while ((c = getc (sec_ptr))! = EOF)
      putchar(c);
      fclose (sec_ptr);
}
```

Example of Numeric Storing and Retrieving

In this example, we will create three files: file1, file2 and file3. File1 will have a certain numeric values while file2 will contain the odd numbers from file1 and file3 will contain the even.

```
# include <stdio.h>
void main ( )
{
   FILE*f1,*f2,*f3;
   int no, i;
   printf ("The numbers present in the file1 are:");
   f1 = fopen ("file1", "w");
   for (i=1, i<=30; i++)
   {
      scanf ("%d", & no);
      if (no = = -1) break;
      putw (number, f1);
   }
   fclose (f1);
   f1 = fopen ("file1", "r");
   f2 = fopen ("file2", "w");
   f3 = fopen ("file3", "w");
   while ((no = getw (f1)! = EOF)
   {
      If (no %2! = 0)
         putw (no, f2);
      else
         putw (no, f3)
   }
   fclose (f1);
   fclose (f2);
   fclose (f3);
   f2 = fopen ("file2", "r");
   f3 = fopen ("file3", "r");
```

```
   printf ("\n\n the constants of the file2 are \n");
   while (no = getw (f2)! = EOF)
      printf ("% 4d", no);
      printf ("\n Contents of the file3 are \n");
   while (no = getw (f3)! = EOF)
      printf ("%4d", no);
   fclose (f2);
   fclose (f3);
}
```

Examples on command line arguments

Here we will open one file and copy its contents to another file. The name of the two files will be taken as the input from the user.

```
# include <stdio.h>
void main (argc, argv)
int argc;
char *argv [ ];
{
   FILE *fs, *ft;
   char ch;
   if (argc! = 3)
   {
      printf ("\n In sufficient arguments");
      exit ( );
   }
   fs = fopen (argv [1], "r");
   if (fs == NULL)
   {
      puts ("Source file could not be opened");
      exit;
   }
   ft = fopen (argv [2], "w");
```

```
    if (ft == NULL)
    {
       puts ("cannot open target file");
       fclose (fs);
       exit ( );
    }
    while (1)
    {
       ch = getc (fs);
       if (ch ==EOF)
        break;
       else
        putc (ch, ft);
    }
    fclose (fs);
    fclose (ft);
}
```

Summary

- The file is a place on the disk where a group of related data is stored.
- Different functions are available in the library files which are used for file related operations.
- For reading and writing on files, *getc( ), putc( ), getw( ), puts( )* functions are used.
- To handle errors during file operations *ferror( )* function is used.
- For random access on files, the following functions are provided, namely, *ftell*, *rewind*, and *fseek*.

Review Exercise

Multiple-Choice Questions

1. The function used for closing a file is
   (a) *close*( )
   (b) *fclose*( )
   (c) *exit*( )
   (d) none of these

2. *w mode* opens the file for
   (a) only reading
   (b) only writing
   (c) both (a) and (b)
   (d) none of these
3. *putc* ( ) function
   (a) reads a character from a file
   (b) writes a character to a file
   (c) both (a) and (b)
   (d) none of these
4. *fscanf* ( ) functions reads _____ from a file.
   (a) single character
   (b) single field
   (c) multiple field
   (d) none of these
5. *open* function requires
   (a) file pointer
   (b) integer pointer
   (c) character pointer
   (d) none of these

State whether True or False

1. The FILE structure contains information about the file being used, such as its current size, its location in memory etc.
2. The EOF macro has been defined in the file "stdio.h".
3. *putc*( ) function always writes to the file, whereas *putch*( ) writes to the video.
4. The *fscanf* function permits formatted data to be read from a data file.
5. A file is a place in the main memory used for storing related data.

Fill in the Blanks

1. _____ mode is used for opening a file for reading only.
2. _____ function is used to handle errors during file operations.
3. *fseek* function is used to place _____ pointer at a desired location.
4. Rewind function is used to place the file pointer at the _____ of file.
5. Files can be accessed _____ .

Descriptive Questions

1. What is the primary advantage of using a data file?
2. What is meant by opening a data file? How is this accomplished?
3. What are the different file types that can be specified by the *fopen* function?

4. What is the purpose of the *fclose* function?
5. Contrast the use of the *fread* and *fwrite* functions with the use of the *fscanf* and *fprintf* functions.
6. Write a function to display statistics of a file.
7. Write a program to search a particular data in a file.
8. Write a program to copy one file to another using the command line arguments.
9. Write a program to sort data file that contains records. Each line consists one record and each field is separated by 'comma' delimiter.
10. Write a program to merge two sorted files F1 and F2 in F3 in a sorted order.

CHAPTER

17

C Preprocessor

Key Features

- Header File and Library File
- Introduction to Preprocessors
- Difference between Function & Macro
- Macro Substitution (#define)
- Undefinining a Macro (#undef)
- File Inclusion
- Conditional Compilation Directives (#if, #else, #elif, #endif, #ifdef, and #ifndef)

Preprocessor is a unique feature in 'C' language. The 'C' preprocessor is a collection of special statements, called *directives*, that are executed at the beginning of the compilation process. The #include and #define statements considered earlier in this book are preprocessor directives. Additional preprocessor directives are #if, #elif, #else, #endif, #ifdef, #ifndef, #line and #undef.

In this chapter we are going to discuss 'C' preprocessors in detail.

HEADER FILE AND LIBRARY FILE

Header files let the compiler know the prototype for a function in a library, plus other definitions. A *library* is just a collection of functions.

If you want to make your own statically linked library group, your function is a project, with no main and build it as a library. Note that the linker brings-in an entire code module at a time. This means if you put 20 functions in a single 'C' file and make it into a library, then call 1 function and you get all 20. If you make the library out of 20 'C' file of 1 function each, you only get the 1 file.

INTRODUCTION TO PREPROCESSORS

The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines* or *directives*.

Preprocessor directives are placed in the source program before calling the function *main( )*.

Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives follow special syntax rules that are different from the normal 'C' syntax. They all begin with the symbol # in column one and do not require a semicolon at the end.

A set of commonly used preprocessor directives and their functions are given below in Table 17.1.

Table 17.1 C preprocessor directives

Directive	*Function*
#define	Defines a macro substitution.
#undef	Undefines a macro.
#include	Specifies the files to be included.
#ifdef	Tests for a macro definition.
#ifndef	Tests whether a macro is not defined.
#if	Tests a compile-time condition.
#endif	Specifies the end of #if.
#else	Specifies alternatives when *#if* test fails.

These directives can be divided into three categories:

1. Macro Substitution Directives
2. File Inclusion Directives
3. Compiler Control Directives

DIFFERENCE BETWEEN FUNCTION AND MACRO

When a *macro* is called, its macro template is replaced with its macro expansion. On the other hand when a function is called, the control is passed to the function along with some arguments, some calculations are performed in the function and a useful value is returned back to the function.

Macros make the programs run faster but increase its size. Functions make the program smaller and compact.

A macro should be used only if it is small. If the macro is large then we should replace it with a function.

MACRO SUBSTITUTION (#define)

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of *#define* statement.

#define: This statement, usually known as a macro definition (or simply a macro) takes the following general form:

```
#define identifier string
```

- If this statement is included in the program at the beginning, the preprocessor replaces every occurrence of the *identifier* in the source code by the string.
- The string may be any text, while the *identifier* must be a valid 'C' name.

Simple Macro Substitution

Simple string replacement is commonly used to define constants.

Example:
```
#define    TRUE      1
#define    CAPITAL   "DELHI"
```

- A macro inside a string does not get replaced.

Example:
```
#define M 5
```

will replace all occurrences of *M* with 5, starting from the line of definition to the end of the program. But in the following case,

```
total = M * value;
printf ("M = %d\n", M);
```

these two lines will change during preprocessing as follows:

```
total = 5 * value;
printf ("M = %d\n", 5);
```

Here the string "M = *%d**n*" is left unchanged.

A macro definition can include more than a simple constant value. It can include expressions as well.

Example:
```
#define    AREA      5 * 12.46
#define    SIZE      sizeof (int) * 4
```

- Whenever expressions are to be used for replacement, care should be taken to prevent an unexpected order of evaluation. Correct results can be obtained by using parenthesis around the strings as shown below:

```
#define D (45-22)
```

The preprocessor performs a literal text substitution whenever the defined name occurs. This is the reason why we prefer not to use a semicolon to terminate the *#define* statement. This also suggests that we can use a macro to define almost anything.

Example:
```
#define TEST if (x>y)
#define AND
#define PRINT printf ("Very Good.\n");
```

to build a statement as : *TEST AND PRINT*

The preprocessor would translate this line to *if(x>y)printf("*Very Good.*n");*

Some tokens of 'C' syntax are confusing or are error-prone. Following are a few definitions that might be useful in building error-free and more readable programs.

```
#define    EQUALS      = =
#define    AND         &&
#define    INCREMENT   ++
```

```
#define     START       main( ) {
#define     END         }
```

Example: use of syntactic replacement

```
START
.....
if(total EQUALS 240 AND average EQUALS 60)
    INCREMENT count;
.....
END
```

Macros with Arguments

#define identifier(*f*1, *f*2, ..., *fn*) string

The identifiers *f*1, *f*2, ... *fn* are formal macro arguments that are analogous to the formal arguments in a function definition.

Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call).

When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters.

Example: *#define CUBE(x) (x*x*x)*

If the statement *volume = CUBE (side);* appears later in the program
then the preprocessor would expand this statement to *volume = (side * side * side);*

> **Note:** A parenthesis should be used for each occurrence of a formal argument in the string. Also the whole string should be enclosed within the parenthesis.

Actual parameters are substituted for parameters in a macro call, even if they are within a string.

Example: #define PRINT(variable, format) printf("variable= *%f* \n", variable)

can be called-in by

PRINT(price × quantity, *f*);

The preprocessor will expand this as

printf("price × quantity = *%f* \n", price × quantity);

Nesting of Macros

One macro can be used in the definition of another macro.

Example:

#define	M	5
#define	N	M+1
#define	SQUARE(*x*)	((*x*) * (*x*))
#define	CUBE(*x*)	(SQUARE(*x*) * (*x*))
#define	SIXTH(*x*)	(CUBE(*x*) * CUBE(*x*))

The preprocessor expands each *#define* macro, until no more macros appear in the text.

Example: The last definition is first expanded into

((SQUARE(*x*) * (*x*)) * (SQUARE(*x*) *(*x*))

since *SQUARE(x)* is still a macro, it is further expanded into

((((*x*) * (*x*)) * (*x*)) * (((*x*) * (*x*)) * (*x*)))

which is finally evaluated as x^6

Macros can be used as parameters of other macros.

Example: Given the definitions of *M* and *N*, the following macro can be defined to give the maximum of these two :

```
#define MAX(M, N) (( (M) > (N) ) ? (M) : (N) )
```

Macro calls can be nested in much the same fashion as function calls.

Example:

#define	HALF(*x*)	((*x*)/2.0)
#define	*Y*	HALF(HALF (*x*))

UNDEFINING A MACRO (*#undef*)

A defined macro can be undefined, using the statement *#undef identifier.*

This is useful when there is a need to restrict the definition only to a particular part of the program.

FILE INCLUSION

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions.

This is achieved by the preprocessor directive *#include "filename"*.

Where *filename* is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of *filename* into the source code of the program. When the *filename* is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.

Alternatively, this directive can take the form *#include* <*filename*> without double quotation marks. In this case, the file is searched only in the standard directories, and this type of declaration is used only in case of standard header files. Nesting of included files is allowed that is, an included file can include other files. However, a file cannot include itself. If an included file is not found, an error is reported and compilation is terminated.

CONDITIONAL COMPILATION DIRECTIVES (#if, #else, #elif, #endif, #ifdef, AND #ifndef)

#ifdef and #endif

The 'C' preprocessor offers a feature known as *conditional compilation,* which can be used to *switch on* or *off* a particular line or group of lines in a program. This is achieved by inserting the preprocessing commands *#ifdef* and *#endif,* which have the general form.

```
#ifdef macroname
    statement1;
    statement2;
    statement3;
#endif
```

If macroname has been *#defined,* the block of code will be processed as usual, otherwise not.

Example:

```
main( )
    {
        #ifdef OKAY
            statement1;
            statement2;
        #endif
            statement3;
    }
```

Here, statements 1and 2 would get compiled only if the macro *OKAY* has been defined and the definition of the macro has been purposely omitted. Later on if these statements are to be compiled, the only requirement is the deletion of *#ifdef* and *#endif.*

#ifndef (If Not Defined)

It works exactly opposite to *#ifdef.*

Example:

```
main( )
    {
        #ifndef PCAT        code suitable for a PC/XT
        #else               code suitable for a PC/AT
        #endif              code common to both the computers
    }
```

Thus the program is portable, that is, it is possible to work on two totally different computers.

#if and #elif

The *#if* directive can be used to test whether an expression evaluates to a non zero value or not. If the result of the expression is non zero, then subsequent lines upto a *#else, #elif* or *#endif* are compiled, otherwise they are skipped.

The conditional compilation directives can be nested as shown below:

```
#if ADAPTER == MA
   code for monochrome adapter
#else
   #if ADAPTER == CGA
      code for colour graphics adapter
   #else
      #if ADAPTER == EGA
         code for enhanced graphics adapter
      #else
         #if ADAPTER == VGA
            code for video graphics array
         #else
            code for super video graphics array
         #endif
      #endif
   #endif
#endif
```

The above program can be made more compact by using another conditional compilation directive called *#elif*. The same program using this directive can be rewritten as shown below:

```
#if ADAPTER == MA
   code for monochrome adapter
#elif ADAPTER == CGA
   code for colour graphics adapter
#elif ADAPTER == EGA
   code for enhanced graphics adapter
#elif ADAPTER == VGA
   code for video graphics array
#else
   code for super video graphics array
#endif
```

A # operator

ANSI C called # a stringizing operator. This facilitates the replacement of a parameter inside a string in the body of a macro and provides string concatenation.

Example:

#define PRINT (*v*, *f*) printf (#*v* "=" #*f*, *v*)

then macro call

PRINT (*i*, *%d*);

expands into

printf ("*i*" "=" "*%d*", *i*) ;

which after concatenations becomes:

printf ("*i = %d*", *i*);

The ## operator

ANSI C calls it the token pasting operator. It concatenates the two preprocessor tokens surrounding it into one composite token during a macro expansion.

Example:

#define processor (n) intel ##n

then macro call

processor (386)

expands into intel 386

Summary

- A preprocessor processes the source code before it passes through the compiler.
- Preprocessor directives are placed before calling the function main( ).
- #define macro replaces the identifier in the program by a predefined string.
- A preprocessor performs a literal text substitution whenever the defined name occurs.
- Macros can also contain arguments.
- One macro can be nested within the other.
- #*include* is the preprocessor directive for file inclusion.

Review Exercise

Multiple-Choice Questions

1. A preprocessor processes the _____ code before it passes through the compiler.
   (a) object code
   (b) source code

(c) executable code
(d) none of these

2. Which preprocessor directive is used to undefine an existing macro?
   (a) *#dif*
   (b) *#not def*
   (c) *##*
   (d) *#undef*
3. In a 'C' program the preprocessor directives are placed
   (a) within the main function
   (b) after the main function
   (c) before the main function
   (d) anywhere in the program
4. Which of the following is not a conditional compilation directive?
   (a) *# if*
   (b) *#if def*
   (c) *# define*
   (d) *# endif*
5. For including external files in a program the directive used is
   (a) *# attach*
   (b) *# include*
   (c) *# link*
   (d) none of these

State whether True or False

1. If the file to be included does not exist, the preprocessor flashes an error message.
2. The preprocessor can trap simple errors like missing declaration, nested comments, or mismatch of traces.
3. The following program has no errors:

```
main( )
{
   printf ("Tips" "Traps");
}
```

4. The following program will print the message infinite number of times:
   #define INFINITELOOP while(1)

```
main( )
{
   INFINITELOOP
   printf ("\n trey haired");
}
```

5. Nesting of macros is not permitted in 'C' language.

Fill in the Blanks

1. Preprocessor directives are _____ terminated by a semi-colon.
2. During a macro call, the _____ parameters are replaced by actual parameters.
3. In # *include* directive if the file name is enclosed within < and >, it is searched only in the _____.
4. The #*undef* directive _____ a macro definition only to a particular part of the program.
5. During call of a nested macro, the _____ macro is expanded first.

Descriptive Questions

1. What is the scope of a preprocessor directive within a program file?
2. What is a macro? Differentiate between macros and functions.
3. What is meant by a conditional compilation? How is a conditional compitation carried out? What preprocessor directives are available for this purpose?
4. Describe the use of arguments within a macro.
5. Write the output of the following programs:

(a)
```
#define PRINT int) printf ("%d", int)
main( )
{
   int x = 2, y = 3, z = 5;
   PRINT (x);
   PRINT (y);
   PRINT (z);
}
```

(b)
```
#define str(x) #x
#define Xstr (x) str(x)
#define oper multiply
main( )
{
   char * opername = Xstr (Oper);
   printf ("%s", opername);
}
```

(c)
```
#define MESS junk
main( )
{
   printf ("MESS");
}
```

Appendix I

STANDARD HEADER FILES AND LIBRARY FUNCTIONS

Header Files

Each C compiler provides a library of around 200 predefined functions and macros designed for use in C programs. These library functions make programming easier by providing the following:

(a) An interface to the operating system function (opening and closing files).
(b) Fast and efficient functions to perform common programming tasks (string manipulation), sparing the programmer the time and effort needed to write such functions.

For using these functions certain files need to be included in the programs which makes call to these functions.

These files are known as *header files* and they contain macro definitions, type definitions, and function declarations. These header files usually have an extension *.h*, as in *stdio.h*.

A few commonly required header files and the standard library functions that they support are listed here.

stdio.h

The header file *stdio.h* contains definitions of *constants*, *macros* and *types*, along with *function declaration* for *standard I/O functions.* These functions are listed below.

calloc	fgetchar	fputchar	getc	remove
fclose	fgetpos	fputs	getchar	rename
fcloseall	fgets	fread	gets	rewind
feof	flushall	fscanf	printf	scanf
ferror	fopen	fseek	putc	ungetc
fflush	fprintf	ftell	putchar	puts
fgetc	fputc	fwrite		

ctype.h

The *ctype.h* header file defines macros and constants and declares a global array used in character classification. The macros defined in *ctype.h* are listed below.

isalnum	isdigit	ispunct	toascii	_toupper
isalpha	isgraph	isspace	tolower	
isascii	islower	isupper	toupper	
iscntrl	isprint	isxdigit	_tolower	

string.h

The *string.h* header file declares the string manipulation functions, as listed below.

memccpy	memmove	strcmpi	strlwr	strchr
memchr	memset	strcpy	strncat	strrev
memcmp	stract	strdup	strncmp	strset
memcpy	strchr	stricmp	strncpy	strstr
memicmp	strcmp	strlen	strnicmp	strupr

math.h

The header file *math.h* contains function declarations for all floating point math routines as listed below.

abs	atof	cosh	log	sinh
acos	cabs	exp	log10	sqrt
asin	ceil	fabs	pow	tan
atan	cos	floor	sin	tanh

stdlib.h

The *stdlib.h* header file contains function declarations for the following functions.

abort	bsearch	itoa	rand	system
abs	calloc	labs	realloc	tolower
atof	exit	malloc	srand	toupper
atoi	free	perror	strtod	
atol	gcvt	qsort	strtol	

stdarg.h

The header file *stdarg.h* defines macros that allow you to access arguments in functions which are passed variable number of arguments, such as *vprintf().* These macros are defined to be machine independent and portable. These macros are listed below.

va_arg	va_end	va_start

time.h

The time.h header file declares the following time related functions.

asctime	difftime	clock	gmtime	ctime
time				

Library Functions

There is a vast collection of functions some of them are grouped together and listed below.

String Functions

stpcpy	
Function	Copies one string into another.
Syntax	char *stpcpy(char **dest,* const char **src);*
Prototype in	string.h
Remarks	*stpcpy* copies the string *src* to *dest,* stopping after the terminating null character has been reached.
Return value	*stpcpy* returns *dest + strlen(src).*
Portability	*stpcpy* is available on UNIX systems.

strcat	
Function	Appends one string to another.
Syntax	char *strcat(char **dest,* const char **src);*
Prototype in	string.h
Remarks	*strcat* appends a copy of *src* to the end of *dest*. The length of the resulting string is *strlen(dest) + strlen(src).*
Return value	*strcat* returns a pointer to the concatenated strings.
Portability	*strcat* is available on UNIX systems and is compatible with ANSI C. It is defined in K&R C.

strchr	
Function	Scans a string for the first occurrence of a given character.
Syntax	char *strchr(const char **s,* int *c);*
Prototype in	string.h
Remarks	*strchr* scans a string in the forward direction, looking for a specific character. *strchr* finds the first occurrence of the character *c* in the string*s*. The null-terminator is considered to be the part of the string, so that, for example, *strchr(strs, 0)* returns a pointer to the terminating null character of the string *strs.*
Return value	*strchr* returns a pointer to the first occurrence of the character *c* in *s*; if *c* does not occur in *s, strchr* returns null.
Portability	*strchr* is available on UNIX systems and is compatible with ANSI C.

strcmp	
Function	Compares one string to another.
Syntax	int strcmp(const char **s1,* const char **s2);*
Prototype in	string.h
Remarks	*strcmp* performs an unsigned comparison of *s1, s2* starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

Return value	*strcmp* returns a value that is < 0 if *s1* is less than *s2* = = 0 if *s1* is the same as *s2* > 0 if *s1* is greater than *s2*
Portability	*strcmp* is available on UNIX systems and is compatible with ANSI C.

strcmpi

Function	Compares one string to another, without case sensitivity.
Syntax	int strcmpi(const char **s1,* const char **s2*);
Prototype in	string.h
Remarks	*strcmpi* performs an unsigned comparison of *s1, s2* without case sensitivity (same as *stricmp*-implemented as a macro). It returns value (<0, 0, or >0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it). The routine *strcmpi* is the same, respectively, as stricmp.*strcmpi* is implemented via macro in *string.h* and translates calls from *strcmpi* to *stricmp.* Therefore, in order to use *strcmpi,* you must *#include* the header file *string.h* for the macro to be available. This macro is provided for compatibility with other C compilers.
Return value	*strcmpi* returns an *int* value that is < 0 if *s1* is less than *s2* = =0 if *s1* is the same as *s2* > 0 if *s1* is greater than *s2*

strcpy

Function	Copies one string into another.
Syntax	char* strcpy(char **dest,* const char **src);*
Prototype in	string.h
Remarks	copies string *src* to *dest,* stopping after the terminating null character has been moved.
Return value	*strcpy* returns *dest.*
Protability	*strcpy* is available on UNIX systems and is compatible with ANSI C.

strlen

Function	Calculates the length of a string.
Syntax	size_t strlen(const char **s);*
Prototype in	string.h
Remarks	*strlen* calculates the length of s.
Return value	*strlen* returns the number of characters in *s,* not counting the null-terminating character.
Portability	*strlen* is available on UNIX systems and is compatible with ANSI C.

strncmpi

Function	Compares a portion of one string to a portion of another, without case sensitivity.

Syntax	int strncmpi(const char *s1, const char *s2 size_tn);
Prototype in	string.h
Remarks	*strncmpi* performs a signed comparison of *s1, s2* for a maximum length of *n* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until *n* characters have been examined. The comparison is not case sensitive. (*strncmpi* is the same as *strnicmp*-implemented as a macro). It returns a value (<0, 0, or >0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it). The routines *strncmp* and *strncmpi* are the same; *strncmpi* is implemented via a macro in *string.h* that translates calls from *strncmpi* to *strnicmp.* you must include the header file *string.h* for the macro to be available. This macro is provided for compatibility with other C compilers.
Return value	*strnicmp* returns an *int* value that is < 0 if *s1* is less than *s2* = =0 if *s1* is the same as *s2* > 0 if *s1* is greater than *s2*

strrev

Function	Reverses a string.
Syntax	char *strrev(char *s);
Prototype in	string.h
Remarks	*strrev* changes all characters in a string to reverse order, except the terminating null character. For example, it would change string/0 to gnirts/0.
Return value	*strrev* returns a pointer to the reversed string. There is no error return.

Some other string manipulation functions and their brief description are as given below :

strcspn	:	Scans a string for initial segment not containing any subset of a given set of characters.
strdup	:	Copies a string into a newly-created location.
strerror	:	Returns a pointer to an error message string.
stricmp	:	Compares one string to another, without case sensitivity.
strlwr	:	Converts uppercase letters in a string to lowercase.
strncat	:	Appends a portion of one string to another.
strncmp	:	Compares a portion of one string to a portion of another.
strncmpi	:	Compares a portion of one string to a portion of another; without case sensitivity.
strncpy	:	Copies a given number of bytes from one string into another, truncating or padding as necessary.
strnicmp	:	Compares a portion of one string to a portion of another, without case sensitivity.
strnset	:	Sets a specified number of characters in a string to a given character.

strpbrk	:	Scans a string for the first occurrence of any character from a given set.
strrchr	:	Scans a string for the last occurrence of a given character.
strset	:	Sets all character in a string to a given character.
strspn	:	Scans a string for the first segment that is a subset of a given set of characters.
strstr	:	Scans a string for the occurrence of a given substring.
strtod	:	Converts a string to a double value.
strtok	:	Searches one string for tokens, which are separated by delimiters defined in a second string.
strtol	:	Converts a string to a long value.
strtoul	:	Converts a string to an unsigned long in the given radix.
strupr	:	Converts lowercase letters in a string to uppercase.

Mathematical Functions

Several Functions are provided by C for mathematical functions some of them are listed below:

abs

Function	Returns the absolute value of an integer
Syntax	int abs(int *x*);
Prototype in	math.h, stdlib.h
Remarks	*abs* returns the absolute value of the integer argument *x*. If *abs* is called when stdlib.h has been included, it will be treated as a macro that expands to inline code. If you want to use the *abs* function instead of the macro, include *#undef abs* in your program, after the *#include <stdlib.h>*.
Return value	*abs* returns an integer in the range of 0 to 32, 767, with the exception that an argument of -32, 768 is returned as -32, 768.
Portability	*abs* is available on UNIX systems and is compatible with ANSI C.

acos

Function	Calculates the arc cosine.
Syntax	double acos(double *x*);
Prototype in	math.h
Remarks	*acos* returns the arc cosine of the input value. Arguments to *acos* must be in the range -1 to 1. Arguments outside that range will cause *acos* to return 0 and set *errno* to EDOM Domain error
Return value	*acos* returns a value in the range 0 to *pi*. Error-handling for this routine can be modified through the function *matherr*.
Portability	*acos* is available on UNIX systems and is compatible with ANSI C.

asin

Function	Calculates the arc sine.
Syntax	double asin(double *x*);

Prototype in	math.h
Remarks	*asin* returns the arc sine of the input value. Arguments to *asin* must be in the range -1 to 1. Arguments outside that range will cause *asin* to return 0 and set *errno* to EDOM Domain error.
Return value	*asin* returns a value in the range 0 to *pi/2*. Error-handling for this routine can be modified through the function *matherr*.
Portability	*asin* is available on UNIX systems and is compatible with ANSI C.

atan

Function	Calculates the arc tangent.
Syntax	double atan(double *x*);
Prototype in	math.h
Remarks	*atan* calculates the arc tangent of the input value.
Return value	*atan* returns a value in the range *-pi/2* to *pi/2*. Error-handling for this routine can be modified through the function *matherr*.
Portability	*atan* is available on UNIX systems and is compatible with ANSI C.

atof

Function	Converts a string to a floating-point number.
Syntax	double atof(const char **s*);
Prototype in	math.h, stdlib.h
Remarks	*atof* converts a string pointed to by *s* to double; this function recognizes the character representation of a floating-point number, made up of the following: • an optional string of tabs and spaces • an optional sign • a string of digits and an optional decimal point (the digit can be on both sides of the decimal point) • an optional *e* or *E* followed by an optional signed integer The characters must match this generic format : [ws] [sn] [ddd] [.] [ddd] [fmt] [sn] [ddd] *atof* also recognizes +INF and -INF for plus and minus infinity, and +NAN and- NAN for Not-a-Number. In this function, the first unrecognized character ends the conversion.
Return value	*atof* returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (double), the return value is 0. If there is an overflow, *atof* returns plus or minus HUGE_VAL, and *matherr* is not called.
Portability	*atof* is available on UNIX systems and is compatible with ANSI C.

ceil

Function	Rounds up.
Syntax	double ceil(double *x*);

Prototype in	math.h
Remarks	*ceil* finds the smallest integer not less than *x*.
Return value	*ceil* returns the integer found (as a double).
Portability	*ceil* is available on UNIX systems and is compatible with ANSI C.

cos

Function	Calculates the cosine.
Syntax	double cos(double *x*);
Prototype in	math.h
Remarks	*cos* returns the cosine of the input value. The angle is specified in radians.
Return value	*cos* returns a value in the range -1 to 1. Error-handling for this routine can be modified through the function *matherr*.
Portability	*cos* is available on UNIX systems and is compatible with ANSI C.

exp

Function	Calculates the exponential e to the x^{th} power.
Syntax	double exp(double *x*);
Prototype in	math.h
Remarks	*exp* calculates the exponential function e^x.
Return value	*exp* returns e^x. Sometimes the arguments passed to *exp* produce results that overflow or are incalculable. When the correct value overflows, *exp* returns the value HUGE_VAL. Results of excessively large magnitude can cause *errno* to be set to ERANGE Result out of range. On underflow, *exp* returns 0.0, and *errno* is not changed. Error-handling for *exp* can be modified through the function *matherr.*
Portability	*exp* is available on UNIX systems and is compatible with ANSI C.

fabs

Function	Returns the absolute value of a floating-point number.
Syntax	double fabs(double *x*);
Prototype in	math.h
Remarks	*fabs* calculates the absolute value of *x,* a double.
Return value	*fabs* returns the absolute value of *x*. There is no return on error.
Portability	*fabs* is available on UNIX systems and is compatible with ANSI C.

floor

Function	Rounds down.
Syntax	double floor(double *x*);
Prototype in	math.h
Remarks	*floor* finds the largest integer not greater than *x.*
Return value	*floor* returns the integer found (as a *double).*
Portability	*floor* is available on UNIX systems and is compatible with ANSI C.

log	
Function	Calculates the natural logarithm of *x*.
Syntax	double log(double *x*);
Prototype in	math.h
Remarks	*log* calculates the natural logarithm of *x*.
Return value	On success, *log* returns the calculated value *In(x).* If the argument *x* passed to *log* is less than or equal to 0, *errno* is set to EDOM Domain error. When this error occurs, *log* returns the value negative HUGE_VAL. Error-handling for *log* can be modified through the function *matherr.*
Portability	*log* is available on UNIX systems and is compatible with ANSI C.

log10	
Function	Calculates log_{10} (*x*).
Syntax	double log10(double *x*);
Prototype in	math.h
Remarks	*log10* calculates the base 10 logarithm of *x.*
Return value	On success, *log10* returns the calculated value log_{10} *(x).* If the argument *x* passed to *log10* is less than or equal to 0, *errno* is set to EDOM Domain error. When this error occurs, *log10* returns the value negative HUGE_VAL. Error-handling for *log10* can be modified through the function *matherr.*
Portability	*log10* is available on UNIX systems and is compatible with ANSI C.

pow	
Function	Calculates *x* to the power of *y*.
Syntax	double pow(double *x,* double *y*);
Prototype in	math.h
Remarks	*pow* calculates x^y.
Return value	On success, *pow* returns the calculated value x^y . Sometimes the arguments passed to *pow* produce results that overflow or are incalculable. When the correct value would overflow, *pow* returns the value HUGE_VAL. Results of excessively large magnitude can cause *errno* to be set to ERANGE Result out of range. *errno* is set to EDOM Domain error. If the argument *x* passed to *pow* is less than or equal to *0*, and *y* is not a whole number. When this error occurs, *pow* returns the value negative HUGE_VAL. If the argument *x* and *y* passed to *pow* are both 0, *pow* returns 1. Error-handling for *pow* can be modified through the function *matherr.*
Portability	*pow* is available on UNIX systems and is compatible with ANSI C.

sin	
Function	Calculates sine.
Syntax	double sin(double *x*);
Prototype in	math.h

Remarks	*sin* computes the sine of the input value. Angles are specified in radians. Error-handling for this routine can be modified through the function *matherr.*
Return value	*sin* returns the sine of the input value.
Portability	*sin* is available on UNIX systems and is compatible with ANSI C.

sqrt

Function	Calculates the positive square root of input value.
Syntax	double sqrt(double *x*);
Prototype in	math.h
Remarks	*sqrt* calculates the positive square root of the input value. Error-handling for *sqrt* can be modified through the function *matherr.*
Return value	On success, *sqrt* returns the value calculated, the positive square root of *x.* If *x* is negative, *errno* is set to EDOM Domain error.
Portability	*sqrt* is available on UNIX systems and is compatible with ANSI C.

tan

Function	Calculates the tangent.
Syntax	double tan(double *x*);
Prototype in	math.h
Remarks	*tan* calculates the tangent. Angles are specified in radians. Error-handling for this routine can be modified through the function *matherr*.
Return value	*tan* returns the tangent of *x,* and value for valid angles. For angles close to *pi/2* or *-pi/2, tan* returns 0 and *errno* is set to ERANGE Result out of range.
Portability	*tan* is available on UNIX systems and is compatible with ANSI C.

Some more mathematical routines supported by math.h are as under :

atan2	:	Calculates the arc tangent of *y/x.*
cabs	:	Absolute value of complex number.
cosh	:	Calculates the hyperbolic cosine.
div	:	Divides two integers, returning quotient and remainder.
fmod	:	Calculates *x* modulo *y,* the remainder of *x/y.*
frexp	:	Splits *double* number into mantissa and exponent.
hypot	:	Calculates hypotenuse of a right angle triangle.
ldexp	:	Calculates $x \times 2^{exp}$.
ldiv	:	Divides two longs, returns quotient and remainder.
matherr	:	User -modifiable math error handler.
modf	:	Splits *double* into integer and fraction parts.
poly	:	Generates a polynomial from arguments.
pow10	:	Calculates 10 to the power of *p.*
sinh	:	Calculates hyperbolic sine.
tanh	:	Calculates the hyperbolic tangent.

Some mathematical routines supported by *stdlib.h* are as under :

atoi	:	Converts a string to an integer.
atol	:	Converts a string to a long.
ecvt	:	Converts a floating-point number to a string.
fcvt	:	Converts a floating-point number to a string.
gcvt	:	Converts a floating-point number to a string.
itoa	:	Converts an integer to a string.
labs	:	Gives long absolute value.
ltoa	:	Converts a long to a string.
rand	:	Random number generator.
random		Random number generator.
randomize	:	Initializes random number generator.
srand	:	Initializes random-number generator.

Date and Time Functions

asctime

Function	Converts date and time.to ASCII
Syntax	char *asctime(const struct tm **tblock*);
Prototype in	time.h
Remarks	*asctime* converts a time stored as a structure in **tblock* to a 26-character string of the same form as the *ctime* string: *Sun Sep 16 01:03:52 1973\n\0*
Return value	*asctime* returns a pointer to the character string containing the date and time. This string is a static variable that is overwritten with each call to *asctime*.
Portability	*asctime* is available on UNIX systems and is compatible with ANSI C.

Example Program:

```
#include <stdio.h>

#include <time.h>

main()

{

          struct tm *tm_now;

          time_t secs_now;

          char *str_now;

          /* get time in seconds */
```

```
            time(&secs_now);

            /* make it a string */

            str_now = ctime(&secs_now);

            printf("\nThe number of seconds since Jan 1, 1970 is %ld\n",
secs_now);

            printf("\nIn other words, the current time is %s", str_now);

            /* make it a structure */

            tm_now = localtime(&secs_now);

            printf("\nFrom the structure: day %d-%02d-%02d-%02d
%02d:%02d:%02d",

            tm_now->tm_yday, tm_now->tm_mon, tm_now->tm_mday,
            tm_now->tm_year, tm_now->tm_hour, tm_now->tm_min,
            tm_now->tm_sec);

            /* from structure to string */

            str_now = asctime(tm_now);

            printf("\nOnce more, the current time is %s", str_now);

  }
```

Output: *The number of seconds since Jan 1, 1970 is 315594553.*
In other words, the current time is Tue Jan 01 12:09:12 1980
From the structure : day 0 00-01-80 12:09:13
Once more, the current time is Tue Jan 01 12:09:12 1980

clock

Function	Determines processor time
Syntax	clock_t clock(void);
Prototype in	time.h
Remarks	*clock* can be used to determine the time interval between two events. To determine the time in seconds, the value returned by *clock* should be divided by the value of the macro CLK_TCK.
Return value	The *clock* function returns the processor time elapsed since the beginning of the program invocation. If the processor time is not available or its value cannot be represented, the function returns the value-1.
Portability	*clock* is compatible with ANSI C.

Example Program:

```
#include <time.h>

#include <stdio.h>

void main()

{

          clock_t start, end;

          start = clock();    /*  Code to be timed goes here */

          end = clock();

          printf("\nThe time was: %f\n", (end - start) /CLK_TCK);

}
```

getdate

Function	Gets system date.
Syntax	void getdate(struct date **datep);*
Prototype in	dos.h
Remarks	*getdate* fills in the *date* structure (pointed to by *datep)* with the system's current date.
	The *date* structure is defined follows :

```
struct date  {

                int da_year; /* current year */

                char da_day; /* day of the month */

                char da_mon; /* month (1 = Jan) */

             };
```

Return value	None.
Portability	*getdate* is unique to DOS.

Example Program:

```
#include <stdio.h>

#include <dos.h>
```

```
main()

{

          struct date today;

          struct time now;

          getdate (&today);

          printf ("\nToday's date is %d%d%d\n", today.da_mon,
today.da_day, today.da_year);

          gettime (&now);

          printf ("\nThe time is %02d:%02d:02d.%02d\n", now.ti_hour,
now.ti_min, now.ti_sec, now.ti_hund);

}
```

Output: Today's date is 1/1/1980
The time is 17:08:22.42

gettime

Function	Gets system time.
Syntax	void gettime(struct time **timep*);
Prototype in	dos.h
Remarks	*gettime* fills in the *time* structure pointed to by *timep* with the system's current time.

The *time* structure is defined as follows:

```
struct time {

   unsigned char ti_min;  /* minutes */

   unsigned char ti_hour; /* hours */

   unsigned char ti_hund; /* hundredths of seconds */

   unsigned char ti_sec;  /* seconds */

};
```

Return value	None.
Portability	*gettime* is unique to DOS.

setdate

Function	Sets DOS date.
Syntax	void setdate(struct date **datep*);
Prototype in	dos.h
Remarks	*setdate* sets the system date (month, day and year) to that in the *date* structure pointed to by *datep.* The *date* structure is defined as follows :

```
struct date

{

   int da_year;         /* current year */

   char da_day;      /* day of the month */

   char da_mon;      /* month (1 = Jan) */  };
```

Return value	None.
Portability	*setdate* is unique to DOS.

settime

Function	Sets system time.
Syntax	void settime(struct time **timep*);
Prototype in	dos.h
Remarks	*settime* sets the system time to the values in the time structure pointed to by *timep*. The *time* structure is defined as follows :

```
struct time

{  unsigned char ti_min;  /* minutes */

   unsigned char ti_hour; /* hours */

   unsigned char ti_hund; /* hundredths of seconds */

   unsigned char ti_sec;  /* seconds */

};
```

Return value	None.
Portability	*settime* is unique to DOS.

time

Function	Gets time of day.
Syntax	time_t time(time_t **time);*

Prototype in	time.h
Remarks	*time* gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by *timer,* provided that *timer* is not a null pointer.
Return value	*time* returns the elapsed time is seconds, as described.
Portability	*time* is available on UNIX systems and is compatible with ANSI C.

Some other time and date routines are :

ctime (time.h)	:	Converts date and time to a string.
difftime (time.h)	:	Computes the difference between two times.
dostounix (dos.h)	:	Converts date and time to UNIX time format.
ftime (sys\timeb.h)	:	Stores current time in *timeb* structure.
gmtime (time.h)	:	Converts date and time to Greenwich Mean Time (GMT).
localtime (time.h)	:	Converts date and time to a structure.
stime (time.h)	:	Sets system date and time.
tzset (time.h)	:	Sets a value of global variable *daylight, timezone,* and *tzname.*
unixtodos (dos.h)	:	Converts date and time to DOS format.

Variable Argument List Functions

These routines are for use when accessing variable argument lists. (such as with vscanf, vprintf etc.)

va_...

Function	Implement a variable argument list.

Syntax	void va_start(va_list *param, lastfix*);
	type va_arg(va_list *param, type*);
	void va_end(va_list *param*);

Prototype in	stdarg.h
Remarks	Some C functions, such as *vfprintf* and *vprintf*, take variable argument lists in addition to taking a number of fixed (known) parameters. The *va_...* macros provide a portable way to access these argument lists.They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being passed. The header file stdarg.h declares one type (va_list), and three macros (*va_start, va_arg,* and *va_end*). ***va_list :*** array holds information needed by *va_arg* and *va_end*. When a called function takes a variable argument list, it declares a variable *param* of type *va_list*. ***va_start :*** This routine (implemented as a macro) sets *param* to point to the first of the variable arguments being passed to the funcion. *va_start* must be used before the first call to *va_arg* or *va_end va_start* takes two parameters: *param* and *lastfix.* (*param* is explained under *va_list* in the preceding paragraph; *lastfix* is the name of the last fixed parameter being passed to the called function.)

va_arg : This routine (also implemented as a macro) expands to an expression that has the same type and value as next argument being passed (one of the variable arguments). The variable *param* to *va_arg* should be the same *param* that *va_start* initialized. The first time va_arg is used, it returns the first argument in the list. Each successive time *va_arg* is used, it returns the next argument in the list. It does this by first de-referencing *param* to point to the following to the item. *va_arg* uses the *type* to both perform the de-reference and to locate the following item. Each successive time *va_arg* is invoked, it modifies *param* to point to the next argument in the list.

va_end : This macro helps the called function perform a normal return. *va_end* might modify *param* in such a way that it cannot be used unless *va_start* is re-called. *va_end* should be called after *va_arg* has read all the arguments; failure to do so might cause strange, undefined behavior in your program.

Return value — *va_start* and va_end return no values; *va_arg* returns the current argument in the list (the one that *param* is pointing to).

Portability — *va_arg, va_start,* and *va_end* are available on UNIX systems.

Example Programs:

1.

```
#include <stdio.h>

#include <stdarg.h>

void sum(char *msg, ...) /* calculate sum of a 0 terminated
list */

{

   int arg, total = 0;

   va_list ap;

   va_start(ap, msg);

   while ((arg = va_arg(ap, int)) != 0)

   {

      total += arg;

   }

   printf (msg, total);

}
```

```
main()
{
   sum ("\nThe total of 1+2+3+4 is %d\n", 1, 2, 3, 4, 0);
}
```

Output: *The total of 1+2+3+4 is 10*

2.

```
#include <stdio.h>
#include <stdarg.h>
void error(char *format, ...)
{
   va_list argptr;
   printf ("error: ");
   va_start (argptr, format);
   vprintf (format, argptr);
   va_end (argptr);
}
main()
{
   int  value = -1;
   error ("\nThis is just an error message.\n");
   error ("\nInvalid value %d encountered\n", value);
}
```

Output: *error :This is just an error message.*
error :Invalid value -1 encountered

Other variable argument list function are :

vfprintf (stdio.h) : Writes formatted output to a stream.

vfscanf (stdio.h)	:	Scans and formats input from a stream.
vprintf (stdarg.h	:	Write formatted output to *stdout.*
vscanf (stdarg.h	:	Scans and formats input from *stdin*.
vsprintf (stdarg.h)	:	Writes formatted output to a string
vsscanf (stdarg.h)	:	Scans and formats input from a stream.

Utility Functions

abort()	Abnormally terminates a process. Syntax: *void abort()*;
bsearch()	Binary search of an array. Syntax: *void bsearch (void *key, void *base, unsigned nelem, unsigned width, int (*fcmp) (void*, CONST void*));*
calloc()	Allocates main memory. Syntax:*void calloc (unsigned nitems, unsigned size);*
exit()	Terminates execution of a program. Syntax: *void exit (int status);*
free()	Frees allocated block. Syntax:*void free (void *block);*
malloc()	Allocates main memory. Syntax:*void malloc (unsigned size);*
perror()	Prints a system error message. Syntax:*void perror (char *s);*
qsort()	Sorts using the quicksort algorithm. Syntax:*void qsort (void *base, unsigned nelem, unsigned width, int (*fcmp) (void*, void*));*
realloc()	Reallocates the main memory. Syntax:*void *realloc (void * block, unsigned size);*
system()	Issues a DOS command. Syntax:*int system (char *command);*
tolower()	Translates characters to lowercase. Syntax:*int tolower (int ch);*
toupper()	Translates characters to uppercase. Syntax:*int toupper (int ch);*

Character Class Test Functions

isalnum()	Tests whether a character is an alphabet or a number. Syntax:i*salnum (int c);*
isalpha()	Character classification macro that returns nonzero if c is a letter (A-Z or a-z). Syntax:*int isalpha (int c);*

isascii()	Tests whether a character is an ascii (0 to 127) character. Syntax:*iasascii (int c);*
iscntrl()	Tests whether a character is a control character. Syntax:*iscntrl (int c);*
isdigit()	Character classification macro that returns nonzero value if c is a digit ('0'-'9'). Syntax:*int isdigit (int c);*
isgraph()	Character classification macro that returns nonzero if c is a printing character, space character is excluded. Syntax:*int isgraph (int c);*
islower()	Character classification macro that returns non zero value if c is a lower case alphabet and zero otherwise. Syntax: *int islower (int c);*
isprint()	Character classification macro that returns value is c is a printable character (ASCII 32 to 126) and non-zero otherwise. Syntax: *int isprint (int c);*
ispunct()	Character classification macro that returns non-zero value if c is a punctuation character *(iscntrl or isspace).* Syntax: *int ispunct (int c);*
isspace()	Character classification macro that returns non-zeroif c is a space, tab, carriage return, newline, vertical tab, formfeed. Syntax:int isspace (int c);
isupper()	Character classification macro that return non-zero if c is an uppercase letter (A-Z). Syntax:int isupper (int c);
isxdigit()	Character classification macro that returns non-zero if c is a hexadecimal digit (0-9, A-F, a-f). Syntax:int isxdigit (int c);
toascii()	Translates character to ASCII format. Syntax : int toascii (int ch);
tolower()	Translates character to uppercase. Syntax : int tolower (int ch);
toupper( )	Translates character to uppercase. Syntax : int toupper (int ch) ;
_tolower( )	Translates character tab lowercase. Syntax : int _tolower (int ch) ;
_toupper()	Translates character to uppercase. Syntax : int _touppter (int ch);

Appendix II

ASCII VALUES OF CHARACTERS

Note : The first 32 characters and the last character are control characters — they cannot be printed.

ASCII		ASCII		ASCII		ASCII	
Value	*Character*	*Value*	*Character*	*Value*	*Character*	*Value*	*Character*
000	NUL	018	DC2	036	$	054	6
001	SOH	019	DC3	037	%	055	7
002	STX	020	DC4	038	&	056	8
003	ETX	021	NAK	039	‘	057	9
004	EOT	022	SYN	040	(	058	:
005	ENQ	023	ETB	041	)	059	;
006	ACK	024	CAN	042	*	060	<
007	BEL	025	EM	043	+	061	=
008	BS	026	SUB	044	‘	062	>
009	HT	027	ESC	045	-	063	?
010	LF	028	FS	046	.	064	@
011	VT	029	GS	047	/	065	A
012	FF	030	RS	048	0	066	B
013	CR	031	US	049	1	067	C
014	SO	032	blank	050	2	068	D
015	SI	033	!	051	3	069	E
016	DLE	034	“	052	4	070	F
017	DC1	035	#	053	5	071	G

ASCII		ASCII		ASCII		ASCII	
Value	*Character*	*Value*	*Character*	*Value*	*Character*	*Value*	*Character*
072	H	086	V	100	d	114	r
073	I	087	W	101	e	115	s
074	J	088	X	102	f	116	t
075	K	089	Y	103	g	117	u
076	L	090	Z	104	h	118	v
077	M	091	[	105	i	119	w
078	N	092	\	106	j	120	x
079	O	093	]	107	k	121	y
080	P	094	↑	108	l	122	z
081	Q	095	-	109	m	123	{
082	R	096	‘	110	n	124	|
083	S	097	a	111	o	125	}
084	T	098	b	112	p	126	~
085	U	099	c	113	q	127	DEL

Model Question Papers

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(July 2007)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 Which of the following is a valid octal constant?

(a) 32 (b) 032

(c) 049 (d) 0x49

1.2 Which of the following switch statement is not a valid statement to print "RED" if a character variable 'color' has the value 'R' or 'r'?

(a) switch (color) { case 'R': case 'r'; printf("RED"); break; }

(b) switch (color) { case 'R': printf("RED"); break; case 'r': printf("RED"); break; }

(c) switch (toupper(color)) { case 'R': printf("RED"); break; }
(d) switch (color) { case 'R' || 'r': printf("RED"); break; }

1.3 What will be the output of the following code segment, if the function is called as larger (10,20)?

```
int larger (int x, int y){
   int max = x;
   if (max <y){
      max = y;
      return y;
   }
   else
   return x;
   printf ("Larger of %d and %d is %d", x,y, max);
}
```

(a) Program will not compile as the function has two return statements
(b) Program will not compile as no statement is allowed after return statement
(c) Larger of 10 and 20 is 20
(d) No output

1.4 Given the code segment:

```
char a[] = "abc", *p;
```

Which of the following assigns the starting address of the string "abc" to p?

(a) p = a; (b) p = &a;
(c) p = *a; (d) *p = a;

1.5 To read and write an existing file without overwriting, the following mode is used

(a) r (b) w
(c) r+ (d) w+

1.6 If an array is defined as static char a[10]; then the elements of *a* will be set to

(a) an undetermined value (b) zero
(c) blank character (d) character '~0'

1.7 For the code segment

```
struct DOB {int date, month, year;};
struct person {char name[30];
struct DOB birthdate;}p, *ptr=&p;
```

Which of the following is not a valid expression to access year of birth date?

(a) ptr -> birthdate.year (b) (*ptr).birthdate.year
(c) ptr.birthdate.year (d) p.birthdate.year

1.8 Which of the following statement is false for the statement?

```
main (int ac, char *av[])
```

(a) av is an array of pointers to strings
(b) av[0] represents the name of the program under execution
(c) the formal arguments names have to be argc and argv only
(d) the main function can return an integer to the calling function/program

1.9 What will be the output of the following?

```
main () {
   int a ='A';
   printf (("%d", a);
```

(a) 65 (b) A
(c) a
(d) the program will not compile as an integer variable is assigned a character constant

1.10 Consider the following code segment:

```
for (odd_sum=0, j=1; ***; j+=2)
odd_sum+=j;
```

In order to sum all the odd numbers between 1 to 100; which of the following statements cannot replace ***?

(a) j < = 99 (b) j < 99
(c) j < = 100 (d) j < 100

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 An escape sequence begins with a backward slash followed by an alphabetical character.
2.2 The for loop can be used only for the cases when the number of passes is known in advance.
2.3 Let an array **arr** be a member of a structure S. If S is passed to a function, test as test(S) then the changes in arr, if any, by test will not be reflected in the calling function.
2.4 In a recursive function with local variables, a different set of local variables with the same name are created during each call.
2.5 Two enumeration constants defined in an enumeration definition can have same integral value.
2.6 The sizeof operator can only be used with variables that are allocated space using malloc() function.
2.7 If u is a union variable, then using isalpha(u) it is possible to know whether u is storing an alphabet or not.
2.8 If a file is created with fwrite() function, then it is valid to read it using fscanf() function.
2.9 NULL is a keyword in C.
2.10 A function name can be passed as an argument to another function.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 Self referential data structure	A. Arrays
3.2 Converting to a different data type	B. Flowchart
3.3 Creating new data type	C. while (0)
3.4 Removing repetitive coding	D. register variables

3.5	Infinite loop	E.	recursion
3.6	Pictorial representation of logic	F.	Function
3.7	Request to compiler	G.	Declaration
3.8	Defining constants	H.	typedef
3.9	Global variables	I.	#define
3.10	Space allocation to variables	J.	static
		K.	algorithm
		L.	linked lists
		M.	typecasting
		N.	Definition
		O.	External variables
		P.	for (;;)

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a)	actual	(b)	4	(c)	character
(d)	false	(e)	self-referencial structure	(f)	formal
(g)	stream oriented	(h)	heterogeneous	(i)	declaration
(j)	1	(k)	5	(l)	text
(m)	unknown	(n)	non linear	(o)	union
(p)	pointer	(q)	0	(r)	linear
(s)	sparse	(t)	true	(u)	unformatted
(v)	definition	(w)	integer	(x)	linked list

4.1 The data structure with most of the entries as 0 (zero) is a (n) _____.
4.2 A null terminated array of _____ is a string.
4.3 A constant can be valid _____ argument to a function.
4.4 The statement struct point {int x,y;} is a structure _____.
4.5 Linked list is a _____ data structure.
4.6 The expression pv + 3 is valid but not pv*3 if pv is a(n) _____ variable.
4.7 A(n) _____ file can be created with specially written program only.
4.8 The size of an array defined as char color[] ='Blue'; is _____.
4.9 In expression ((j+k>10)|| (n>-3)), (n>-3) will be evaluated if (j+k > 10) is _____.
4.10 The loop do {...} while (0); will be executed _____ times.

PART TWO

(Answer Any FOUR Questions)

5. (a) Write the C program to compute the following series:
$1 - x + x^2/2 - x^3/6 + x^4/24 + \ + (-1)^n x^n/n!$
Where n and x is to be accepted by the user.

(b) Develop a flowchart and then write a 'C' program to sort strings passed to the program through the command line arguments. Also display the sorted strings. **(6+9)**

6. (a) Define a structure to store roll_no, name and marks of a student.
(b) Using the structure of Q6. (a), above, write a C program to create a file "student.data". There must be one record for every student in the file. Accept the data from the user.
(c) Using the "student.dat" of Q6. (b), above, write a C program to search for the details of the student whose name is entered by the user. **(3+6+6)**

7. (a) Write a 'C' function to reverse a singly linked list by traversing it only once.
(b) Write a 'C' function to remove those nodes of a singly linked list which have duplicate dat(a) Assume that the linked list is already in ascending order. **(7+8)**

8. (a) What do you understand by loading and linking of a program?
(b) Write a 'C' function to generate the following figure for $n = 7$.

```
        1
      1 3
    1 3 5
  1 3 5 7
    1 3 5
      1 3
        1
```

The value of n is passed to the function as an argument. Print the triangle only if n is odd otherwise print an error message.

(c) Write a ‘C’ function to arrange the elements of an integer array in such a way that all the negative elements are before the positive elements. The array is passed to it as an argument. **(3+6+6)**

9. (a) Write a recursive function in ‘C’ to count the number of nodes in a singly linked list.
(b) Develop a flowchart and then write a ‘C’ program to add two very large positive integers using arrays. The maximum number of digits in a number can be 15. **(5+10)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(January 2007)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART TWO

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 The && and || operators
- (a) compare two numeric values
- (b) combine two numeric values
- (c) compare two boolean values
- (d) combine two boolean values

1.2 The break statement causes an exit
- (a) only from innermost loop
- (b) only from innermost switch
- (c) from the innermost loop or switch
- (d) none of the above

1.3 Assuming var1 has value 20. What will following code print?

```
printf("%d%d\n", var1--,++var1);
```

(a) 20 20 (b) 19 20
(c) 20 21 (d) 21 22

1.4 When accessing a structure member, the identifier to the left of the dot operator is the name of
- (a) a structure member (b) a structure tag
- (c) a structure variable (d) the keyword struct

1.5 A static automatic variable is used to
(a) make a variable visible to several functions
(b) retain a value when a function is not executing
(c) conserve memory when a function is not executing
(d) none of the above

1.6 Which of the following directive creates functions like macros?
(a) # include (b) # define
(c) # undef (d) # ifdef

1.7 Which format specification is used in printf statement to print hexadecimal format
(a) %i (b) %c
(c) %x (d) %u

1.8 What will be the output of the following program:

```
main()
{
   int val =500;
   int *ptr = &val;
   int **ptr1 = &ptr;
   printf ("val = %d", **ptr1);
}
```

(a) 500 (b) address of ptr
(c) contents of ptr (d) none of the above

1.9 Size of operator returns the size in bytes of
(a) identifier (b) type
(c) identifier or type (d) array

1.10 The value of variable x after executing the following code will be:

```
val = -200;
x=(val >=0) ? val : -val
```

(a) 0 (b) 200
(c) –200 (d) 1

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 The #undef directive removes a name previously defined with #define directive.
2.2 The 'C' program can have only one command line argument.
2.3 The goto statement causes control to go to a function.
2.4 If you don't use a return type in the function declaration, the compiler assumes that the function does not return anything.
2.5 An array element is accessed using the dot operator.
2.6 Continue statement skips all subsequent statements in the loop body and triggers the next iteration for the loop.

2.7 The strcmpi function compares two strings irrespective of case.
2.8 For loop allows a statement or compound statement to be executed at least once.
2.9 The fread function reads formatted data from a stream.
2.10 In 'C', unsigned int can have maximum range of values between 0 to 65535.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 Group of related data items	A. ?:
3.2 A file not used for text processing	B. Recursion
3.3 Reserved word	C. Structure
3.4 A variable that keeps its value even after program exits the block in which it is declared	D. typecast
3.5 A data type used for saving storage area	E. keyword
3.6 The process by which function calls itself	F. Union
3.7 An operation in which value of one type converted into value of different type	G. Binary file
3.8 An operator expressed in three part expression	H. Call by reference
3.9 A sequence of bytes flowing into or out of program	I. Automatic variable
3.10 An external source file that contains declarations and definitions	J. Executable file
	K. Type checking
	L. Static variable
	M. stream
	N. Header file

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a) Union	(b) 4	(c) float
(d) indirection	(e) NULL	(f) semicolon
(g) void	(h) header file	(i) function
(j) 2	(k) pragma	(l) 5
(m) integers	(n) macro	(o) recursion

4.1 The expression 11%3 evaluates to _____.
4.2 In a 32 bit system float data type will occupy _____ bytes.
4.3 The closing brace of a structure is followed by _____.
4.4 A function that does not return anything has return type _____.
4.5 A(n) _____ cannot be passed to a function as an argument.
4.6 An instruction to the compiler to perform an action at compile time is called _____.

4.7 _____ is an example of derived data type.

4.8 Enumerations can be internally treated as _____.

4.9 Accessing a data object through a pointer rather than directly by name is called _____.

4.10 An identifier defined in a #define preprocessor directive to represent another series of character is called _____.

PART TWO

(Answer Any FOUR Questions)

5. (a) Develop a flowchart and then write a C program to display all prime numbers less than the number entered by the user.

(b) Explain the difference between an array, structure and an enumerated data type. **(10+5)**

6. Write an algorithm and then develop a program to evaluate the roots of a quadratic equation. Define and use a function cal_roots() to calculate the roots such that roots are also available in calling function i.e., use pointers. **(15)**

7. Develop a flowchart and then write a C program to find the occurrence (single or multiple) of a substring in a given string. The substring and string are entered by the user. Also point out the location at which the substring occurs. **(15)**

8. (a) Explain the difference between parameter passing mechanism "call by value" and "call by reference". Which is more efficient and why?

(b) Develop a flowchart and logic to implement the stack data structure using link list. **(5+10)**

9. (a) Draw a flowchart and then write a C program to enter the roll number and marks of any three subjects of few students from the keyboard and write to a file.

(b) It is said that "C is a middle level language and is good for system level programming." Describe three facilities available in 'C' which support this statement. **(10+5)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(July 2006)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 In a for loop with a multi statement loop body, semicolons should appear following:
- (a) the for statement itself
- (b) the crossing brace in the multiple statement loop body
- (c) each statement within the loop body and the test expression
- (d) each statement within the loop only

1.2 When we execute *x*++; the value of the expression *x*++:
- (a) is equal to the original value of *x*
- (b) is one more than the original value of *x*
- (c) is *x* times more than the original value of *x*
- (d) none of the above

1.3 An array's name is a:
- (a) pointer constant
- (b) pointer variable
- (c) variable name
- (d) none of the above

1.4 What is printed?

```
for (i=1;i<=5;)
i++;
printf ("%d",i)
```

- (a) 23456
- (b) 12345
- (c) 123456
- (d) error

1.5 What will assign in s, when we use the following pair of statements in c-program:

```
char *s;
s="my car color is : white";
```

(a) first character of the string constant
(b) complete string
(c) address of the string storage
(d) is a logical error

1.6 C uses pointers explicitly with:

(a) arrays (b) structures
(c) functions (d) all of the above

1.7 The values of the following storage classes are initialized by the compiler

(a) auto and extern (b) register and static
(c) static and extern (d) auto and register

1.8 Consider the following declarations

```
union id {
   char color;
   int size;
}
struct {
   char country;
   int date;
   union id i;
} flag;
```

To assign a color to a flag, the correct statement would be

(a) flag.color='W'; (b) flag.i.color='W';
(c) flag.color='White'; (d) flag.i.color='White';

1.9 Which of the following is true for the switch statement:

```
switch (var)
{
};
```

(a) can be used when only one variable is tested
(b) the variable must be an integral type
(c) each possible value of the variable can control a single branch
(d) all of the above

1.10 Enumeration is:

(a) a list of strings (b) a set of numbers
(c) a set of legal values possible (d) none of the above

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 An ampersand (&) is required before each variable name is printf.

2.2 It is an error to place the pound(#) sign of a preprocessor control line in any column except column 1.
2.3 The goto statement is a branching statement in 'C' programming.
2.4 An expression with the star operator, such as *ptr. cannot occur on the left-hand side of an assignment statement.
2.5 An array's name by itself cannot occur as the left-hand side of an assignment statement.
2.6 Function calls cannot be nested.
2.7 Each function must have at least one return statement.
2.8 A linked list is a data structure, which is created by dynamic allocation of memory.
2.9 The declaration void function-name () indicates that function-name returns nothing to the calling program.
2.10 Member variables of two different structures may have the same name.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 a[i] can be written as	A. Unending loop if no break statement inside the body
3.2 Calloc()	B. p is a pointer to a function that returns integer
3.3 union	C. allocate and clear memory
3.4 for(;;)	D. *(a+i)
3.5 register variable	E. allocates memory but does not clear memory
3.6 int (*p)[10]	F. is a memory location that is used by several different variables, which may be of different type
3.7 int (*p) (void*,void*)	G. directives
3.8 do-while loop	H. Increase in speed of execution
3.9 #include, #define	I. p is pointer to an array of integers
3.10 typedef	J. p is function that returns pointer to integer
	K. This guarantees that the loop is executed at least once before continuing
	L. preprocessor
	M. can be used to create variables of new types

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a) array	(b) string	(c) typedef
(d) cast	(e) structure	(f) EOF
(g) void	(h) c functions	(i) gets
(j) c preprocessor	(k) getstr	(l) DEFINE
(m) &	(n) char	(o) malloc

4.1 The _____ operator is used to convert one data type to another.
4.2 If the pointer variable ptr holds the address of a char, the data type of *ptr will be _____.
4.3 File is defined with a(n) _____ statement.
4.4 Functions of type _____ do not return a value to the calling segment.
4.5 A(n) _____ is a collection of variables under a single name.
4.6 _____ reads a whole line of input into a string until a new line of EOF is encountered.
4.7 Expressions can be made equivalent to a single identifier using the preprocessor _____ command.
4.8 The _____ is a program that is executed before the source code is compiled.
4.9 The _____ operator is unary operator to find the value of a variable.
4.10 Dynamic allocation of memory for structure can be done with the help of the _____ function.

PART TWO

(Answer Any FOUR Questions)

5. (a) Write the C statements (all necessary statements) that open the file inf.dat for reading, and open the file outf.dat for writing.

(b) Write the C program to write
"Introduction to C-Programming"
to the file outf.dat.

(c) Write a C program that reads integers from the file scores.dat. After all the integers have been read, the program writes the sum of all the nonnegative integers to the video display. Assume that the file scores.dat contains at least one integer. **(5+4+6)**

6. (a) Write a 'C' program to calculate and display the monthly income of a salesperson corresponding to the value of monthly sales input in the scanf() function, let us consider the following commission schedule: (Note : use if-else statement)

Monthly sales	**Income**
Greater than or equal to Rs. 50,000	375 plus 16% of sales
Less than Rs. 50,000 but Greater than or equal to Rs. 40,000	350 plus 14% of sales
Less than Rs. 40,000 but Greater than or equal to Rs. 30,000	325 plus 12% of sales
Less than Rs. 30,000 but Greater than or equal to Rs. 20,000	300 plus 9% of sales
Less than Rs. 20,000 but Greater than or equal to Rs. 10,000	250 plus 5% of sales
Less than Rs. 10,000	200 plus 3% of sales

(b) What is printed after execution of each of the following c-programs?

1.
```
void main ()
{
float reals[5];
*(reals+1)=245.8;
*reals = *(reals +1);
printf("%f",reals[0]);
}
```

2.
```
void main()
{
int nums[3];
int *ptr = nums;
nums[0]=100;
nums[1]=1000;
nums[2]=10000;
printf("%d\n",++*ptr);
printf("%d",*ptr);
}
```

3.
```
void main()
{
int digit=0;
while (digit <=9)
printf("%d\n",digit++);
}
```

4.
```
void main()
{
int a=7,b=6;
fun1(a,b);
printf("\n a is %d b is %d", a,b);
}
int fun1(int c, int d)
{ int e;
e=c*d;
d=7*c;
printf(\n c is %d d is %d e is %d", c,d,e);
return;
}
```

(7+[2*4])

7. (a) Write a C function word_count() to count the number of words in a given string and then call in main().

(b) Write a C function print_upper() to prints its character argument in uppercase.

(c) Write a macro that clears an array to zero. **(7+4+4)**

8. (a) What is a structure? Define a structure that contains the following members:
   (i) An integer quantity called acct_no
   (ii) A character called acct_type
   (iii) A 40-element character array called name
   (iv) A floating-point quantity called balance
   (v) A structure variable called lastpayment, of type date: defined as an integer called month; an integer called day; an integer called year.
   (vi) Include the user_defined data type *account* within the definition.
   (vii) Include structure variable customer, which is 100-element array of structures called account.

   (b) What is pointer in C? How pointers and arrays are related? **(8+7)**

9. Write short notes on any three of the following:
   (a) Switch statement (give proper syntax and examples)
   (b) What do you mean by loop? How while-loop and do-loop differs?
   (c) What is C preprocessor? Explain any two C preprocessor commands with example.
   (d) Break and Continue statements. **(3*5)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(January 2006)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 The following is a program

```
Main( )
      {
         int x = 0;
         while (x<=10)
         for ( ; ; )
            if( ++ x%10 = = 0)
               break;
            printf ("x = %d", x);
      }
```

What will be the output of the above program?

(a) Will print $x = 10$
(b) Will give compilation error
(c) Will give runtime error
(d) Will print $x = 20$

1.2 Consider the following variable declaration

```
Union x{
      int i;
      float f;
```

```
        char c;
    }
```

If the size of *i*, *f* and *c* are 2 bytes, 4 bytes and 1 byte respectively, then the size of the variable *y* is:-

(a) 1 byte (b) 2 bytes
(c) 4 bytes (d) 7 bytes

1.3 Pick up the odd one out from the following

(a) $x = x - 1$ (b) $x - = 1$
(c) $x - -$ (d) $x = - 1$

1.4 What is the value of 'average' after the following program is executed?

```
main( )
   {
   int sum, index,
   float average;
   sum = 0;
   for ( ; ; ) {
      sum = sum + index;
      ++ index;
      if (sum > = 100) break;
         }
   average = sum / index;
   }
```

(a) 91/13 (b) 91/14
(c) 105/14 (d) 105/15

1.5 Suppose *i*, *j*, *k* are integer variables with values 1, 2, 3 respectively. What is the value of the following expression?

```
! (( j + k ) > ( i + 5 ))
```

(a) 6 (b) 5
(c) 1 (d) 0

1.6 If a = –11 and b = –3. What is the value of a % b?

(a) –3 (b) –2
(c) 2 (d) 3

1.7 If c is a variable initialized to 1, how many times will the following loop be executed?

```
while (( c > 0 && (c < 60))
   {
      c ++;
   }
```

(a) 61 (b) 60
(c) 59 (d) 1

1.8 Which one of the following describes correctly a static variable?

(a) This cannot be initialized.

(b) This is initialized once at the commencement of execution and cannot be changed at run time.
(c) This retains its value through the life of the program.
(d) This is same as an automatic variable but is placed at the head of a program.

1.9 What will be the output of the following program?

```
main( )
   {
   int a, *ptr, b, c;
   a = 25;
   ptr = &a;
   b = a + 30;
   c = *ptr;
   printf ("%d %d %d", a, b, c);
   }
```

(a) 25, 25, 25 (b) 25, 55, 25
(c) 25, 55, 25 (d) None of the above

1.10 If a = 0xaa and $b = a << 1$ then which of the following is true
(a) $b = a$ (b) $b = 2a$
(c) $a = 2b$ (d) $n = a - 1$

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 It is not possible to print the % character as the function printf treats % as the beginning of a conversion specification.
2.2 A structure can include one or more pointers as members.
2.3 It is not possible to have formatted input / output in 'C'.
2.4 It is not possible to have nested if - else statements in 'C'.
2.5 The increment operator ++ does not work with float variable.
2.6 *a is the same as a[ ] in a parameter declaration.
2.7 In 'C' programming language, strings are represented using an array.
2.8 Relational operators have higher precedence than arithmetic operators.
2.9 A structure cannot be a member of a union.
2.10 *p++ increments the content of the location pointed by p.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 An Operator in 'C' permits two different expression to appear in situations where only one expression is ordinarily be used.	A. The operator &&

3.2	Variables, internal to a function, come into existence when the function is called	B.	Static variables
3.3	No space allocated for storage of character during compilation time	C.	Global variables
3.4	p is pointer to a function that returns a pointer to integer	D.	The comma operator (,)
3.5	Self-referencing structure	E.	int *p[10]
3.6	Accomplishing indirection with pointer to structure	F.	automatic variable
3.7	Random access in the file; file specified through file descriptor	G.	Useful for link-list implementation
3.8	Returns initialized storage in run-time	H.	seek
3.9	Variables can be defined in 'C' which occupies less space than character variables	I.	fseek
3.10	Valid mode for opening a file; permits read and write	J.	calloc
		K.	w+
		L.	malloc
		M.	r+
		N.	bit-fields
		O.	arrow operator
		P.	int(*pc))
		Q.	char *s

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a)	Change	(b)	trinary	(c)	-operator
(d)	Automatic	(e)	Variable	(f)	"&" operator
(g)	Static	(h)	Dynamically	(i)	Externally
(j)	void	(k)	A character	(l)	An integer
(m)	EOF	(n)	fclose	(o)	True

4.1 printf( ) function uses _____ number of arguments.
4.2 _____ can be used as both binary and unary operators.
4.3 _____ Link lists can be created _____.
4.4 Loop invariants are assertions that remain _____ before and after execution of loops.
4.5 File descriptor is _____.
4.6 Pointer arguments enable a function to access and _____ objects defined in the calling routine.
4.7 The function getchar( ) returns _____ when there is no more input character.
4.8 Any pointer can be cast to _____ without loss of information.
4.9 To prevent the use of functions across different files, _____ storage class is used.
4.10 ? : is _____ operator.

PART TWO
(Answer Any FOUR Questions)

5. (a) Discuss with the help of examples the action of break statement and the continue statement.
(b) Does the null statement have any uses besides indication that the body of a loop is empty? Explain.
(c) What is the purpose of the \? Escape sequence? **(8+4+3)**

6. (a)If it legal to put a function declaration inside the body of another function? If yes, give an example.
(b) Is it legal for a function *f*1 to call *f*2, which then calls *f*1? Justify your answer.
(c) Write a 'C' function that returns the k-th digit from the right in the positive integer *n*. For example, digit (829, 3) returns 89. If *k* is greater than the number of digits in *n* then the function is to return -1. Include appropriate documentation in your program. **(4+2+9)**

7. (a) If a pointer is an address, what does the expression like *p* + *j* mean?
(b) Is i[a] same as a[i]? Justify your answer.
(c) Write the following function:

```
Bool search (int a[ ], int n, int x);
```

Where *a* is an array to be searched, *n* is the number of elements in the array, and x is the search key. "search" should return TRUE if *x* matches some element of *a*, FALSE if it doesn't. Use pointer arithmetic to visit array elements. Include appropriate documentation in your program. **(4+2+9)**

8. (a) Develop an algorithm to do the following:
Read an array of 20 elements and then send all negative elements of the array to the end without altering the original sequence.
(b) Draw a flow chart and then write a 'C' program to generate first 15 members of the following sequence.
1, 3, 4, 7, 11, 18, 29, . . . **(5+10)**

9. Develop a flowchart and then write a program for analyzing a line of text stored in a file by examining each of the characters and displaying into which of several different categories vowels, constants, digits, white spaces it fails. Count of the number of vowels, consonants, digits and white space characters. Include an appropriate documentation in your program. **(15)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(July 2005)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 Consider the following code segment

```
Main( ){
   char s[100];
   scanf("%s", s);
   printf("%3s", s);}
```

If "India is Great" is entered upon the execution of the program for *s*, then that will be the output of the above code segment?

(a) India is Great
(b) Ind
(c) India
(d) The compiler gives the error that "*%3s*" is not valid format string in printf.

1.2 Consider the following code segment:

```
char s[ ] = "abc";
char *p = s;
```

Which of the following is not a valid statement to access 'b' in string s?

(a) *(++s) (b) *(++p)
(c) s[1] (d) *(p+1)

1.3 What will be the output of the following code segment?

```
main( ){
   int j = 1, k = 2;
```

```
            if (j & k)
               printf(" Successful ");
            else
               printf(" Unsuccessful ");}
```

(a) Successful (b) Unsuccessful
(c) Successful Unsuccessful (d) Unsuccessful Successful

1.4 What will be the output of the following code segment?

```
         main( ){
            int j = 1, k = 2;
            if (j = = 1)
               printf("%d", j);
            else;
               printf("%d", k); }
```

(a) 1 (b) 2
(c) 12 (d) 21

1.5 What will be the output of the following code segment?

```
      main( ) {
      int n = 10, d = 0;
      if (d ! = 0 && n / d > 1)
         printf (" Successful " );
      else
         printf (" Unsuccessful " ); }
```

(a) Successful
(b) Unsuccessful
(c) Compile time error of division by zero will be generated.
(d) Run time error of division by zero will be generated.

1.6 What will be the output of the following code segment?

```
         Main( ) {
            char s[10];
            strcpy(s, "abc");
            printf("%d %d", strlen(s), sizeof(s)); }
```

(a) 3 10 (b) 3 3
(c) 10 10 (d) 10 3

1.7 Which of the following is true for the following statement?

```
      NurseryLand.Nursery.Students = 10;
```

(a) The structure Students is nested within the structure Nursery.
(b) The structure NurseryLand is nested within the structure Nursery.
(c) The structure Nursery is nested within the structure students.
(d) The structure Nursery is nested within the structure NurseryLand.

1.8 Which of the following statements is the odd one out?

(a) j = j + 1; (b) j++;
(c) j += 1; (d) j =+ 1;

1.9 The default return type of main( ) is
(a) int (b) void
(c) char (d) float

1.10 The scope of a variable of automatic storage class is limited to the
(a) Program in which it is defined
(b) Block in which it is defined
(c) Loop in which it is defined
(d) Function in which it is defined

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 Only numeric constants can be #defined.
2.2 The default group may appear anywhere within the switch statement.
2.3 Like nested structures, functions can also be nested.
2.4 In call by reference, a change in the value of arguments by the called function is reflected in the calling function.
2.5 The C computer generates an error if the subscript used for an array exceeds the size of the array.
2.6 All members of a structure are allocated contiguous memory locations.
2.7 The statement FILE *fp means the variable fp is a pointer pointing to the actual file on the disk.
2.8 To pass the command line arguments, only argc and argv can be used as the format argument names.
2.9 The file stdin has to be explicitly opened before it can be used.
2.10 The ++ operator is a binary operator.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 Multiple initializations in a for loop	A. goto
3.2 One's Complement	B. register
3.3 Retaining values	C. int *p[10];
3.4 Fast access	D. ;
3.5 Array of pointers	E. static
3.6 Null terminated array of characters	F. gotoxy
3.7 Excape character	G. !
3.8 Moving read/ write pointer	H. structure
3.9 Changing data type	I. array
3.10 Unconditional jump	J. ftell
	K. extern

L. int (*p)(10;]
M. String
N. /
O. auto
P. type casting
Q. fseek
R. \
S. typedef
T. ~
U. ,

4. Each sentence below has a blank space to fit one of the words or phrases in the list below. Enter your choice in the answer sheet provided following instructions therein. (1×10)

(a)	int	(b)	Right to left	(c)	for
(d)	library function	(e)	fread	(f)	do..while
(g)	column major	(h)	&	(i)	char
(j)	structure	(k)	operator	(l)	fwrite
(m)	call by reference	(n)	row major	(o)	division
(p)	left to right	(q)	*	(r)	fclose
(s)	call by value	(t)	multiplication	(u)	fopen
(v)	–>	(w)	linked list		

4.1 The sizeof is a(n) _____.
4.2 In C, the arguments are passed from _____ to the called function.
4.3 To return more than one value from a function use _____.
4.4 An array is stored in _____ order in the memory.
4.5 The operator _____ is used exclusively with pointers.
4.6 Internally, the enumerators are stored as _____.
4.7 The function _____ must appear in the programs manipulating disk files.
4.8 Right shifting an unsigned integer is equivalent to its _____ by 2.
4.9 The body of a _____ loop might not be executed even once.
4.10 The self-referential structures are used to implement.

PART TWO

(Answer Any FOUR Questions)

5. Write a 'C' program to copy one file to another such that the first character of every word is removed from it and appended at its end. Assume that the maximum size of each word is 10 characters and each word is separated either by newline(s) or space(s). For example, if "Test String" is a line in the source file then "estT tringS" is copied in the target file. **(15)**

6. (a) Write a function to sort the characters of the string passed to it as argument.
(b) Write a function to remove all blank spaces in the string passed to it as argument. **(9+6)**

7. (a) Write a function, my_atoi, similar to the library function atoi that returns the numeric value corresponding to the string passed to it as an argument.

(b) Write a function to display the product of two matrices passed to it. The display must be in the matrix form itself. **(8+7)**

8. (a) Write a function to find which of the two unsigned numbers passed to it is greater using the bitwise operators. The function should return 1, if first is greater, –1 if second is greater and 0 if they are equal.

(b) Write a function to generate a triangle as shown below: (for $n = 4$)

```
      A
    A B A
  A B C B A
A B C D C B A
```

The number n is passed as an argument to the function. **(8+7)**

9. (a) Define a structure of a node of a linked list having an integer data member.

(b) Use the above structure and write the function for the following:

(i) a function to merge two sorted linked lists. The function accepts pointers to the head of sorted linked lists and returns the pointer to the head of the merged linked list.

(ii) a function to find the maximum and minimum of the data in the linked list. The function accepts a pointer to the head of the linked list, and returns the maximum and minimum values found in the list. **(2+[8+5])**

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Appendix I

STANDARD HEADER FILES AND LIBRARY FUNCTIONS

Header Files

Each C compiler provides a library of around 200 predefined functions and macros designed for use in C programs. These library functions make programming easier by providing the following:

(a) An interface to the operating system function (opening and closing files).
(b) Fast and efficient functions to perform common programming tasks (string manipulation), sparing the programmer the time and effort needed to write such functions.

For using these functions certain files need to be included in the programs which makes call to these functions.

These files are known as *header files* and they contain macro definitions, type definitions, and function declarations. These header files usually have an extension *.h*, as in *stdio.h*.

A few commonly required header files and the standard library functions that they support are listed here.

stdio.h

The header file *stdio.h* contains definitions of *constants*, *macros* and *types*, along with *function declaration* for *standard I/O functions.* These functions are listed below.

calloc	fgetchar	fputchar	getc	remove
fclose	fgetpos	fputs	getchar	rename
fcloseall	fgets	fread	gets	rewind
feof	flushall	fscanf	printf	scanf
ferror	fopen	fseek	putc	ungetc
fflush	fprintf	ftell	putchar	puts
fgetc	fputc	fwrite		

ctype.h

The *ctype.h* header file defines macros and constants and declares a global array used in character classification. The macros defined in *ctype.h* are listed below.

isalnum	isdigit	ispunct	toascii	_toupper
isalpha	isgraph	isspace	tolower	
isascii	islower	isupper	toupper	
iscntrl	isprint	isxdigit	_tolower	

string.h

The *string.h* header file declares the string manipulation functions, as listed below.

memccpy	memmove	strcmpi	strlwr	strchr
memchr	memset	strcpy	strncat	strrev
memcmp	stract	strdup	strncmp	strset
memcpy	strchr	stricmp	strncpy	strstr
memicmp	strcmp	strlen	strnicmp	strupr

math.h

The header file *math.h* contains function declarations for all floating point math routines as listed below.

abs	atof	cosh	log	sinh
acos	cabs	exp	log10	sqrt
asin	ceil	fabs	pow	tan
atan	cos	floor	sin	tanh

stdlib.h

The *stdlib.h* header file contains function declarations for the following functions.

abort	bsearch	itoa	rand	system
abs	calloc	labs	realloc	tolower
atof	exit	malloc	srand	toupper
atoi	free	perror	strtod	
atol	gcvt	qsort	strtol	

stdarg.h

The header file *stdarg.h* defines macros that allow you to access arguments in functions which are passed variable number of arguments, such as *vprintf().* These macros are defined to be machine independent and portable. These macros are listed below.

va_arg	va_end	va_start

time.h

The time.h header file declares the following time related functions.

asctime	difftime	clock	gmtime	ctime
time				

Library Functions

There is a vast collection of functions some of them are grouped together and listed below.

String Functions

stpcpy	
Function	Copies one string into another.
Syntax	char *stpcpy(char **dest,* const char **src);*
Prototype in	string.h
Remarks	*stpcpy* copies the string *src* to *dest,* stopping after the terminating null character has been reached.
Return value	*stpcpy* returns *dest + strlen(src).*
Portability	*stpcpy* is available on UNIX systems.

strcat	
Function	Appends one string to another.
Syntax	char *strcat(char **dest,* const char **src);*
Prototype in	string.h
Remarks	*strcat* appends a copy of *src* to the end of *dest*. The length of the resulting string is *strlen(dest) + strlen(src).*
Return value	*strcat* returns a pointer to the concatenated strings.
Portability	*strcat* is available on UNIX systems and is compatible with ANSI C. It is defined in K&R C.

strchr	
Function	Scans a string for the first occurrence of a given character.
Syntax	char *strchr(const char **s,* int *c);*
Prototype in	string.h
Remarks	*strchr* scans a string in the forward direction, looking for a specific character. *strchr* finds the first occurrence of the character *c* in the string*s*. The null-terminator is considered to be the part of the string, so that, for example, *strchr(strs, 0)* returns a pointer to the terminating null character of the string *strs.*
Return value	*strchr* returns a pointer to the first occurrence of the character *c* in *s*; if *c* does not occur in *s, strchr* returns null.
Portability	*strchr* is available on UNIX systems and is compatible with ANSI C.

strcmp	
Function	Compares one string to another.
Syntax	int strcmp(const char **s1,* const char **s2);*
Prototype in	string.h
Remarks	*strcmp* performs an unsigned comparison of *s1, s2* starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

Return value	*strcmp* returns a value that is < 0 if *s1* is less than *s2* = = 0 if *s1* is the same as *s2* > 0 if *s1* is greater than *s2*
Portability	*strcmp* is available on UNIX systems and is compatible with ANSI C.

strcmpi

Function	Compares one string to another, without case sensitivity.
Syntax	int strcmpi(const char **s1,* const char **s2*);
Prototype in	string.h
Remarks	*strcmpi* performs an unsigned comparison of *s1, s2* without case sensitivity (same as *stricmp*-implemented as a macro). It returns value (<0, 0, or >0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it). The routine *strcmpi* is the same, respectively, as stricmp.*strcmpi* is implemented via macro in *string.h* and translates calls from *strcmpi* to *stricmp.* Therefore, in order to use *strcmpi,* you must *#include* the header file *string.h* for the macro to be available. This macro is provided for compatibility with other C compilers.
Return value	*strcmpi* returns an *int* value that is < 0 if *s1* is less than *s2* = =0 if *s1* is the same as *s2* > 0 if *s1* is greater than *s2*

strcpy

Function	Copies one string into another.
Syntax	char* strcpy(char **dest,* const char **src);*
Prototype in	string.h
Remarks	copies string *src* to *dest,* stopping after the terminating null character has been moved.
Return value	*strcpy* returns *dest.*
Protability	*strcpy* is available on UNIX systems and is compatible with ANSI C.

strlen

Function	Calculates the length of a string.
Syntax	size_t strlen(const char **s);*
Prototype in	string.h
Remarks	*strlen* calculates the length of s.
Return value	*strlen* returns the number of characters in *s,* not counting the null-terminating character.
Portability	*strlen* is available on UNIX systems and is compatible with ANSI C.

strncmpi

Function	Compares a portion of one string to a portion of another, without case sensitivity.

Syntax	int strncmpi(const char *s1, const char *s2 size_tn);
Prototype in	string.h
Remarks	*strncmpi* performs a signed comparison of *s1, s2* for a maximum length of *n* bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until *n* characters have been examined. The comparison is not case sensitive. (*strncmpi* is the same as *strnicmp*-implemented as a macro). It returns a value (<0, 0, or >0) based on the result of comparing *s1* (or part of it) to *s2* (or part of it). The routines *strncmp* and *strncmpi* are the same; *strncmpi* is implemented via a macro in *string.h* that translates calls from *strncmpi* to *strnicmp.* you must include the header file *string.h* for the macro to be available. This macro is provided for compatibility with other C compilers.
Return value	*strnicmp* returns an *int* value that is < 0 if *s1* is less than *s2* = =0 if *s1* is the same as *s2* > 0 if *s1* is greater than *s2*

strrev

Function	Reverses a string.
Syntax	char *strrev(char *s);
Prototype in	string.h
Remarks	*strrev* changes all characters in a string to reverse order, except the terminating null character. For example, it would change string/0 to gnirts/0.
Return value	*strrev* returns a pointer to the reversed string. There is no error return.

Some other string manipulation functions and their brief description are as given below :

strcspn	:	Scans a string for initial segment not containing any subset of a given set of characters.
strdup	:	Copies a string into a newly-created location.
strerror	:	Returns a pointer to an error message string.
stricmp	:	Compares one string to another, without case sensitivity.
strlwr	:	Converts uppercase letters in a string to lowercase.
strncat	:	Appends a portion of one string to another.
strncmp	:	Compares a portion of one string to a portion of another.
strncmpi	:	Compares a portion of one string to a portion of another; without case sensitivity.
strncpy	:	Copies a given number of bytes from one string into another, truncating or padding as necessary.
strnicmp	:	Compares a portion of one string to a portion of another, without case sensitivity.
strnset	:	Sets a specified number of characters in a string to a given character.

strpbrk	:	Scans a string for the first occurrence of any character from a given set.
strrchr	:	Scans a string for the last occurrence of a given character.
strset	:	Sets all character in a string to a given character.
strspn	:	Scans a string for the first segment that is a subset of a given set of characters.
strstr	:	Scans a string for the occurrence of a given substring.
strtod	:	Converts a string to a double value.
strtok	:	Searches one string for tokens, which are separated by delimiters defined in a second string.
strtol	:	Converts a string to a long value.
strtoul	:	Converts a string to an unsigned long in the given radix.
strupr	:	Converts lowercase letters in a string to uppercase.

Mathematical Functions

Several Functions are provided by C for mathematical functions some of them are listed below:

abs

Function	Returns the absolute value of an integer
Syntax	int abs(int *x*);
Prototype in	math.h, stdlib.h
Remarks	*abs* returns the absolute value of the integer argument *x*. If *abs* is called when stdlib.h has been included, it will be treated as a macro that expands to inline code. If you want to use the *abs* function instead of the macro, include *#undef abs* in your program, after the *#include <stdlib.h>*.
Return value	*abs* returns an integer in the range of 0 to 32, 767, with the exception that an argument of -32, 768 is returned as -32, 768.
Portability	*abs* is available on UNIX systems and is compatible with ANSI C.

acos

Function	Calculates the arc cosine.
Syntax	double acos(double *x*);
Prototype in	math.h
Remarks	*acos* returns the arc cosine of the input value. Arguments to *acos* must be in the range -1 to 1. Arguments outside that range will cause *acos* to return 0 and set *errno* to EDOM Domain error
Return value	*acos* returns a value in the range 0 to *pi*. Error-handling for this routine can be modified through the function *matherr*.
Portability	*acos* is available on UNIX systems and is compatible with ANSI C.

asin

Function	Calculates the arc sine.
Syntax	double asin(double *x*);

Prototype in	math.h
Remarks	*asin* returns the arc sine of the input value. Arguments to *asin* must be in the range -1 to 1. Arguments outside that range will cause *asin* to return 0 and set *errno* to EDOM Domain error.
Return value	*asin* returns a value in the range 0 to *pi/2*. Error-handling for this routine can be modified through the function *matherr*.
Portability	*asin* is available on UNIX systems and is compatible with ANSI C.

atan

Function	Calculates the arc tangent.
Syntax	double atan(double *x*);
Prototype in	math.h
Remarks	*atan* calculates the arc tangent of the input value.
Return value	*atan* returns a value in the range *-pi/2* to *pi/2*. Error-handling for this routine can be modified through the function *matherr*.
Portability	*atan* is available on UNIX systems and is compatible with ANSI C.

atof

Function	Converts a string to a floating-point number.
Syntax	double atof(const char **s*);
Prototype in	math.h, stdlib.h
Remarks	*atof* converts a string pointed to by *s* to double; this function recognizes the character representation of a floating-point number, made up of the following: • an optional string of tabs and spaces • an optional sign • a string of digits and an optional decimal point (the digit can be on both sides of the decimal point) • an optional *e* or *E* followed by an optional signed integer The characters must match this generic format : [ws] [sn] [ddd] [.] [ddd] [fmt] [sn] [ddd] *atof* also recognizes +INF and -INF for plus and minus infinity, and +NAN and- NAN for Not-a-Number. In this function, the first unrecognized character ends the conversion.
Return value	*atof* returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (double), the return value is 0. If there is an overflow, *atof* returns plus or minus HUGE_VAL, and *matherr* is not called.
Portability	*atof* is available on UNIX systems and is compatible with ANSI C.

ceil

Function	Rounds up.
Syntax	double ceil(double *x*);

Prototype in	math.h
Remarks	*ceil* finds the smallest integer not less than *x*.
Return value	*ceil* returns the integer found (as a double).
Portability	*ceil* is available on UNIX systems and is compatible with ANSI C.

cos

Function	Calculates the cosine.
Syntax	double cos(double *x*);
Prototype in	math.h
Remarks	*cos* returns the cosine of the input value. The angle is specified in radians.
Return value	*cos* returns a value in the range -1 to 1. Error-handling for this routine can be modified through the function *matherr*.
Portability	*cos* is available on UNIX systems and is compatible with ANSI C.

exp

Function	Calculates the exponential e to the x^{th} power.
Syntax	double exp(double *x*);
Prototype in	math.h
Remarks	*exp* calculates the exponential function e^x.
Return value	*exp* returns e^x. Sometimes the arguments passed to *exp* produce results that overflow or are incalculable. When the correct value overflows, *exp* returns the value HUGE_VAL. Results of excessively large magnitude can cause *errno* to be set to ERANGE Result out of range. On underflow, *exp* returns 0.0, and *errno* is not changed. Error-handling for *exp* can be modified through the function *matherr.*
Portability	*exp* is available on UNIX systems and is compatible with ANSI C.

fabs

Function	Returns the absolute value of a floating-point number.
Syntax	double fabs(double *x*);
Prototype in	math.h
Remarks	*fabs* calculates the absolute value of *x,* a double.
Return value	*fabs* returns the absolute value of *x*. There is no return on error.
Portability	*fabs* is available on UNIX systems and is compatible with ANSI C.

floor

Function	Rounds down.
Syntax	double floor(double *x*);
Prototype in	math.h
Remarks	*floor* finds the largest integer not greater than *x.*
Return value	*floor* returns the integer found (as a *double).*
Portability	*floor* is available on UNIX systems and is compatible with ANSI C.

log	
Function	Calculates the natural logarithm of *x*.
Syntax	double log(double *x*);
Prototype in	math.h
Remarks	*log* calculates the natural logarithm of *x*.
Return value	On success, *log* returns the calculated value *In(x).* If the argument *x* passed to *log* is less than or equal to 0, *errno* is set to EDOM Domain error. When this error occurs, *log* returns the value negative HUGE_VAL. Error-handling for *log* can be modified through the function *matherr.*
Portability	*log* is available on UNIX systems and is compatible with ANSI C.

log10	
Function	Calculates log_{10} (*x*).
Syntax	double log10(double *x*);
Prototype in	math.h
Remarks	*log10* calculates the base 10 logarithm of *x.*
Return value	On success, *log10* returns the calculated value log_{10} *(x).* If the argument *x* passed to *log10* is less than or equal to 0, *errno* is set to EDOM Domain error. When this error occurs, *log10* returns the value negative HUGE_VAL. Error-handling for *log10* can be modified through the function *matherr.*
Portability	*log10* is available on UNIX systems and is compatible with ANSI C.

pow	
Function	Calculates *x* to the power of *y*.
Syntax	double pow(double *x,* double *y*);
Prototype in	math.h
Remarks	*pow* calculates x^y.
Return value	On success, *pow* returns the calculated value x^y . Sometimes the arguments passed to *pow* produce results that overflow or are incalculable. When the correct value would overflow, *pow* returns the value HUGE_VAL. Results of excessively large magnitude can cause *errno* to be set to ERANGE Result out of range. *errno* is set to EDOM Domain error. If the argument *x* passed to *pow* is less than or equal to *0*, and *y* is not a whole number. When this error occurs, *pow* returns the value negative HUGE_VAL. If the argument *x* and *y* passed to *pow* are both 0, *pow* returns 1. Error-handling for *pow* can be modified through the function *matherr.*
Portability	*pow* is available on UNIX systems and is compatible with ANSI C.

sin	
Function	Calculates sine.
Syntax	double sin(double *x*);
Prototype in	math.h

Remarks	*sin* computes the sine of the input value. Angles are specified in radians. Error-handling for this routine can be modified through the function *matherr.*
Return value	*sin* returns the sine of the input value.
Portability	*sin* is available on UNIX systems and is compatible with ANSI C.

sqrt

Function	Calculates the positive square root of input value.
Syntax	double sqrt(double *x*);
Prototype in	math.h
Remarks	*sqrt* calculates the positive square root of the input value. Error-handling for *sqrt* can be modified through the function *matherr.*
Return value	On success, *sqrt* returns the value calculated, the positive square root of *x.* If *x* is negative, *errno* is set to EDOM Domain error.
Portability	*sqrt* is available on UNIX systems and is compatible with ANSI C.

tan

Function	Calculates the tangent.
Syntax	double tan(double *x*);
Prototype in	math.h
Remarks	*tan* calculates the tangent. Angles are specified in radians. Error-handling for this routine can be modified through the function *matherr*.
Return value	*tan* returns the tangent of *x,* and value for valid angles. For angles close to *pi/2* or *-pi/2, tan* returns 0 and *errno* is set to ERANGE Result out of range.
Portability	*tan* is available on UNIX systems and is compatible with ANSI C.

Some more mathematical routines supported by math.h are as under :

atan2	:	Calculates the arc tangent of *y/x.*
cabs	:	Absolute value of complex number.
cosh	:	Calculates the hyperbolic cosine.
div	:	Divides two integers, returning quotient and remainder.
fmod	:	Calculates *x* modulo *y,* the remainder of *x/y.*
frexp	:	Splits *double* number into mantissa and exponent.
hypot	:	Calculates hypotenuse of a right angle triangle.
ldexp	:	Calculates $x \times 2^{exp}$.
ldiv	:	Divides two longs, returns quotient and remainder.
matherr	:	User -modifiable math error handler.
modf	:	Splits *double* into integer and fraction parts.
poly	:	Generates a polynomial from arguments.
pow10	:	Calculates 10 to the power of *p.*
sinh	:	Calculates hyperbolic sine.
tanh	:	Calculates the hyperbolic tangent.

Some mathematical routines supported by *stdlib.h* are as under :

atoi	:	Converts a string to an integer.
atol	:	Converts a string to a long.
ecvt	:	Converts a floating-point number to a string.
fcvt	:	Converts a floating-point number to a string.
gcvt	:	Converts a floating-point number to a string.
itoa	:	Converts an integer to a string.
labs	:	Gives long absolute value.
ltoa	:	Converts a long to a string.
rand	:	Random number generator.
random		Random number generator.
randomize	:	Initializes random number generator.
srand	:	Initializes random-number generator.

Date and Time Functions

asctime

Function	Converts date and time.to ASCII
Syntax	char *asctime(const struct tm **tblock*);
Prototype in	time.h
Remarks	*asctime* converts a time stored as a structure in **tblock* to a 26-character string of the same form as the *ctime* string: *Sun Sep 16 01:03:52 1973\n\0*
Return value	*asctime* returns a pointer to the character string containing the date and time. This string is a static variable that is overwritten with each call to *asctime*.
Portability	*asctime* is available on UNIX systems and is compatible with ANSI C.

Example Program:

```
#include <stdio.h>

#include <time.h>

main()

{

          struct tm *tm_now;

          time_t secs_now;

          char *str_now;

          /* get time in seconds */
```

```
            time(&secs_now);

            /* make it a string */

            str_now = ctime(&secs_now);

            printf("\nThe number of seconds since Jan 1, 1970 is %ld\n",
secs_now);

            printf("\nIn other words, the current time is %s", str_now);

            /* make it a structure */

            tm_now = localtime(&secs_now);

            printf("\nFrom the structure: day %d-%02d-%02d-%02d
%02d:%02d:%02d",

            tm_now->tm_yday, tm_now->tm_mon, tm_now->tm_mday,
            tm_now->tm_year, tm_now->tm_hour, tm_now->tm_min,
            tm_now->tm_sec);

            /* from structure to string */

            str_now = asctime(tm_now);

            printf("\nOnce more, the current time is %s", str_now);

  }
```

Output: *The number of seconds since Jan 1, 1970 is 315594553.*
In other words, the current time is Tue Jan 01 12:09:12 1980
From the structure : day 0 00-01-80 12:09:13
Once more, the current time is Tue Jan 01 12:09:12 1980

clock

Function	Determines processor time
Syntax	clock_t clock(void);
Prototype in	time.h
Remarks	*clock* can be used to determine the time interval between two events. To determine the time in seconds, the value returned by *clock* should be divided by the value of the macro CLK_TCK.
Return value	The *clock* function returns the processor time elapsed since the beginning of the program invocation. If the processor time is not available or its value cannot be represented, the function returns the value-1.
Portability	*clock* is compatible with ANSI C.

Example Program:

```
#include <time.h>

#include <stdio.h>

void main()

{

          clock_t start, end;

          start = clock();    /*  Code to be timed goes here */

          end = clock();

          printf("\nThe time was: %f\n", (end - start) /CLK_TCK);

}
```

getdate

Function	Gets system date.
Syntax	void getdate(struct date **datep);*
Prototype in	dos.h
Remarks	*getdate* fills in the *date* structure (pointed to by *datep)* with the system's current date.
	The *date* structure is defined follows :

```
struct date  {

                int da_year; /* current year */

                char da_day; /* day of the month */

                char da_mon; /* month (1 = Jan) */

             };
```

Return value	None.
Portability	*getdate* is unique to DOS.

Example Program:

```
#include <stdio.h>

#include <dos.h>
```

```
main()

{

          struct date today;

          struct time now;

          getdate (&today);

          printf ("\nToday's date is %d%d%d\n", today.da_mon,
today.da_day, today.da_year);

          gettime (&now);

          printf ("\nThe time is %02d:%02d:02d.%02d\n", now.ti_hour,
now.ti_min, now.ti_sec, now.ti_hund);

}
```

Output: Today's date is 1/1/1980
The time is 17:08:22.42

gettime

Function	Gets system time.
Syntax	void gettime(struct time **timep*);
Prototype in	dos.h
Remarks	*gettime* fills in the *time* structure pointed to by *timep* with the system's current time.

The *time* structure is defined as follows:

```
struct time {

   unsigned char ti_min;  /* minutes */

   unsigned char ti_hour; /* hours */

   unsigned char ti_hund; /* hundredths of seconds */

   unsigned char ti_sec;  /* seconds */

};
```

Return value	None.
Portability	*gettime* is unique to DOS.

setdate

Function	Sets DOS date.
Syntax	void setdate(struct date **datep*);
Prototype in	dos.h
Remarks	*setdate* sets the system date (month, day and year) to that in the *date* structure pointed to by *datep.* The *date* structure is defined as follows :

```
struct date

{

   int da_year;         /* current year */

   char da_day;      /* day of the month */

   char da_mon;      /* month (1 = Jan) */  };
```

Return value	None.
Portability	*setdate* is unique to DOS.

settime

Function	Sets system time.
Syntax	void settime(struct time **timep*);
Prototype in	dos.h
Remarks	*settime* sets the system time to the values in the time structure pointed to by *timep*. The *time* structure is defined as follows :

```
struct time

{  unsigned char ti_min;  /* minutes */

   unsigned char ti_hour; /* hours */

   unsigned char ti_hund; /* hundredths of seconds */

   unsigned char ti_sec;  /* seconds */

};
```

Return value	None.
Portability	*settime* is unique to DOS.

time

Function	Gets time of day.
Syntax	time_t time(time_t **time);*

Prototype in	time.h
Remarks	*time* gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by *timer,* provided that *timer* is not a null pointer.
Return value	*time* returns the elapsed time is seconds, as described.
Portability	*time* is available on UNIX systems and is compatible with ANSI C.

Some other time and date routines are :

ctime (time.h)	:	Converts date and time to a string.
difftime (time.h)	:	Computes the difference between two times.
dostounix (dos.h)	:	Converts date and time to UNIX time format.
ftime (sys\timeb.h)	:	Stores current time in *timeb* structure.
gmtime (time.h)	:	Converts date and time to Greenwich Mean Time (GMT).
localtime (time.h)	:	Converts date and time to a structure.
stime (time.h)	:	Sets system date and time.
tzset (time.h)	:	Sets a value of global variable *daylight, timezone,* and *tzname.*
unixtodos (dos.h)	:	Converts date and time to DOS format.

Variable Argument List Functions

These routines are for use when accessing variable argument lists. (such as with vscanf, vprintf etc.)

va_...

Function	Implement a variable argument list.

Syntax	void va_start(va_list *param, lastfix*);
	type va_arg(va_list *param, type*);
	void va_end(va_list *param*);

Prototype in	stdarg.h
Remarks	Some C functions, such as *vfprintf* and *vprintf*, take variable argument lists in addition to taking a number of fixed (known) parameters. The *va_...* macros provide a portable way to access these argument lists.They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being passed. The header file stdarg.h declares one type (va_list), and three macros (*va_start, va_arg,* and *va_end*). ***va_list :*** array holds information needed by *va_arg* and *va_end*. When a called function takes a variable argument list, it declares a variable *param* of type *va_list*. ***va_start :*** This routine (implemented as a macro) sets *param* to point to the first of the variable arguments being passed to the funcion. *va_start* must be used before the first call to *va_arg* or *va_end va_start* takes two parameters: *param* and *lastfix.* (*param* is explained under *va_list* in the preceding paragraph; *lastfix* is the name of the last fixed parameter being passed to the called function.)

va_arg : This routine (also implemented as a macro) expands to an expression that has the same type and value as next argument being passed (one of the variable arguments). The variable *param* to *va_arg* should be the same *param* that *va_start* initialized. The first time va_arg is used, it returns the first argument in the list. Each successive time *va_arg* is used, it returns the next argument in the list. It does this by first de-referencing *param* to point to the following to the item. *va_arg* uses the *type* to both perform the de-reference and to locate the following item. Each successive time *va_arg* is invoked, it modifies *param* to point to the next argument in the list.

va_end : This macro helps the called function perform a normal return. *va_end* might modify *param* in such a way that it cannot be used unless *va_start* is re-called. *va_end* should be called after *va_arg* has read all the arguments; failure to do so might cause strange, undefined behavior in your program.

Return value — *va_start* and va_end return no values; *va_arg* returns the current argument in the list (the one that *param* is pointing to).

Portability — *va_arg, va_start,* and *va_end* are available on UNIX systems.

Example Programs:

1.

```
#include <stdio.h>

#include <stdarg.h>

void sum(char *msg, ...) /* calculate sum of a 0 terminated
list */

{

   int arg, total = 0;

   va_list ap;

   va_start(ap, msg);

   while ((arg = va_arg(ap, int)) != 0)

   {

      total += arg;

   }

   printf (msg, total);

}
```

```
main()
{
   sum ("\nThe total of 1+2+3+4 is %d\n", 1, 2, 3, 4, 0);
}
```

Output: *The total of 1+2+3+4 is 10*

2.

```
#include <stdio.h>
#include <stdarg.h>
void error(char *format, ...)
{
   va_list argptr;
   printf ("error: ");
   va_start (argptr, format);
   vprintf (format, argptr);
   va_end (argptr);
}
main()
{
   int  value = -1;
   error ("\nThis is just an error message.\n");
   error ("\nInvalid value %d encountered\n", value);
}
```

Output: *error :This is just an error message.*
error :Invalid value -1 encountered

Other variable argument list function are :

vfprintf (stdio.h) : Writes formatted output to a stream.

vfscanf (stdio.h)	:	Scans and formats input from a stream.
vprintf (stdarg.h	:	Write formatted output to *stdout.*
vscanf (stdarg.h	:	Scans and formats input from *stdin*.
vsprintf (stdarg.h)	:	Writes formatted output to a string
vsscanf (stdarg.h)	:	Scans and formats input from a stream.

Utility Functions

abort()	Abnormally terminates a process. Syntax: *void abort()*;
bsearch()	Binary search of an array. Syntax: *void bsearch (void *key, void *base, unsigned nelem, unsigned width, int (*fcmp) (void*, CONST void*));*
calloc()	Allocates main memory. Syntax:*void calloc (unsigned nitems, unsigned size);*
exit()	Terminates execution of a program. Syntax: *void exit (int status);*
free()	Frees allocated block. Syntax:*void free (void *block);*
malloc()	Allocates main memory. Syntax:*void malloc (unsigned size);*
perror()	Prints a system error message. Syntax:*void perror (char *s);*
qsort()	Sorts using the quicksort algorithm. Syntax:*void qsort (void *base, unsigned nelem, unsigned width, int (*fcmp) (void*, void*));*
realloc()	Reallocates the main memory. Syntax:*void *realloc (void * block, unsigned size);*
system()	Issues a DOS command. Syntax:*int system (char *command);*
tolower()	Translates characters to lowercase. Syntax:*int tolower (int ch);*
toupper()	Translates characters to uppercase. Syntax:*int toupper (int ch);*

Character Class Test Functions

isalnum()	Tests whether a character is an alphabet or a number. Syntax:i*salnum (int c);*
isalpha()	Character classification macro that returns nonzero if c is a letter (A-Z or a-z). Syntax:*int isalpha (int c);*

isascii()	Tests whether a character is an ascii (0 to 127) character. Syntax:*iasascii (int c);*
iscntrl()	Tests whether a character is a control character. Syntax:*iscntrl (int c);*
isdigit()	Character classification macro that returns nonzero value if c is a digit ('0'-'9'). Syntax:*int isdigit (int c);*
isgraph()	Character classification macro that returns nonzero if c is a printing character, space character is excluded. Syntax:*int isgraph (int c);*
islower()	Character classification macro that returns non zero value if c is a lower case alphabet and zero otherwise. Syntax: *int islower (int c);*
isprint()	Character classification macro that returns value is c is a printable character (ASCII 32 to 126) and non-zero otherwise. Syntax: *int isprint (int c);*
ispunct()	Character classification macro that returns non-zero value if c is a punctuation character *(iscntrl or isspace).* Syntax: *int ispunct (int c);*
isspace()	Character classification macro that returns non-zeroif c is a space, tab, carriage return, newline, vertical tab, formfeed. Syntax:int isspace (int c);
isupper()	Character classification macro that return non-zero if c is an uppercase letter (A-Z). Syntax:int isupper (int c);
isxdigit()	Character classification macro that returns non-zero if c is a hexadecimal digit (0-9, A-F, a-f). Syntax:int isxdigit (int c);
toascii()	Translates character to ASCII format. Syntax : int toascii (int ch);
tolower()	Translates character to uppercase. Syntax : int tolower (int ch);
toupper( )	Translates character to uppercase. Syntax : int toupper (int ch) ;
_tolower( )	Translates character tab lowercase. Syntax : int _tolower (int ch) ;
_toupper()	Translates character to uppercase. Syntax : int _touppter (int ch);

Appendix II

ASCII VALUES OF CHARACTERS

Note : The first 32 characters and the last character are control characters — they cannot be printed.

ASCII		ASCII		ASCII		ASCII	
Value	*Character*	*Value*	*Character*	*Value*	*Character*	*Value*	*Character*
000	NUL	018	DC2	036	$	054	6
001	SOH	019	DC3	037	%	055	7
002	STX	020	DC4	038	&	056	8
003	ETX	021	NAK	039	‘	057	9
004	EOT	022	SYN	040	(	058	:
005	ENQ	023	ETB	041	)	059	;
006	ACK	024	CAN	042	*	060	<
007	BEL	025	EM	043	+	061	=
008	BS	026	SUB	044	‘	062	>
009	HT	027	ESC	045	-	063	?
010	LF	028	FS	046	.	064	@
011	VT	029	GS	047	/	065	A
012	FF	030	RS	048	0	066	B
013	CR	031	US	049	1	067	C
014	SO	032	blank	050	2	068	D
015	SI	033	!	051	3	069	E
016	DLE	034	“	052	4	070	F
017	DC1	035	#	053	5	071	G

ASCII		ASCII		ASCII		ASCII	
Value	*Character*	*Value*	*Character*	*Value*	*Character*	*Value*	*Character*
072	H	086	V	100	d	114	r
073	I	087	W	101	e	115	s
074	J	088	X	102	f	116	t
075	K	089	Y	103	g	117	u
076	L	090	Z	104	h	118	v
077	M	091	[	105	i	119	w
078	N	092	\	106	j	120	x
079	O	093	]	107	k	121	y
080	P	094	↑	108	l	122	z
081	Q	095	-	109	m	123	{
082	R	096	‘	110	n	124	|
083	S	097	a	111	o	125	}
084	T	098	b	112	p	126	~
085	U	099	c	113	q	127	DEL

Model Question Papers

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(July 2007)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 Which of the following is a valid octal constant?

(a) 32 (b) 032

(c) 049 (d) 0x49

1.2 Which of the following switch statement is not a valid statement to print "RED" if a character variable 'color' has the value 'R' or 'r'?

(a) switch (color) { case 'R': case 'r'; printf("RED"); break; }

(b) switch (color) { case 'R': printf("RED"); break; case 'r': printf("RED"); break; }

(c) switch (toupper(color)) { case 'R': printf("RED"); break; }
(d) switch (color) { case 'R' || 'r': printf("RED"); break; }

1.3 What will be the output of the following code segment, if the function is called as larger (10,20)?

```
int larger (int x, int y){
   int max = x;
   if (max <y){
      max = y;
      return y;
   }
   else
   return x;
   printf ("Larger of %d and %d is %d", x,y, max);
}
```

(a) Program will not compile as the function has two return statements
(b) Program will not compile as no statement is allowed after return statement
(c) Larger of 10 and 20 is 20
(d) No output

1.4 Given the code segment:

```
char a[] = "abc", *p;
```

Which of the following assigns the starting address of the string "abc" to p?

(a) p = a; (b) p = &a;
(c) p = *a; (d) *p = a;

1.5 To read and write an existing file without overwriting, the following mode is used

(a) r (b) w
(c) r+ (d) w+

1.6 If an array is defined as static char a[10]; then the elements of *a* will be set to

(a) an undetermined value (b) zero
(c) blank character (d) character '~0'

1.7 For the code segment

```
struct DOB {int date, month, year;};
struct person {char name[30];
struct DOB birthdate;}p, *ptr=&p;
```

Which of the following is not a valid expression to access year of birth date?

(a) ptr -> birthdate.year (b) (*ptr).birthdate.year
(c) ptr.birthdate.year (d) p.birthdate.year

1.8 Which of the following statement is false for the statement?

```
main (int ac, char *av[])
```

(a) av is an array of pointers to strings
(b) av[0] represents the name of the program under execution
(c) the formal arguments names have to be argc and argv only
(d) the main function can return an integer to the calling function/program

1.9 What will be the output of the following?

```
main () {
   int a ='A';
   printf (("%d", a);
```

(a) 65 (b) A
(c) a
(d) the program will not compile as an integer variable is assigned a character constant

1.10 Consider the following code segment:

```
for (odd_sum=0, j=1; ***; j+=2)
odd_sum+=j;
```

In order to sum all the odd numbers between 1 to 100; which of the following statements cannot replace ***?

(a) j < = 99 (b) j < 99
(c) j < = 100 (d) j < 100

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 An escape sequence begins with a backward slash followed by an alphabetical character.
2.2 The for loop can be used only for the cases when the number of passes is known in advance.
2.3 Let an array **arr** be a member of a structure S. If S is passed to a function, test as test(S) then the changes in arr, if any, by test will not be reflected in the calling function.
2.4 In a recursive function with local variables, a different set of local variables with the same name are created during each call.
2.5 Two enumeration constants defined in an enumeration definition can have same integral value.
2.6 The sizeof operator can only be used with variables that are allocated space using malloc() function.
2.7 If u is a union variable, then using isalpha(u) it is possible to know whether u is storing an alphabet or not.
2.8 If a file is created with fwrite() function, then it is valid to read it using fscanf() function.
2.9 NULL is a keyword in C.
2.10 A function name can be passed as an argument to another function.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 Self referential data structure	A. Arrays
3.2 Converting to a different data type	B. Flowchart
3.3 Creating new data type	C. while (0)
3.4 Removing repetitive coding	D. register variables

3.5	Infinite loop	E.	recursion
3.6	Pictorial representation of logic	F.	Function
3.7	Request to compiler	G.	Declaration
3.8	Defining constants	H.	typedef
3.9	Global variables	I.	#define
3.10	Space allocation to variables	J.	static
		K.	algorithm
		L.	linked lists
		M.	typecasting
		N.	Definition
		O.	External variables
		P.	for (;;)

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a)	actual	(b)	4	(c)	character
(d)	false	(e)	self-referencial structure	(f)	formal
(g)	stream oriented	(h)	heterogeneous	(i)	declaration
(j)	1	(k)	5	(l)	text
(m)	unknown	(n)	non linear	(o)	union
(p)	pointer	(q)	0	(r)	linear
(s)	sparse	(t)	true	(u)	unformatted
(v)	definition	(w)	integer	(x)	linked list

4.1 The data structure with most of the entries as 0 (zero) is a (n) _____.
4.2 A null terminated array of _____ is a string.
4.3 A constant can be valid _____ argument to a function.
4.4 The statement struct point {int x,y;} is a structure _____.
4.5 Linked list is a _____ data structure.
4.6 The expression pv + 3 is valid but not pv*3 if pv is a(n) _____ variable.
4.7 A(n) _____ file can be created with specially written program only.
4.8 The size of an array defined as char color[] ='Blue'; is _____.
4.9 In expression ((j+k>10)|| (n>-3)), (n>-3) will be evaluated if (j+k > 10) is _____.
4.10 The loop do {...} while (0); will be executed _____ times.

PART TWO

(Answer Any FOUR Questions)

5. (a) Write the C program to compute the following series:
$1 - x + x^2/2 - x^3/6 + x^4/24 + \ + (-1)^n x^n/n!$
Where n and x is to be accepted by the user.

(b) Develop a flowchart and then write a 'C' program to sort strings passed to the program through the command line arguments. Also display the sorted strings. **(6+9)**

6. (a) Define a structure to store roll_no, name and marks of a student.
(b) Using the structure of Q6. (a), above, write a C program to create a file "student.data". There must be one record for every student in the file. Accept the data from the user.
(c) Using the "student.dat" of Q6. (b), above, write a C program to search for the details of the student whose name is entered by the user. **(3+6+6)**

7. (a) Write a 'C' function to reverse a singly linked list by traversing it only once.
(b) Write a 'C' function to remove those nodes of a singly linked list which have duplicate dat(a) Assume that the linked list is already in ascending order. **(7+8)**

8. (a) What do you understand by loading and linking of a program?
(b) Write a 'C' function to generate the following figure for $n = 7$.

```
        1
      1 3
    1 3 5
  1 3 5 7
    1 3 5
      1 3
        1
```

The value of n is passed to the function as an argument. Print the triangle only if n is odd otherwise print an error message.

(c) Write a ‘C’ function to arrange the elements of an integer array in such a way that all the negative elements are before the positive elements. The array is passed to it as an argument. **(3+6+6)**

9. (a) Write a recursive function in ‘C’ to count the number of nodes in a singly linked list.
(b) Develop a flowchart and then write a ‘C’ program to add two very large positive integers using arrays. The maximum number of digits in a number can be 15. **(5+10)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(January 2007)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART TWO

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 The && and || operators
- (a) compare two numeric values
- (b) combine two numeric values
- (c) compare two boolean values
- (d) combine two boolean values

1.2 The break statement causes an exit
- (a) only from innermost loop
- (b) only from innermost switch
- (c) from the innermost loop or switch
- (d) none of the above

1.3 Assuming var1 has value 20. What will following code print?

```
printf("%d%d\n", var1--,++var1);
```

(a) 20 20 (b) 19 20
(c) 20 21 (d) 21 22

1.4 When accessing a structure member, the identifier to the left of the dot operator is the name of
- (a) a structure member (b) a structure tag
- (c) a structure variable (d) the keyword struct

1.5 A static automatic variable is used to
(a) make a variable visible to several functions
(b) retain a value when a function is not executing
(c) conserve memory when a function is not executing
(d) none of the above

1.6 Which of the following directive creates functions like macros?
(a) # include (b) # define
(c) # undef (d) # ifdef

1.7 Which format specification is used in printf statement to print hexadecimal format
(a) %i (b) %c
(c) %x (d) %u

1.8 What will be the output of the following program:

```
main()
{
   int val =500;
   int *ptr = &val;
   int **ptr1 = &ptr;
   printf ("val = %d", **ptr1);
}
```

(a) 500 (b) address of ptr
(c) contents of ptr (d) none of the above

1.9 Size of operator returns the size in bytes of
(a) identifier (b) type
(c) identifier or type (d) array

1.10 The value of variable x after executing the following code will be:

```
val = -200;
x=(val >=0) ? val : -val
```

(a) 0 (b) 200
(c) –200 (d) 1

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 The #undef directive removes a name previously defined with #define directive.
2.2 The 'C' program can have only one command line argument.
2.3 The goto statement causes control to go to a function.
2.4 If you don't use a return type in the function declaration, the compiler assumes that the function does not return anything.
2.5 An array element is accessed using the dot operator.
2.6 Continue statement skips all subsequent statements in the loop body and triggers the next iteration for the loop.

2.7 The strcmpi function compares two strings irrespective of case.
2.8 For loop allows a statement or compound statement to be executed at least once.
2.9 The fread function reads formatted data from a stream.
2.10 In 'C', unsigned int can have maximum range of values between 0 to 65535.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 Group of related data items	A. ?:
3.2 A file not used for text processing	B. Recursion
3.3 Reserved word	C. Structure
3.4 A variable that keeps its value even after program exits the block in which it is declared	D. typecast
3.5 A data type used for saving storage area	E. keyword
3.6 The process by which function calls itself	F. Union
3.7 An operation in which value of one type converted into value of different type	G. Binary file
3.8 An operator expressed in three part expression	H. Call by reference
3.9 A sequence of bytes flowing into or out of program	I. Automatic variable
3.10 An external source file that contains declarations and definitions	J. Executable file
	K. Type checking
	L. Static variable
	M. stream
	N. Header file

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a) Union	(b) 4	(c) float
(d) indirection	(e) NULL	(f) semicolon
(g) void	(h) header file	(i) function
(j) 2	(k) pragma	(l) 5
(m) integers	(n) macro	(o) recursion

4.1 The expression 11%3 evaluates to _____.
4.2 In a 32 bit system float data type will occupy _____ bytes.
4.3 The closing brace of a structure is followed by _____.
4.4 A function that does not return anything has return type _____.
4.5 A(n) _____ cannot be passed to a function as an argument.
4.6 An instruction to the compiler to perform an action at compile time is called _____.

4.7 _____ is an example of derived data type.

4.8 Enumerations can be internally treated as _____.

4.9 Accessing a data object through a pointer rather than directly by name is called _____.

4.10 An identifier defined in a #define preprocessor directive to represent another series of character is called _____.

PART TWO

(Answer Any FOUR Questions)

5. (a) Develop a flowchart and then write a C program to display all prime numbers less than the number entered by the user.

(b) Explain the difference between an array, structure and an enumerated data type. **(10+5)**

6. Write an algorithm and then develop a program to evaluate the roots of a quadratic equation. Define and use a function cal_roots() to calculate the roots such that roots are also available in calling function i.e., use pointers. **(15)**

7. Develop a flowchart and then write a C program to find the occurrence (single or multiple) of a substring in a given string. The substring and string are entered by the user. Also point out the location at which the substring occurs. **(15)**

8. (a) Explain the difference between parameter passing mechanism "call by value" and "call by reference". Which is more efficient and why?

(b) Develop a flowchart and logic to implement the stack data structure using link list. **(5+10)**

9. (a) Draw a flowchart and then write a C program to enter the roll number and marks of any three subjects of few students from the keyboard and write to a file.

(b) It is said that "C is a middle level language and is good for system level programming." Describe three facilities available in 'C' which support this statement. **(10+5)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(July 2006)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 In a for loop with a multi statement loop body, semicolons should appear following:
- (a) the for statement itself
- (b) the crossing brace in the multiple statement loop body
- (c) each statement within the loop body and the test expression
- (d) each statement within the loop only

1.2 When we execute *x*++; the value of the expression *x*++:
- (a) is equal to the original value of *x*
- (b) is one more than the original value of *x*
- (c) is *x* times more than the original value of *x*
- (d) none of the above

1.3 An array's name is a:
- (a) pointer constant
- (b) pointer variable
- (c) variable name
- (d) none of the above

1.4 What is printed?

```
for (i=1;i<=5;)
i++;
printf ("%d",i)
```

- (a) 23456
- (b) 12345
- (c) 123456
- (d) error

1.5 What will assign in s, when we use the following pair of statements in c-program:

```
char *s;
s="my car color is : white";
```

(a) first character of the string constant
(b) complete string
(c) address of the string storage
(d) is a logical error

1.6 C uses pointers explicitly with:

(a) arrays (b) structures
(c) functions (d) all of the above

1.7 The values of the following storage classes are initialized by the compiler

(a) auto and extern (b) register and static
(c) static and extern (d) auto and register

1.8 Consider the following declarations

```
union id {
   char color;
   int size;
}
struct {
   char country;
   int date;
   union id i;
} flag;
```

To assign a color to a flag, the correct statement would be

(a) flag.color='W'; (b) flag.i.color='W';
(c) flag.color='White'; (d) flag.i.color='White';

1.9 Which of the following is true for the switch statement:

```
switch (var)
{
};
```

(a) can be used when only one variable is tested
(b) the variable must be an integral type
(c) each possible value of the variable can control a single branch
(d) all of the above

1.10 Enumeration is:

(a) a list of strings (b) a set of numbers
(c) a set of legal values possible (d) none of the above

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 An ampersand (&) is required before each variable name is printf.

2.2 It is an error to place the pound(#) sign of a preprocessor control line in any column except column 1.
2.3 The goto statement is a branching statement in 'C' programming.
2.4 An expression with the star operator, such as *ptr. cannot occur on the left-hand side of an assignment statement.
2.5 An array's name by itself cannot occur as the left-hand side of an assignment statement.
2.6 Function calls cannot be nested.
2.7 Each function must have at least one return statement.
2.8 A linked list is a data structure, which is created by dynamic allocation of memory.
2.9 The declaration void function-name () indicates that function-name returns nothing to the calling program.
2.10 Member variables of two different structures may have the same name.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 a[i] can be written as	A. Unending loop if no break statement inside the body
3.2 Calloc()	B. p is a pointer to a function that returns integer
3.3 union	C. allocate and clear memory
3.4 for(;;)	D. *(a+i)
3.5 register variable	E. allocates memory but does not clear memory
3.6 int (*p)[10]	F. is a memory location that is used by several different variables, which may be of different type
3.7 int (*p) (void*,void*)	G. directives
3.8 do-while loop	H. Increase in speed of execution
3.9 #include, #define	I. p is pointer to an array of integers
3.10 typedef	J. p is function that returns pointer to integer
	K. This guarantees that the loop is executed at least once before continuing
	L. preprocessor
	M. can be used to create variables of new types

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a) array	(b) string	(c) typedef
(d) cast	(e) structure	(f) EOF
(g) void	(h) c functions	(i) gets
(j) c preprocessor	(k) getstr	(l) DEFINE
(m) &	(n) char	(o) malloc

4.1 The _____ operator is used to convert one data type to another.
4.2 If the pointer variable ptr holds the address of a char, the data type of *ptr will be _____.
4.3 File is defined with a(n) _____ statement.
4.4 Functions of type _____ do not return a value to the calling segment.
4.5 A(n) _____ is a collection of variables under a single name.
4.6 _____ reads a whole line of input into a string until a new line of EOF is encountered.
4.7 Expressions can be made equivalent to a single identifier using the preprocessor _____ command.
4.8 The _____ is a program that is executed before the source code is compiled.
4.9 The _____ operator is unary operator to find the value of a variable.
4.10 Dynamic allocation of memory for structure can be done with the help of the _____ function.

PART TWO

(Answer Any FOUR Questions)

5. (a) Write the C statements (all necessary statements) that open the file inf.dat for reading, and open the file outf.dat for writing.

(b) Write the C program to write
"Introduction to C-Programming"
to the file outf.dat.

(c) Write a C program that reads integers from the file scores.dat. After all the integers have been read, the program writes the sum of all the nonnegative integers to the video display. Assume that the file scores.dat contains at least one integer. **(5+4+6)**

6. (a) Write a 'C' program to calculate and display the monthly income of a salesperson corresponding to the value of monthly sales input in the scanf() function, let us consider the following commission schedule: (Note : use if-else statement)

Monthly sales	**Income**
Greater than or equal to Rs. 50,000	375 plus 16% of sales
Less than Rs. 50,000 but Greater than or equal to Rs. 40,000	350 plus 14% of sales
Less than Rs. 40,000 but Greater than or equal to Rs. 30,000	325 plus 12% of sales
Less than Rs. 30,000 but Greater than or equal to Rs. 20,000	300 plus 9% of sales
Less than Rs. 20,000 but Greater than or equal to Rs. 10,000	250 plus 5% of sales
Less than Rs. 10,000	200 plus 3% of sales

(b) What is printed after execution of each of the following c-programs?

1.
```
void main ()
{
float reals[5];
*(reals+1)=245.8;
*reals = *(reals +1);
printf("%f",reals[0]);
}
```

2.
```
void main()
{
int nums[3];
int *ptr = nums;
nums[0]=100;
nums[1]=1000;
nums[2]=10000;
printf("%d\n",++*ptr);
printf("%d",*ptr);
}
```

3.
```
void main()
{
int digit=0;
while (digit <=9)
printf("%d\n",digit++);
}
```

4.
```
void main()
{
int a=7,b=6;
fun1(a,b);
printf("\n a is %d b is %d", a,b);
}
int fun1(int c, int d)
{ int e;
e=c*d;
d=7*c;
printf(\n c is %d d is %d e is %d", c,d,e);
return;
}
```

(7+[2*4])

7. (a) Write a C function word_count() to count the number of words in a given string and then call in main().

(b) Write a C function print_upper() to prints its character argument in uppercase.

(c) Write a macro that clears an array to zero. **(7+4+4)**

8. (a) What is a structure? Define a structure that contains the following members:
   (i) An integer quantity called acct_no
   (ii) A character called acct_type
   (iii) A 40-element character array called name
   (iv) A floating-point quantity called balance
   (v) A structure variable called lastpayment, of type date: defined as an integer called month; an integer called day; an integer called year.
   (vi) Include the user_defined data type *account* within the definition.
   (vii) Include structure variable customer, which is 100-element array of structures called account.

   (b) What is pointer in C? How pointers and arrays are related? **(8+7)**

9. Write short notes on any three of the following:
   (a) Switch statement (give proper syntax and examples)
   (b) What do you mean by loop? How while-loop and do-loop differs?
   (c) What is C preprocessor? Explain any two C preprocessor commands with example.
   (d) Break and Continue statements. **(3*5)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(January 2006)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 The following is a program

```
Main( )
      {
         int x = 0;
         while (x<=10)
         for ( ; ; )
            if( ++ x%10 = = 0)
               break;
            printf ("x = %d", x);
      }
```

What will be the output of the above program?

(a) Will print $x = 10$
(b) Will give compilation error
(c) Will give runtime error
(d) Will print $x = 20$

1.2 Consider the following variable declaration

```
Union x{
      int i;
      float f;
```

```
        char c;
    }
```

If the size of *i*, *f* and *c* are 2 bytes, 4 bytes and 1 byte respectively, then the size of the variable *y* is:-

(a) 1 byte (b) 2 bytes
(c) 4 bytes (d) 7 bytes

1.3 Pick up the odd one out from the following

(a) $x = x - 1$ (b) $x - = 1$
(c) $x - -$ (d) $x = - 1$

1.4 What is the value of 'average' after the following program is executed?

```
main( )
   {
   int sum, index,
   float average;
   sum = 0;
   for ( ; ; ) {
      sum = sum + index;
      ++ index;
      if (sum > = 100) break;
         }
   average = sum / index;
   }
```

(a) 91/13 (b) 91/14
(c) 105/14 (d) 105/15

1.5 Suppose *i*, *j*, *k* are integer variables with values 1, 2, 3 respectively. What is the value of the following expression?

```
! (( j + k ) > ( i + 5 ))
```

(a) 6 (b) 5
(c) 1 (d) 0

1.6 If a = –11 and b = –3. What is the value of a % b?

(a) –3 (b) –2
(c) 2 (d) 3

1.7 If c is a variable initialized to 1, how many times will the following loop be executed?

```
while (( c > 0 && (c < 60))
   {
      c ++;
   }
```

(a) 61 (b) 60
(c) 59 (d) 1

1.8 Which one of the following describes correctly a static variable?

(a) This cannot be initialized.

(b) This is initialized once at the commencement of execution and cannot be changed at run time.
(c) This retains its value through the life of the program.
(d) This is same as an automatic variable but is placed at the head of a program.

1.9 What will be the output of the following program?

```
main( )
   {
   int a, *ptr, b, c;
   a = 25;
   ptr = &a;
   b = a + 30;
   c = *ptr;
   printf ("%d %d %d", a, b, c);
   }
```

(a) 25, 25, 25 (b) 25, 55, 25
(c) 25, 55, 25 (d) None of the above

1.10 If a = 0xaa and $b = a << 1$ then which of the following is true
(a) $b = a$ (b) $b = 2a$
(c) $a = 2b$ (d) $n = a - 1$

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 It is not possible to print the % character as the function printf treats % as the beginning of a conversion specification.
2.2 A structure can include one or more pointers as members.
2.3 It is not possible to have formatted input / output in 'C'.
2.4 It is not possible to have nested if - else statements in 'C'.
2.5 The increment operator ++ does not work with float variable.
2.6 *a is the same as a[ ] in a parameter declaration.
2.7 In 'C' programming language, strings are represented using an array.
2.8 Relational operators have higher precedence than arithmetic operators.
2.9 A structure cannot be a member of a union.
2.10 *p++ increments the content of the location pointed by p.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 An Operator in 'C' permits two different expression to appear in situations where only one expression is ordinarily be used.	A. The operator &&

3.2	Variables, internal to a function, come into existence when the function is called	B.	Static variables
3.3	No space allocated for storage of character during compilation time	C.	Global variables
3.4	p is pointer to a function that returns a pointer to integer	D.	The comma operator (,)
3.5	Self-referencing structure	E.	int *p[10]
3.6	Accomplishing indirection with pointer to structure	F.	automatic variable
3.7	Random access in the file; file specified through file descriptor	G.	Useful for link-list implementation
3.8	Returns initialized storage in run-time	H.	seek
3.9	Variables can be defined in 'C' which occupies less space than character variables	I.	fseek
3.10	Valid mode for opening a file; permits read and write	J.	calloc
		K.	w+
		L.	malloc
		M.	r+
		N.	bit-fields
		O.	arrow operator
		P.	int(*pc))
		Q.	char *s

4. Each sentence below has a blank space to fit one of the word(s) or phrase(s) in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

(a)	Change	(b)	trinary	(c)	-operator
(d)	Automatic	(e)	Variable	(f)	"&" operator
(g)	Static	(h)	Dynamically	(i)	Externally
(j)	void	(k)	A character	(l)	An integer
(m)	EOF	(n)	fclose	(o)	True

4.1 printf( ) function uses _____ number of arguments.
4.2 _____ can be used as both binary and unary operators.
4.3 _____ Link lists can be created _____.
4.4 Loop invariants are assertions that remain _____ before and after execution of loops.
4.5 File descriptor is _____.
4.6 Pointer arguments enable a function to access and _____ objects defined in the calling routine.
4.7 The function getchar( ) returns _____ when there is no more input character.
4.8 Any pointer can be cast to _____ without loss of information.
4.9 To prevent the use of functions across different files, _____ storage class is used.
4.10 ? : is _____ operator.

PART TWO
(Answer Any FOUR Questions)

5. (a) Discuss with the help of examples the action of break statement and the continue statement.
(b) Does the null statement have any uses besides indication that the body of a loop is empty? Explain.
(c) What is the purpose of the \? Escape sequence? **(8+4+3)**

6. (a)If it legal to put a function declaration inside the body of another function? If yes, give an example.
(b) Is it legal for a function *f*1 to call *f*2, which then calls *f*1? Justify your answer.
(c) Write a 'C' function that returns the k-th digit from the right in the positive integer *n*. For example, digit (829, 3) returns 89. If *k* is greater than the number of digits in *n* then the function is to return -1. Include appropriate documentation in your program. **(4+2+9)**

7. (a) If a pointer is an address, what does the expression like *p* + *j* mean?
(b) Is i[a] same as a[i]? Justify your answer.
(c) Write the following function:

```
Bool search (int a[ ], int n, int x);
```

Where *a* is an array to be searched, *n* is the number of elements in the array, and x is the search key. "search" should return TRUE if *x* matches some element of *a*, FALSE if it doesn't. Use pointer arithmetic to visit array elements. Include appropriate documentation in your program. **(4+2+9)**

8. (a) Develop an algorithm to do the following:
Read an array of 20 elements and then send all negative elements of the array to the end without altering the original sequence.
(b) Draw a flow chart and then write a 'C' program to generate first 15 members of the following sequence.
1, 3, 4, 7, 11, 18, 29, . . . **(5+10)**

9. Develop a flowchart and then write a program for analyzing a line of text stored in a file by examining each of the characters and displaying into which of several different categories vowels, constants, digits, white spaces it fails. Count of the number of vowels, consonants, digits and white space characters. Include an appropriate documentation in your program. **(15)**

M4.1-R3 : PROGRAMMING & PROBLEM SOLVING THROUGH 'C' LANGUAGE
(July 2005)

NOTE:

1. There are **TWO PARTS** in this Module/Paper. **PART ONE** contains **FOUR** questions and **PART TWO** contains **FIVE** questions.
2. **PART ONE** is to be answered in the **TEAR-OFF ANSWER SHEET** only, attached to the question paper, as per the instructions contained therein. **PART ONE** is **NOT** to be answered in the answer book.
3. Maximum time allotted for **PART ONE** is **ONE HOUR**. Answer book for **PART TWO** will be supplied at the table when the answer sheet for **PART ONE** is returned. However, candidates, who complete **PART ONE** earlier than one hour, can collect the answer book for **PART TWO** immediately on handing over the answer sheet for **PART ONE**.

Total Time : 3 Hours Max. Marks : 100

(PART ONE - 40; PART TWO - 60)

PART ONE

(Answer ALL Questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 Consider the following code segment

```
Main( ){
   char s[100];
   scanf("%s", s);
   printf("%3s", s);}
```

If "India is Great" is entered upon the execution of the program for *s*, then that will be the output of the above code segment?

(a) India is Great
(b) Ind
(c) India
(d) The compiler gives the error that "*%3s*" is not valid format string in printf.

1.2 Consider the following code segment:

```
char s[ ] = "abc";
char *p = s;
```

Which of the following is not a valid statement to access 'b' in string s?

(a) *(++s) (b) *(++p)
(c) s[1] (d) *(p+1)

1.3 What will be the output of the following code segment?

```
main( ){
   int j = 1, k = 2;
```

```
            if (j & k)
               printf(" Successful ");
            else
               printf(" Unsuccessful ");}
```

(a) Successful (b) Unsuccessful
(c) Successful Unsuccessful (d) Unsuccessful Successful

1.4 What will be the output of the following code segment?

```
         main( ){
            int j = 1, k = 2;
            if (j = = 1)
               printf("%d", j);
            else;
               printf("%d", k); }
```

(a) 1 (b) 2
(c) 12 (d) 21

1.5 What will be the output of the following code segment?

```
      main( ) {
      int n = 10, d = 0;
      if (d ! = 0 && n / d > 1)
         printf (" Successful " );
      else
         printf (" Unsuccessful " ); }
```

(a) Successful
(b) Unsuccessful
(c) Compile time error of division by zero will be generated.
(d) Run time error of division by zero will be generated.

1.6 What will be the output of the following code segment?

```
         Main( ) {
            char s[10];
            strcpy(s, "abc");
            printf("%d %d", strlen(s), sizeof(s)); }
```

(a) 3 10 (b) 3 3
(c) 10 10 (d) 10 3

1.7 Which of the following is true for the following statement?

```
      NurseryLand.Nursery.Students = 10;
```

(a) The structure Students is nested within the structure Nursery.
(b) The structure NurseryLand is nested within the structure Nursery.
(c) The structure Nursery is nested within the structure students.
(d) The structure Nursery is nested within the structure NurseryLand.

1.8 Which of the following statements is the odd one out?

(a) j = j + 1; (b) j++;
(c) j += 1; (d) j =+ 1;

1.9 The default return type of main( ) is
(a) int (b) void
(c) char (d) float

1.10 The scope of a variable of automatic storage class is limited to the
(a) Program in which it is defined
(b) Block in which it is defined
(c) Loop in which it is defined
(d) Function in which it is defined

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and ENTER in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 Only numeric constants can be #defined.
2.2 The default group may appear anywhere within the switch statement.
2.3 Like nested structures, functions can also be nested.
2.4 In call by reference, a change in the value of arguments by the called function is reflected in the calling function.
2.5 The C computer generates an error if the subscript used for an array exceeds the size of the array.
2.6 All members of a structure are allocated contiguous memory locations.
2.7 The statement FILE *fp means the variable fp is a pointer pointing to the actual file on the disk.
2.8 To pass the command line arguments, only argc and argv can be used as the format argument names.
2.9 The file stdin has to be explicitly opened before it can be used.
2.10 The ++ operator is a binary operator.

3. Match words and phrases in column X with the closest related meaning/word(s)/phrase(s) in column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X	*Y*
3.1 Multiple initializations in a for loop	A. goto
3.2 One's Complement	B. register
3.3 Retaining values	C. int *p[10];
3.4 Fast access	D. ;
3.5 Array of pointers	E. static
3.6 Null terminated array of characters	F. gotoxy
3.7 Excape character	G. !
3.8 Moving read/ write pointer	H. structure
3.9 Changing data type	I. array
3.10 Unconditional jump	J. ftell
	K. extern

L. int (*p)(10;]
M. String
N. /
O. auto
P. type casting
Q. fseek
R. \
S. typedef
T. ~
U. ,

4. Each sentence below has a blank space to fit one of the words or phrases in the list below. Enter your choice in the answer sheet provided following instructions therein. (1×10)

(a)	int	(b)	Right to left	(c)	for
(d)	library function	(e)	fread	(f)	do..while
(g)	column major	(h)	&	(i)	char
(j)	structure	(k)	operator	(l)	fwrite
(m)	call by reference	(n)	row major	(o)	division
(p)	left to right	(q)	*	(r)	fclose
(s)	call by value	(t)	multiplication	(u)	fopen
(v)	–>	(w)	linked list		

4.1 The sizeof is a(n) _____.
4.2 In C, the arguments are passed from _____ to the called function.
4.3 To return more than one value from a function use _____.
4.4 An array is stored in _____ order in the memory.
4.5 The operator _____ is used exclusively with pointers.
4.6 Internally, the enumerators are stored as _____.
4.7 The function _____ must appear in the programs manipulating disk files.
4.8 Right shifting an unsigned integer is equivalent to its _____ by 2.
4.9 The body of a _____ loop might not be executed even once.
4.10 The self-referential structures are used to implement.

PART TWO

(Answer Any FOUR Questions)

5. Write a 'C' program to copy one file to another such that the first character of every word is removed from it and appended at its end. Assume that the maximum size of each word is 10 characters and each word is separated either by newline(s) or space(s). For example, if "Test String" is a line in the source file then "estT tringS" is copied in the target file. **(15)**

6. (a) Write a function to sort the characters of the string passed to it as argument.
(b) Write a function to remove all blank spaces in the string passed to it as argument. **(9+6)**

7. (a) Write a function, my_atoi, similar to the library function atoi that returns the numeric value corresponding to the string passed to it as an argument.

(b) Write a function to display the product of two matrices passed to it. The display must be in the matrix form itself. **(8+7)**

8. (a) Write a function to find which of the two unsigned numbers passed to it is greater using the bitwise operators. The function should return 1, if first is greater, –1 if second is greater and 0 if they are equal.

(b) Write a function to generate a triangle as shown below: (for $n = 4$)

```
      A
    A B A
  A B C B A
A B C D C B A
```

The number n is passed as an argument to the function. **(8+7)**

9. (a) Define a structure of a node of a linked list having an integer data member.

(b) Use the above structure and write the function for the following:

(i) a function to merge two sorted linked lists. The function accepts pointers to the head of sorted linked lists and returns the pointer to the head of the merged linked list.

(ii) a function to find the maximum and minimum of the data in the linked list. The function accepts a pointer to the head of the linked list, and returns the maximum and minimum values found in the list. **(2+[8+5])**

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W