VENKATESHWARA OPEN UNIVERSITY

www.vou.ac.in

COMPUTER FUNDAMENTALS AND PROGRAMMING

BCA

[BCA-101]

COMPUTER FUNDAMENTALS AND PROGRAMMING

BCA

[BCA-101]

VENKATESHWARA OPEN UNIVERSITY
www.vou.ac.in

Authors:

B. Basavaraj: Unit (1.0-1.2.4, 1.4-1.5.2, 1.8-1.9) Copyright © Vikas® Publishing House, 2010

Vivek Kesari: Unit (1.6-1.7, 1.11) Copyright © Vivek Kesari, 2010

S. Mohan Naidu: Unit (2.0-2.3, 2.5, 2.7, 3.3, 3.6-3.9, 3.17-3.18, 3.20-3.21, 3.23) Copyright © S. Mohan Naidu, 2010

R. Subburaj: Unit (3.0-3.2, 3.4-3.5, 3.10-3.12, 3.19, 3.22) Copyright © Vikas® Publishing House, 2010

Vikas® Publishing House: Units (1.3, 1.10, 1.12-1.17, 2.4, 2.6, 2.8-2.13, 3.13-3.16, 3.24-3.32)
Copyright © Vikas® Publishing House, 2010

Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT LTD
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

SYLLABI-BOOK MAPPING TABLE

Computer Fundamentals and Programming

Syllabi	Mapping in Book
Computer Fundamentals Number system: Decimal, octal, binary and hexadecimal; Representation of integers, fixed and floating points, character representation: ASCII, EBCDIC; Functional units of computer, I/O devices, primary and secondary memories;	**Unit 1:** Computer Fundamentals (**Pages: 3-40**)
Programming Fundamentals Algorithm development, Techniques of problem solving, Flow-charting, Step-wise refinement, Algorithms for searching, Sorting (exchange and insertion), merging of ordered lists.	**Unit 2:** Programming Fundamentals: Algorithms and Flowcharts (**Pages: 41-161**)
Programming Representation of integers, characters, real Data types: constants and variables; Arithmetic Expressions, Assignment, statement, Logical expression, Sequencing, Alteration and iteration; string processing; Sub programs, Recursion, Files and pointers; Structured programming concepts; Top down Design, Development of efficient programs; program correctness; Debugging and testing of Programs.	**Unit 3:** Programming (**Pages: 163-333**)

CONTENTS

UNIT 3 PROGRAMMING **163-333**

INTRODUCTION

NOTES

Computers have become an indispensable part of our lives. We depend on computers for our day-to-day tasks. But, have you ever wondered how these *dumb* machines have acquired their present-day status in our society? Yes, computers are actually dumb machines, because they cannot function on their own. A bare computer machine, without any software loaded in it, is as good as a piece of wood. However, these bare machines are capable of performing zillions of calculations in almost no time. To reap the benefits of this enormous capability, we need to program these machines, i.e. to provide them with instructions to do our desired tasks.

To make use of computers, we need programs (or a set of instructions) that are written in a computer programming language. A computer programming language is a script language that a computer can understand. Now, to tell the computer to perform a desired task, we need to write our requirements in a computer programming language and feed these instructions in the computer. All this is as simple as it sounds, provided that we have thorough understanding of components of a computer and programming concepts and techniques.

The present book lays out the very basic concepts and techniques required to write a successful computer program. Unit 1 introduces you to the various number systems and character sets the understanding of which is a prerequisite to writing computer programs. The unit also deals with the input–output devices and computer memory types. Unit 2 teaches you the basics of writing algorithms and making use of flowcharts to write efficient and effective programs. Unit 3 covers the basic constructs of a programming language (C in particular) to enable you to write computer programs to perform various tasks. The book includes ample examples to render better understanding of all the concepts.

The book follows the SIM format or the self-instructional mode wherein each Unit begins with an Introduction to the topic followed by an outline of the Unit Objectives. The detailed content is then presented in a simple and an organized manner, interspersed with Check Your Progress questions to test the understanding of the students. A Summary along with a list of Key Terms and a set of Questions and Exercises is also provided at the end of each unit for effective recapitulation.

UNIT 1 COMPUTER FUNDAMENTALS

NOTES

Structure

1.0 INTRODUCTION

The word 'compute' means to calculate and we are all aware of the calculations we are required to do in our everyday life, whether it is addition, subtraction, multiplication or subtraction. The earliest known tool for calculations was the abacus, and it was thought to have been invented in Babylon circa 2400 BC. Today, you cannot even think of a life without computers as they exist in every area of our lives, whether it be banking, airlines, communications or simply working in an office.

1.1 UNIT OBJECTIVES

NOTES

After going through this unit, you will be able to:

- Understand the different systems of arithmetic used in digital systems
- Discuss the different arithmetic concepts of integers and fixed and floating points which are used in computer programming.
- Elaborate on the different functional units of a computer.
- Analyse the main and auxiliary memory units in a computer.

1.2 NUMBER SYSTEM

A number represents a thought that refers to a precise amount of something. Numbers can be expressed in words, gestures and symbols. When expressed in words, numbers are spoken out. Numbers are expressed through gestures using (usually) our hands. Numbers are expressed in symbols that can be written down. A number symbol is known as a numeral. Hence, a number is a precise idea about an amount, which you form in your minds when you look at a numeral, hear it when it is spoken or see it when it is signalled by hands.

On hearing the word number, you immediately think of the familiar decimal number system with its 10 digits; 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. These numerals are called Arabic numerals. Your present number system provides modern mathematicians and scientists with great advantages over those of previous civilizations, and is an important factor in our advancement. Since fingers are the most convenient tools nature has provided, human beings use them in counting. So, the decimal number system followed naturally from this usage.

A number of base or radix *r*, is a system that uses distinct symbols of *r* digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of *r*, and then form the sum of all the weighted digits. To develop a number system, you can choose any whole number greater than 1 as the base of the system. The base of a number system refers to the number of digits used in that system.

There are four systems of arithmetic, which are often used in digital systems. These systems are:

1. Decimal
2. Binary
3. Octal
4. Hexadecimal

NOTES

In any number system, there is an ordered set of symbols known as digits. A collection of these digits makes a number which in general has two parts, integer and fractional, and are set apart by a radix point (.). Hence, a number system can be represented as,

$$N_{\hat{b}} = \underbrace{a_{n-1}a_{n-2}a_{n-3} \ldots a_1a_0}_{\text{Integer portion}} \cdot \underbrace{a_{-1}a_{-2}a_{-3} \ldots a_{-m}}_{\text{Fractional portion}}$$

where N = A number

b = Radix or base of the number system

n = Number of digits in integer portion

m = Number of digits in fractional portion

a_{n-1} = Most significant digit (MSD)

a_{-m} = Least significant digit (LSD)

and $0 \leq (a_i \text{ or } a_{-f}) \leq \times b_{-1}$

Base or Radix: The base or radix of a number is defined as the number of different digits which can occur in each position in the number system.

1.2.1 Decimal Number System

The number system which utilizes ten distinct digits, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 is known as the decimal number system. It represents numbers in terms of groups of ten, as shown in Figure 1.1.

You would be forced to stop at 9 or to invent more symbols if it were not for the use of positional notation. It is necessary to learn only 10 basic numbers and positional notational system in order to count any desired figure.

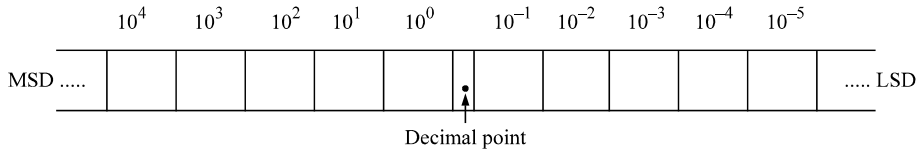


Figure 1.1 *Decimal Position Values as Powers of 10*

The decimal number system has a base or radix of 10. Each of the ten decimal digits 0 through 9, has a place value or weight depending on its position. The weights are units, tens, hundreds and so on. The same can be written as the power of its base as 10^0, 10^1, 10^2, 10^3... etc. Thus, the number 1993 represents quantity equal to 1000 + 900 + 90 + 3. Actually, this should be written as $\{1 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 3 \times 10^0\}$. Hence, 1993 is the sum of all digits multiplied by their weights. Each position has a value 10 times greater than the position to its right.

NOTES

For example, the number 379 actually stands for the following representation:

$$\begin{array}{lll} 100 & 10 & 1 \\ 10^2 & 10^1 & 10^0 \\ 3 & 7 & 9 \end{array}$$

$$3 \times 100 + 7 \times 10 + 9 \times 1$$

$$\therefore \quad 379_{10} = 3 \times 100 + 7 \times 10 + 9 \times 1$$

$$= \mathbf{3 \times 10^2 + 7 \times 10^1 + 9 \times 10^0}$$

In this example, 9 is the least significant digit (LSD) and 3 is the most significant digit (MSD).

Example 1.1: Write the number 1936.469 using decimal representation.

Solution: $1936.469_{10} = 1 \times 10^3 + 9 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} + 6 \times 10^{-2} + 9 \times 10^{-3}$

$= 1000 + 900 + 30 + 6 + 0.4 + 0.06 + 0.009 =$ **1936.469**

It is seen that powers are numbered to the left of the decimal point starting with 0 and to the right of the decimal point starting with –1.

The general rule for representing numbers in the decimal system by using positional notation is as follows:

$$a_n a_{n-1} \ldots a_2 a_1 a_0 = a_n 10^n + a_{n-1} 10^{n-1} + \ldots a_2 10^2 + a_1 10^1 + a_0 10^0$$

Where *n* is the number of digits to the left of the decimal point.

1.2.2 Binary Number System

A number system that uses only two digits, 0 and 1 is called the **binary number system**. The binary number system is also called a **base two system**. The two symbols 0 and 1 are known as **bits** (binary digits).

The binary system groups numbers by twos and by powers of two, shown in Figure 1.2. The word binary comes from a Latin word meaning two at a time.

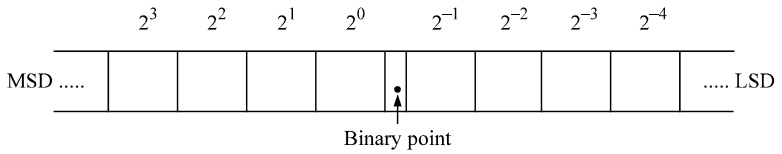


Figure 1.2 *Binary Position Values as a Power of 2*

The weight or place value of each position can be expressed in terms of 2, and is represented as 2^0, 2^1, 2^2, etc. The least significant digit has a weight of 2^0 (= 1). The second position to the left of the least significant digit is multiplied by 2^1 (= 2). The third position has a weight equal to 2^2 (= 4). Thus, the weights are in the ascending powers of 2 or 1, 2, 4, 8, 16, 32, 64, 128, etc.

The numeral 10_{two} (one, zero, base two) stands for **two**, the base of the system.

NOTES

In binary counting, single digits are used for **none** and **one**. Two-digit numbers are used for 10_{two} and 11_{two} [2 and 3 in decimal numerals]. For the next counting number, 100_{two} (4 in decimal numerals) three digits are necessary. After 111_{two} (7 in decimal numerals) four-digit numerals are used until 1111_{two} (15 in decimal numerals) is reached, and so on. In a binary numeral, every position has a value 2 times the value of the position to its right.

A binary number with 4 bits, is called a **nibble** and a binary number with 8 bits is known as **a byte**.

For example, the number 1011_2 actually stands for the following representation:

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$$

$$\therefore \quad 1011_2 = 8 + 0 + 2 + 1 = 11_{10}$$

In general,

$$[b_n b_{n-1} \ldots b_2, b_1, b_0]_2 = b_n 2^n + b_{n-1} 2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

Similarly, the binary number 10101.011 can be written as

1	0	1	0	1	.	0	1	1
2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
(MSD)						(LSD)		

$$\therefore \quad 10101.011_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 16 + 0 + 4 + 0 + 1 + 0 + 0.25 + 0.125 = \mathbf{21.375_{10}}$$

In each binary digit, the value increases in powers of two starting with 0 to the left of the binary point and decreases to the right of the binary point starting with power –1.

Why is Binary Number System Used in Digital Computers?

The binary number system is used in digital computers because all electrical and electronic circuits can be made to respond to the two states concept. A switch, for instance, can be either opened or closed—only two possible states exist. A transistor can be made to operate either in cut-off or saturation; a magnetic tape can be either magnetized or non-magnetized; a signal can be either HIGH or LOW; a punched tape can have a hole or no hole. In all of these examples, each device is operated in any one of the two possible states and the intermediate condition does not exist. Thus, 0 can represent one of the states and 1 can represent the other. Hence, binary numbers are convenient to use in analysing or designing digital circuits.

NOTES

1.2.3 Octal Number System

The octal number system was used extensively by early minicomputers. However, for both large and small systems, it has largely been supplanted by the hexadecimal system. Sets of 3-bit binary numbers can be represented by octal numbers and this can conveniently be used for the entire data in the computer.

A number system that uses eight digits, 0, 1, 2, 3, 4, 5, 6 and 7 is called an **octal number system.**

It has a base of **eight**. The digits, 0 through 7 have exactly the same physical meaning as decimal symbols. In this system, each digit has a weight corresponding to its position as shown:

$$a_n 8^n + \cdots + a_3 8^3 + a_2 8^2 + a_1 8^1 + a_{-1} 8^{-1} + a_{-2} 8^{-2} + \cdots + a_{-n} 8^{-n}$$

Octal Odometer

Octal odometer is a hypothetical device similar to the odometer of a car. Each display wheel of this odometer contains only eight digits (teeth), numbered 0 to 7. When a wheel turns from 7 back to 0 after one rotation, it sends a carry to the next higher wheel. Table 1.1 lists equivalent numbers in decimal, binary and octal systems.

Table 1.1 *Equivalent Numbers in Decimal, Binary and Octal Systems*

Decimal (Radix 10)	Binary (Radix 2)	Octal (Radix 8)
0	000 000	0
1	000 001	1
2	000 010	2
3	000 011	3
4	000 100	4
5	000 101	5
6	000 110	6
7	000 111	7
8	001 000	10
9	001 001	11
10	001 010	12
11	001 011	13
12	001 100	14
13	001 101	15
14	001 110	16
15	001 111	17
16	010 000	20

Consider an octal number $[567.3]_8$. It is pronounced as five, six, seven octal point three and not five hundred sixty seven point three. The co-efficients of the integer part are $a_0 = 7$, $a_1 = 6$, $a_2 = 5$ and the co-efficient of the fractional part is $a_{-1} = 3$.

1.2.4 Hexadecimal Number System

NOTES

The hexadecimal system groups numbers by sixteen and powers of sixteen. Hexadecimal numbers are used extensively in microprocessor work. Most minicomputers and microcomputers have their memories organized into sets of bytes, each consisting of eight binary digits. Each byte either is used as a single entity to represent a single alphanumeric character or broken into two 4 bit pieces. When the bytes are handled in two 4 bit pieces, the programmer is given the option of declaring each 4 bit character as a piece of a binary number or as two BCD numbers.

The hexadecimal number is formed from a binary number by grouping bits in groups of 4 bits each, starting at the binary point. This is a logical way of grouping, since computer words come in 8 bits, 16 bits, 32 bits and so on. In a group of 4 bits, the decimal numbers 0 to 15 can be represented as shown in Table 1.2.

The hexadecimal number system has a base of 16. Thus, it has 16 distinct digit symbols. It uses the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 plus the letters A, B, C, D, E and F as 16 digit symbols. The relationship among octal, hexadecimal, and binary is given in Table 1.2. Each hexadecimal number represents a group of four binary digits.

Table 1.2 *Equivalent Numbers in Decimal, Binary, Octal and Hexadecimal Number Systems*

Decimal (Radix 10)	Binary (Radix 2)	Octal (Radix 8)	Hexadecimal (Radix 16)
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	0001 0000	20	10
17	0001 0001	21	11
18	0001 0010	22	12
19	0001 0011	23	13
20	0001 0100	24	14

NOTES

Counting in Hexadecimal

When counting in **hex**, each digit can be incremented from 0 to *F*. Once it reaches *F*, the next count causes it to recycle to 0 and the next higher digit is incremented. This is illustrated in the following counting sequences: 0038, 0039, 003A, 003B, 003C, 003D, 003E, 003F, 0040; 06B8, 06B9, 06BA, 06BB, 06BC, 06BD, 06BE, 06BF, 06C0, 06C1.

CHECK YOUR PROGRESS

1. Which are the four systems of arithmetic used in digital systems?
2. What is an Octal Odometer?

1.3 INTEGERS

In mathematics, the study of integers is termed as number theory. The integers are the natural numbers which include the numbers 0, 1, 2, 3, …. and their negatives –1, – 2, – 3, – 4, …. These numbers are written without using a fraction or decimal component and can be represented as set of natural numbers in the form {... –2, –1, 0, 1, 2, ...} and is denoted by a **Z** in Boldface or Unicode ℤ which stands for *Zahlen*. The numbers 6, 57, 89, –5, –786 are integers but 1.2, 6.98, 3 4/5, 1 1/2 are not integers. The positive elements of integers are well-ordered. The countably infinite sets are formed by integers. Figure 1.3 represents the natural numbers:

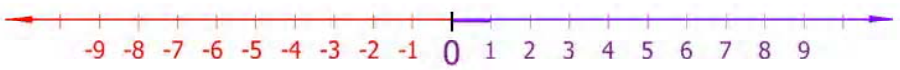


Figure 1.3 *Natural Numbers*

Hence, integers **are the set of whole numbers including their opposites as shown in Figure 1.4.** The number line is used to represent the set of integers.

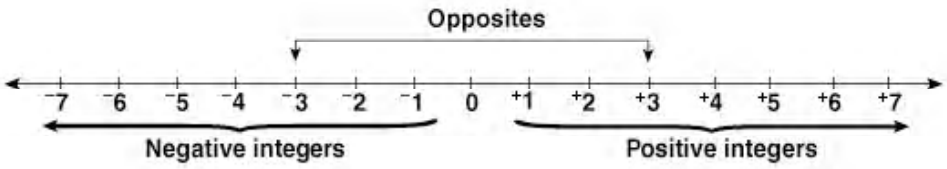


Figure 1.4 *Integers*

- The number line extends in left and right directions starting from centre point zero. The arrows indicate the direction.
- Whole numbers which are greater than zero are termed as *positive integers* and are on the right side of zero on the number line.
- Whole numbers which are less than zero are termed as *negative integers* and are on the left side of zero on the number line.

- Zero is called as neutral integer. It is neither positive nor negative.
- Except zero all the integers are represented using either positive (+) sign or negative (–) sign.
- When the two integers are at the same distance from zero and are on the opposite sides of the number line then they are termed as *opposites*. The integer on the side is represented with positive sign and on the left side is represented with negative sign. In the number line shown above +3 and –3 are the opposites. Each positive integer has its opposite as a negative integer.

Algebraic Properties

Z is associated with addition and multiplication operations same as natural numbers such that the sum and product of the two integers is always an integer. With negative natural numbers and especially zero **Z** is associated with subtraction. **Z** is not considered for division operation because the quotient of two integers may not be an integer. The natural numbers are used as exponentiation but the integers are not because the result may be a fraction if the exponent is negative.

Table 1.3 lists the basic properties of addition and multiplication for integers a, b and c.

Table 1.3 *Properties of Integers*

	Addition	**Multiplication**
Closure:	$a + b$ is an integer	$a \times b$ is an integer
Associativity:	$a + (b + c) = (a + b) + c$	$a \times (b \times c) = (a \times b) \times c$
Commutativity:	$a + b = b + a$	$a \times b = b \times a$
Existence of an identity element:	$a + 0 = a$	$a \times 1 = a$
Existence of inverse elements:	$a + (-a) = 0$	
Distributivity:	$a \times (b + c) = (a \times b) + (a \times c)$	
No zero divisors:		If $a \times b = 0$, then either $a = 0$ or $b = 0$ or both.

In abstract algebra, the above first five properties are used for addition such that **Z** under addition forms an abelian group. As an addition group, **Z** is considered as a cyclic group, because each nonzero integer is written as a finite sum of positive integers as $1 + 1 + \cdots + 1$ and for negative integers as $(-1) + (-1) + \cdots + (-1)$. Basically, **Z** addition is considered as the only infinite cyclic group because an infinite cyclic group is always isomorphic to **Z**.

The above mentioned first four properties for multiplication such that **Z** under multiplication is termed as commutative monoid. Each integer does not has a multiplicative inverse, for example there is no integer x such that $2x = 1$ since left

NOTES

hand side is even and right hand side is odd. This signifies that **Z** under multiplication does not form a group.

Though ordinary division is not explained on **Z** but it possesses a significant property termed as the division algorithm. In division algorithm, for two given integers a and b where $b \neq 0$ there exists exceptional integers q and r such that $a = q \times b + r$ and $0 \leq r < |b|$, where $|b|$ represents the absolute value of b. The integer q is termed as the *quotient* and r is termed as the *remainder*, which are the result of division of a by b. This algorithm is termed as Euclidean algorithm and is the basis to compute greatest common divisors.

Order-Theoretic Properties

Z is considered as a completely ordered set without an upper or lower bound. The ordering of **Z** is as follows:

$$..... -4 < -3 < -2 < -1 < 0 < 1 < 2 < 3 < 4 .....$$

An integer is termed as *positive* integer when it is greater than zero and as *negative* integer when it is less than zero. Zero is neither negative nor positive.

The ordering of integers can be compared with the algebraic operations as follows:

1. When $a < b$ and $c < d$, then $a + c < b + d$.
2. When $a < b$ and $0 < c$, then $ac < bc$.

It shows that **Z** along with the above given ordering forms an ordered ring.

Construction of Integers

In mathematics, we can construct integers as an equivalent class of ordered pairs of natural numbers as (a, b), where (a, b) stands for the result of subtraction of b from a. To verify that $1 - 2$ and $4 - 5$ will represent the same number as a result, we use an equivalence relation $\sim$ on these pairs using the rule $(a, b) \sim (c, d)$ specifically when $a + d = b + c$.

Addition and multiplication of integers can be represented in form of equivalence operations on the natural numbers as shown below:

$$[(a, b)] + [(c, d)] := [(a + c, b + d)]$$

$$[(a, b)] \cdot [(c, d)] := [(ac + bd, ad + bc)]$$

We can obtain the negation or additive inverse of any integer by reversing the order of the pair as follows:

$$-[(a, b)] := [(b, a)]$$

Thus we can define subtraction as the addition of the additive inverse:

$$[(a, b)] - [(c, d)] := [(a + d, b + c)]$$

The standard ordering on the integers is represented as:

$$[(a, b)] < [(c, d)] \quad \text{iff} \quad [(a + d < b + c)$$

Taking 0 as a natural number, we can consider the natural numbers as integers using the maps n to $[(n, 0)]$, where $[(a, b)]$ represents the equivalence class with (a, b) as a member.

This notation represents the integers as $\{..., -3, -2, -1, 0, 1, 2, 3, ...\}$.

The following are some examples of integers:

$0 = [(0, 0)] = [(1, 1)] = \ldots\ldots = [(k, k)]$

$1 = [(1, 0)] = [(2, 1)] = \ldots\ldots = [(k + 1, k)]$

$-1 = [(0, 1)] = [(1, 2)] = \ldots\ldots = [(k, k + 1)]$

$2 = [(2, 0)] = [(3, 1)] = \ldots\ldots = [(k + 2, k)]$

$-2 = [(0, 2)] = [(1, 3)] = \ldots\ldots = [(k, k + 2)]$

Absolute Value of an Integer

The absolute value of an integer is always represented with a positive number or zero. The absolute value of an integer n is specified by representing n in between the two vertical bars as $|n|$. For example, $|7| = 7$, $|-15| = 15$, $|0| = 0$, $|1334| = 1334$, etc.

Integer Coordinates

Integer coordinates are basically the pairs of integers that are used to establish points in a lattice in relation to a specific point termed as the origin. The origin shows coordinates (0, 0) which is the centre of the lattice and is taken as the starting point to find all other points. In the lattice, any other point takes a pair of (x, y) coordinates. Here, the value of x or x-coordinate defines that at what distance left or right the point is from the starting point (0, 0). Similar to the number line theory the negative is on the left side of the starting point while the positive is on the right side of the starting point. The value of y or y-coordinate defines that at what distance up or down the point is from the starting point (0, 0). The negative is towards down from the starting point while the positive is towards up from the starting point. Figure 1.5 shows the starting point (0, 0) where the x-axis and the y-axis meet. Point A shows coordinates (2, 3), because it is 2 points to the right side from the y-axis and 3 points up from the x-axis. Similarly the Point B, Point C, Point D, Point E and Point F show their coordinates. The minus (–) sign specifies that the points are on the left side and down of the starting point (0, 0) as represented below on the x-axis and y-axis.

NOTES

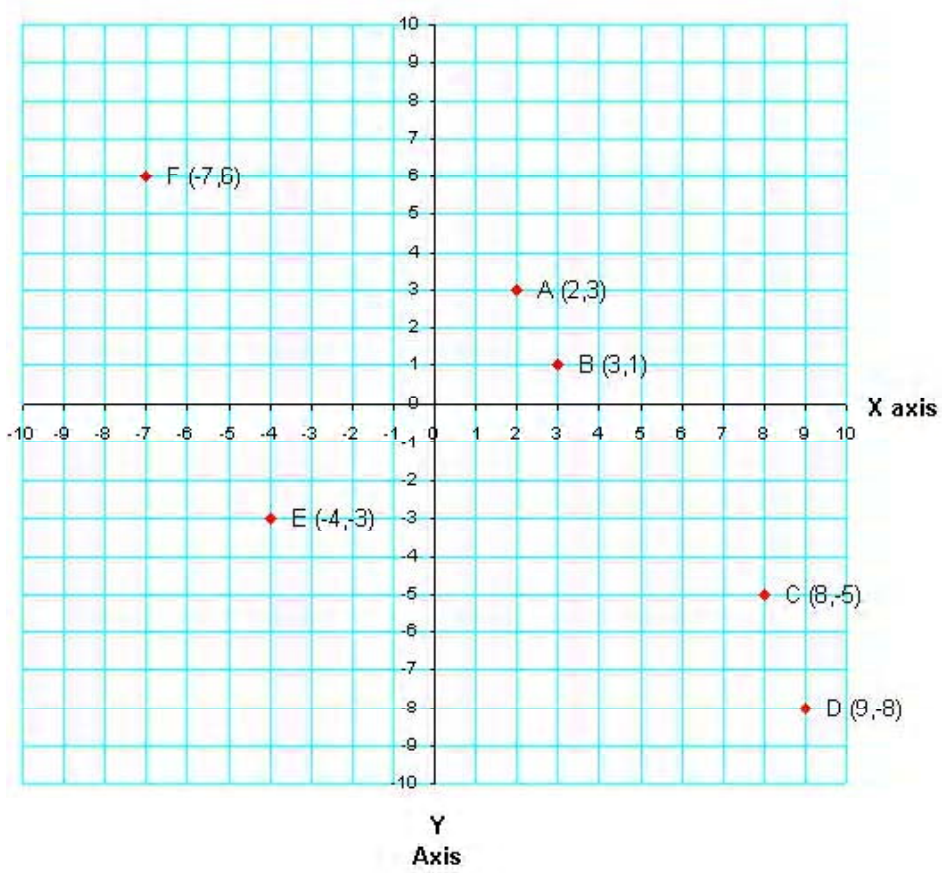


Figure 1.5 *Integer Coordinates*

1.4 FIXED AND FLOATING POINTS ARITHMETIC OPERATIONS

In decimal, very large and very small numbers are expressed in scientific notation as follows: 4.69×10^{23} and 1.601×10^{-19}. Binary numbers can also be expressed in this same notation by floating point representation. The floating point representation of a number consists of two parts. The first part represents a signed, fixed point number called the **mantissa.** The second part designates the position of the decimal (or binary) point and is called the **exponent.** The fixed point mantissa may be a fraction or an integer. The number of bits required to express the exponent and mantissa are determined by the accuracy desired from the computing system as well as its capability to handle such numbers. For example, the decimal number + 6132.789 is represented in floating point as follows:

$$\underbrace{\begin{matrix}\text{sign}\\ 0 \quad .6132789\end{matrix}}_{\text{mantissa}} \qquad \underbrace{\begin{matrix}\text{sign}\\ 0 \quad 04\end{matrix}}_{\text{exponent}}$$

NOTES

The mantissa has a 0 in the leftmost position to denote a plus. The mantissa here is considered to be a fixed point fraction. So, the decimal point is assumed to be at the left of the MSB. The decimal mantissa, when stored in a register requires at least 29 flip-flops : four flip-flops for each BCD digit and one for the sign. The decimal part is not physically indicated in the register; it is only assumed to be there. The exponent contains the decimal number + 04 (in BCD), to indicate the actual position of the decimal point, is four decimal positions to the right of the assumed decimal point. This representation is equivalent to the number expressed as a fraction times 10 ton exponent, that is, $+.6132789 \times 10^{+04}$. Due to this analogy, the mantissa is sometimes called the **fraction part**.

Consider the following decimal numbers to understand floating point notation.

(a) 42300
(b) 369.4202
(c) 0.00385
(d) 643.15

The above numbers can be written in floating point representation as follows:

(a) $42300 = 423 \times 10^2$
(b) $369.4202 = .3694202 \times 10^3$
(c) $0.00385 = 385 \times 10^{-5}$
(d) $643.15 = 64315 \times 10^{-2}$

Here the first or the integer part is known as **mantissa.** The mantissa is multiplied by some power of 10 and this power is known as the **exponent.**

Consider, for example, a computer that assumes integer representation for the mantissa and radix 8 for the numbers. The octal number + 36.754 = 36754×8^{-3}, in its floating point representation will look like this:

When this number is represented in a register, in its binary-coded form, the actual value of the register becomes

0011 110 111 101 100 1 000 011

The register needs 23 flip-flops. The circuits that operate on such data must recognise the flip-flops assigned to the bits of the mantissa and exponent and their associated signs. Note that if the exponent is increased by one (to –2) the actual point of the mantissa is shifted to the right by three bits (one octal digit).

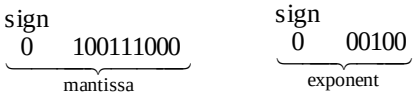
Floating point is always interpreted to represent a number in the following form:

$$m \times r^e$$

Only the mantissa m and the exponent e are physically represented in the register. The radix r and the radix-point position of the mantissa are always assumed.

NOTES

A floating point binary number is represented in a similar manner except that the radix assumed is 2. For example, the number + 1001.11 is represented in a 16-bit register as follows:



The mantissa occupies 10 bits and the exponent 6 bits. The mantissa is assumed to be a fixed point representation. If the mantissa is assumed to be an integer, the exponent will be 1 00101 (–5).

A floating point number is said to be normalised if the most significant position of the mantissa contains a non zero digit. For example, the mantissa 035 is not normalised but 350 is. When 350 is represented in BCD, it becomes 0011 0101 0000 and although two 0's seem to be present in the two most significant positions, the mantissa is normalised. Because the bits represent a decimal number, not a binary number, and decimal numbers in BCD must be taken in groups of four bits, the first digit is 3 and is non zero.

When the mantissa is normalized, it has no leading zeros and therefore contains the maximum possible number of significant digits. Consider, for example, a register that can accommodate a mantissa of five decimal digits and a sign. The number + $0.35748 \times 10^2 = 35.748$ is normalised because the mantissa has a non zero digit 3 in its most significant position. This number can be represented in an unnormalised form as + $.00357 \times 10^4 = 35.7$. This unnormalized number contains two most significant zeros and therefore the mantissa can accommodate only three significant digits. The two least significant digits, 4 and 8, that were accommodated in the normalised form, have no form *n* the unnormalised form because the register can only accommodate five digits.

Arithmetic operations with floating point numbers are more complicated than arithmetic operations with fixed point numbers and their execution takes longer and requires more complex hardware. However, floating point representation is a must for scientific computations because of the scaling problems involved with fixed point computations. Many computers and all electronic calculators have built-in capability of performing floating point arithmetic operations. Computers that do not have hardware for floating point computations have a set of subroutines to help the user program his scientific problems with floating point numbers.

Example 1.2: Determine the number of bits required to represent in floating point notation the exponent for decimal numbers in the range of 10^{+86}.

Solution: Let n be the required number of bits to represent the number 10^{+86}.

$$\therefore \quad 2^n = 10^{86}$$

$$n \log 2 = 86$$

$$\therefore \quad n = 86/\log 2 = \frac{86}{0.3010} = 285.7$$

$$\therefore \quad 10^{\pm 86} = 2^{\pm 285.7}.$$

The exponent ± 285 can be represented by a 10-bit binary word. It has a range of exponent (+ 511 to – 512).

CHECK YOUR PROGRESS

3. What are integers?

NOTES

1.5 CHARACTER REPRESENTATION

Binary data is not the only data handled by the computer. We also need to process alphanumeric data like alphabets (upper and lower case), digits (0 to 9) and special characters like + – * / ( ) space or blank etc. These also must be internally represented as bits.

1.5.1 ASCII

The abbreviation ASCII stands for the *American Standard Code for Information Interchange*. The ASCII code is a 7-bit code used in transferring coded information from keyboards to computer displays and printers. It is used to represent numbers, letters, punctuation marks as well as control characters. For example, the letter *A* is represented by 100 0001. Several computer manufacturers have adopted it as their computers' internal code. This code uses 7 digits to represent 128 characters. Now an advanced ASCII is used having 8 bit character representation code allowing for 256 different characters. This representation is being used in micro computers.

Let us look at the encoding method. Table 1.4 presents shows the bit combinations required for each character.

Table 1.4 *Bit Combinations for Each Character*

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20		!	"	#	$	%	&	‘	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Thus, to code a text string 'Hello' in ASCII using hexadecimal digits:

H	e	l	l	o	.
48	65	6C	6C	6F	2E

The string is represented by the byte sequence 48 65 6C 6C 6F 2E.

NOTES

1.5.2 EBCDIC

The abbreviation EBCDIC stands for the *Extended Binary Coded Decimal Interchange Code*. It is an 8-bit code in which the decimal digits are represented by the 8421 BCD code preceded by 1111.

The major drawback with the BCD code is that it allows only 64 different characters to be represented. This is not sufficient to provide for decimal numbers (10), lowercase letters (26), uppercase letters (26), and a fairly large number of special characters (28 plus).

The BCD code was therefore extended from a 6 bit to an 8 bit code. The added 2 bits are used as additional zone bits, expanding the zone bits to 4. This resulting code is called the Extended Binary Coded Decimal Interchange Code (EBCDIC). Using the EBCDIC it is possible to represent 2^8 or 256 characters. This takes care of the character requirement along with a large quantity of printable and several non-printable control characters (movement of the cursor on the screen, vertical spacing on printer etc.).

Since EBCDIC is an 8 bit code, it can easily be divided into two 4 bit groups. Each of these groups can be represented by one hexadecimal digit (explained earlier in this unit). Thus, hexadecimal number system is used as a notation for memory dump by computers that use EBCDIC for internal representation of characters.

Developed by IBM, EBCDIC code is used in most IBM models and many other computers.

CHECK YOUR PROGRESS

4. What is the ASCII code?
5. What is the EBCDIC code?

1.6 DEFINITION OF COMPUTER

A computer is defined as a machine for carrying out mathematical and non-mathematical operations. It is an electronic machine with some mechanical facilities and is used mainly for data processing. It is based on complex technology but works on the simple principle that after processing a given input it will provide an output, as shown in Figure 1.6.

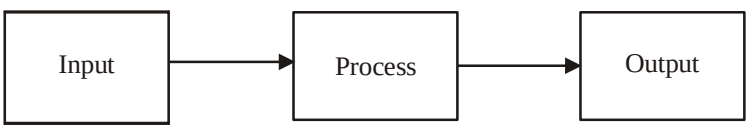


Figure 1.6 *The Working Principle of a Computer*

Computers are used in various fields such as the following:

- Scientific calculations
- Commercial and business data processing
- Air traffic control
- Space guidance
- Education

NOTES

1.7 FUNCTIONAL UNITS OF A COMPUTER

In its simplest form, a computer consists of five functionally independent components: input, output, memory, arithmetic and logic unit, and control unit as shown in Figure 1.7.

A computer accepts information in the form of a program and data through its input unit, which can be an electromechanical device such as a keyboard or from other computers over digital communication lines. The information received by the computer is either stored in the memory for later reference or used immediately by the arithmetic and logic unit (ALU) for performing the desired operations. Finally, the processed information in the form of results is displayed through an output unit. The control unit controls all activities taking place inside the computer. The ALU unit along with the control unit are collectively known as the central processing unit (CPU) or processor; and the input and output units are collectively known as the Input–Output (I/O) unit.

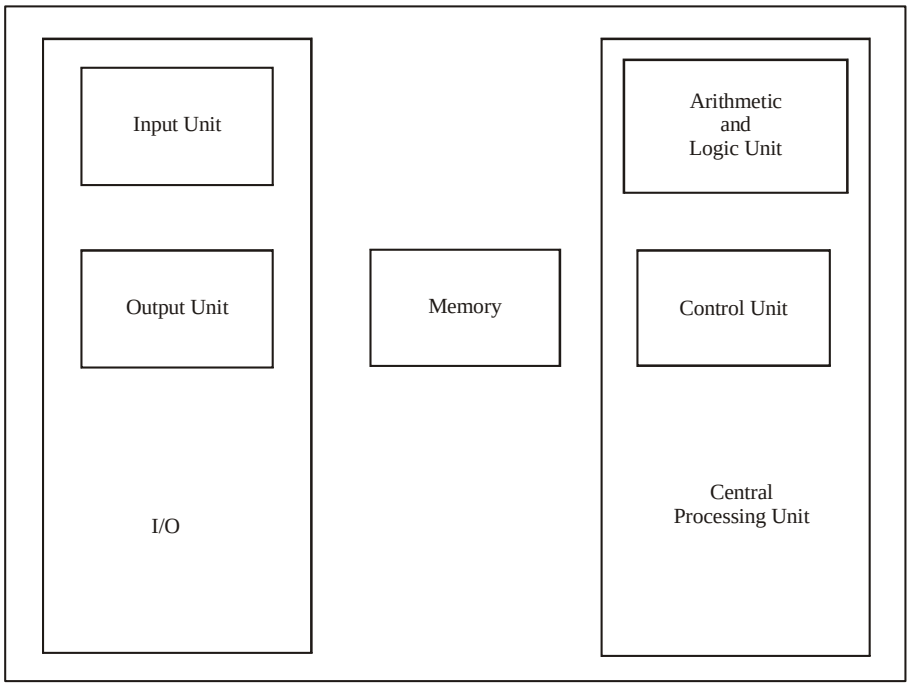


Figure 1.7 *Functional Units of a Computer*

NOTES

- **Input unit:** A computer accepts input in coded form through an input unit. The keyboard is an input devices. Whenever a key is pressed, the binary code of the corresponding letter or digit is transferred to the memory unit or processor. Other types of input devices are mouse, punch car, joysticks, etc.
- **Memory unit:** The task of the memory unit is to safely store programs as well as input, output and intermediate data. The two different classes of memory are primary and secondary storage. Primary storage is a very fast memory and contains a large number of semiconductor cells capable of storing one bit of information. A group (of fixed size) of these cells is referred to as words and the number of bits in each word is referred to as word length, which typically ranges from 16 to 64 bits. When the memory is accessed, usually one word of data is read or written.
- **Processor unit:** The processor unit performs arithmetic and other data processing tasks as specified by a program.
- **Control unit:** It oversees the flow of data among the other units. The control unit retrieves the instructions from a program (one by one), which are safely kept in the memory. For each instruction, the control unit tells the processor to execute the operation marked by the instruction. The control unit supervises the program instructions and the processor manipulates the data as specified by the programs.
- **Output unit:** The output unit receives the result of the computation, which is displayed on the screen or printed on paper using a printer.

CHECK YOUR PROGRESS

6. What are some of the fields where computers are used?

1.8 INPUT UNIT

Both program and data need to be in the computer system before any kind of operation can be done. Program refers to the set of instructions which the computer has to carry out, and data is the information on which these instructions are to operate. If the task is to rearrange a list of telephone subscribers in alphabetical order, the sequence of instructions that will guide the computer through this operation is the program, while the list of names to be sorted is the data.

The process of transferring data and instructions from the external environment into the computer system is performed by the Input Unit. Instructions and data enter the input unit through the particular input device used (keyboard, scanner, card reader, etc.). These instructions and data are then converted into binary codes (computer-acceptable form) and sent to the computer system for further processing.

1.9 OUTPUT UNIT

NOTES

Since the computers understand, process data and return the output in the binary form. The basic function of the output unit therefore is to convert these results into human-readable form before providing the output through various output devices like terminals and printers.

Secondary Storage: The storage capacity of the primary memory of the computer is limited. Often, it is necessary to store large amounts of data. So, additional memory, called secondary storage or auxiliary memory, is used in most computer systems.

Secondary storage is storage other than the primary storage. These are peripheral devices connected to and controlled by the computer to enable permanent storage of data and programs. Typically, hardware devices like magnetic tapes and magnetic disks fall in this category.

1.10 I/O DEVICES

Input/Output devices are required for users to communicate with the computer. Input devices allow you to input data into your computer. In simplest terms, they bring information INTO the computer. Output devices on the other hand display the results of your computer's calculations. They bring information OUT of a computer.

Examples of input/output devices:

Input Devices	Output Devices
Keyboard	Monitor
Mouse	Liquid Crystal Display (LCD)
Scanner	Printer
Touch Screen	Speaker
Light Pen	Plotter

Input Devices

Keyboard

A text based input device that is used to type in letters, numbers and other characters (Figure 1.8).

Figure 1.8 *Keyboard*

NOTES

Enter	To execute a command or a program. Similar to the 'return' key of a typewriter.
Backspace	To remove the character directly to the left of your cursor.
Delete	To remove the character directly to the right of your cursor.
Ctrl (Control key)	Called a helper key because it is always used in combination with other keys to perform specific actions.
Alt (Alternate key)	Called a helper key because it is always used in combination with other keys to perform specific actions.
Shift	Used to type capital letters when pressed with an alphabetic key. Also used to type the symbols located on the upper side of number keys [0-9].
Tab	Used to insert indentation into a document. Jumps from box to box when entering data in a form.
Arrow Keys	Used to move the cursor in top/bottom/left/ right directions.
Caps Lock	Used to make the alphabetic characters to the upper case.
Esc (Escape key)	Used to cancel and abort programs.
Home/End	Home key is used to shift the cursor to the beginning of a line. End key is used to bring the cursor to the end of a line.
Page Up	Used to move the cursor up one screen length. It will not move the cursor to the 1st page if you are in the 2nd page.
Page Down	Used to move the cursor down one screen length. It will not move the cursor to the 2nd page if you are in the 1st page.
F Keys	Called function keys which are located at the top of the keyboard. Their function depends upon the type of the program being used.
Num Lock	The numeric keypad located at the extreme right of the keyboard is activated when the **Num Lock** key is turned on.

NOTES

Mouse

Mouse is used to move the cursor on your computer screen; to give instructions to your computer and to run programs and applications. It can be used to select menu commands, move icons, size windows, start programs, close windows, etc.

Figure 1.9 *Mouse*

Mouse actions:

Click	It is used to select an item by the mouse.
Right Click	It means pressing and releasing the right mouse button. It is used to display a set of commands.
Double Click	It means clicking one of the mouse button twice in a quick succession. There should not be any time gap between the two pressing actions. It is used to open a document or a program.
Drag & Drop	Instead of cutting and pasting a document/program, you may drag and drop it to the place you want in the computer. To do it, select the item you want to move on the screen and press and hold down the left button of the mouse. Now without releasing the button, drag the cursor where you need to put the document/program and then release the button.

Scanner

It is used to input pictures and images into your computer (Figure 1.10). It converts images to digital form so that it can be fed into the computer.

Figure 1.10 *Scanner*

Touch Screen

It allows the user to operate a computer by simply touching the display screen. Example of a touch screen includes, ATM at a bank.

NOTES

Light Pen

It uses a light sensor device to select objects on a display screen. It is similar to a mouse except that with a light pen you can move the pointer and select any object on the screen by directly pointing to the object with the light pen.

Output Devices

Monitor

It is used to display information, programs and applications in a computer. It is also called visual display screen or monitor (Figure 1.11). Like televisions, monitors come in different sizes.

Figure 1.11 *Monitor*

Liquid crystal display (LCD): It is smaller, and lighter as compared to a monitor. It is mostly used with portable computers (Laptops).

Printer

It is used to create a hard copy of the files stored in a computer (Figure 1.12). Printer's have two basic qualities: resolution and print speed. Print resolution is measured as the number of dots per inch (dpi). Print speed is typically measured in pages per minute (ppm). Printers are generally of four types, viz. Impact Line Printers, Dot Matrix Printers, Ink Jet Printers and Laser Printers. Laser printers are superior. They are very fast, easy to use and produce high quality output.

Figure 1.12 *Printer*

Speaker

It is used to produce music or speech from programs (Figure 1.13). A speaker port (a port is a connector in your computer wherein you can connect an external device) allows to connect speaker to the computer. Speakers can be built into the computer or can be attached separately.

Figure 1.13 *Speakers*

Plotter

A plotter interprets commands from the computer to make line drawings on a paper using multicoloured automated pens. Plotters are used for drawing bar charts, graphs, maps, etc. on rolls of papers.

1.11 PRIMARY AND SECONDARY MEMORIES

Main Memory

The memory unit, known as the main memory, directly interacts with the CPU. It is mainly utilized to store programs and data at the time of operating the computer. It is a comparatively fast and large memory. The main memory can be classified into two categories, which are explained in the following sections:

Random Access Memory (RAM)

The memory system in the computer that is easily read from and written to by the processor is the RAM. In the RAM, any address may be accessed at any time, i.e., any memory location can be accessed in a random manner without going through any other memory location. The access search time for all the memory locations is the same.

Thus, RAM is the main memory of a computer system. Its main objective is to safely store applications and information that are currently being used by the processor. The operating system directs the use of the RAM by taking different decisions, such as when data should be stored in the RAM, at what memory locations the data should be stored, and so on. The RAM is a very fast memory, both for reading and writing information. The information written in it is retained as long as the power supply is on. All information stored in the RAM is lost when the power supply is switched off.

The two main classes of RAM are Static RAM (SRAM) and Dynamic RAM (DRAM).

Static RAM (SRAM)

A static RAM is made from an array of flip-flops, where each flip-flop maintains a single bit of data within a single memory address or location.

NOTES

A static RAM retains its data without external refresh as long as electricity is available. The other features of SRAM are as follows:

- It is a type of semiconductor memory.
- It does not require any external refresh to keep its data intact.
- It is used for high-speed registers, caches and small memory banks such as router buffers.
- It has access times in the range of 10 to 30 nanoseconds and, hence, allows for very fast access.
- It is very expensive.

Dynamic RAM (DRAM)

A dynamic RAM keeps its data only if it is accessed by special logic–called refresh circuit–on a continuous basis. This circuitry reads the data of each memory cell very fast, irrespective of whether the memory cell is being used at that time by the computer or not. The memory cells are constructed in such a way that the reading action itself refreshes the contents of the memory. If not done on a regular basis, the DRAM will lose its contents, even if it has uninterrupted power supply. Because of this refreshing action, the memory is called dynamic. The other features of DRAM are as follows:

- It is the most general type of memory. It is used in all PCs for their main memory system
- It has a much higher capacity
- It is cheaper than SRAM
- Due to the refresh circuitry, it is slower than SRAM

Read Only Memory (ROM)

The Read Only Memory, which is a secure memory, is not affected by any interruption in the power supply. It is a non-volatile memory, i.e., information stored in it is not lost even if the power supply goes off. It is used for permanent storage of information and possesses random access properties.

The essential purpose of the ROM is to store the Basic Input/Output System (BIOS) of the computer. The BIOS commands the processors to access its resources on receiving power supply to the system. The other use of ROM is the code for embedded systems.

The different types of ROMs are now discussed.

Programmable Read Only Memory (PROM)

Data is written into a ROM at the time of manufacture. However, a user can program the contents with a special PROM programmer. PROM provides flexible and economical storage for fixed programs and data.

Erasable Programmable Read only Memory (EPROM)

This lets the programmer erase the contents of the ROM and reprogram it. The contents of EPROM cells can be erased with ultraviolet light using an EPROM programmer. This type of ROM provides more flexibility than a ROM during the development of digital systems. Since these are able to retain the stored information for a longer duration, any change can be easily made.

Electrically Erasable Programmable Read Only Memory (EEPROM)

In this type of ROM, the contents of the cell can be erased electrically by applying a high voltage. The EEPROM need not be removed physically for reprogramming.

Organization of RAM and ROM Chips

A RAM chip is best suited for communication with the CPU if it has one or more control lines to select the chip when needed. It has a bi-directional data bus that allows the transfer of data from the memory to the CPU at the time of a read operation or from the CPU to the memory at the time of a write operation. This bi-directional bus can be constructed using a three-state buffer which has the following three possible states:

- Logic 1
- Logic 0
- High impedance

The high impedance state behaves like an open circuit, which means the output does not carry any signal. It leads to very high resistance and, hence, no current flows.

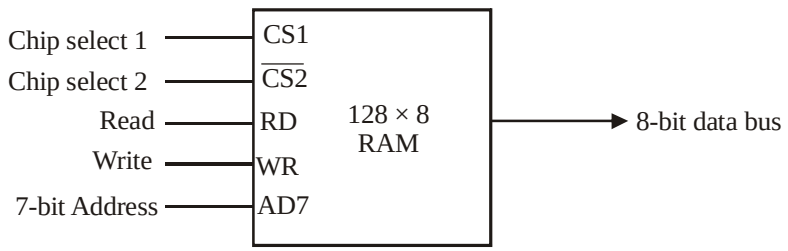


Figure 1.14 *Block Diagram of a RAM Chip*

Figure 1.14 depicts the block diagram of a RAM chip. The capacity of its memory is 128 words of eight bits each. This requires a 7-bit address and 8-bit bi-directional data bus.

RD and WR are the read and write inputs that specify the memory operations during read and write, respectively. Two chips select (CS) control inputs for enabling a particular chip only when it is selected by the microprocessor. The operation of the RAM chip will be according to the functions given in Table 1.5. The unit is in operation only when *CS*1 = 1 and *CS*2 = 0. The bar on top of the second chip select (see Figure 1.15) indicates that this input is enabled when it is equal to 0.

NOTES

Table 1.5 *Function Table*

CS1	CS2	RD	WR	Memory Function	State of Data Bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

Thus, if the chips select inputs that are not enabled or if they are enabled but read and write lines are not enabled, the memory is inhibited and its data bus is in high impedance. When chip select inputs are enabled, i.e. $CS1 = 1$ and $CS2 = 0$, the memory can be in the read or write mode. When the write WR input is enabled, a byte is transferred from the data bus to the memory location specified by the address lines. When the read RD input is enabled, a byte from the specified memory location is placed in the data bus by the address lines.

A ROM chip is organized in the same way as a RAM chip. The block diagram of a ROM chip is shown in Figure 1.15.

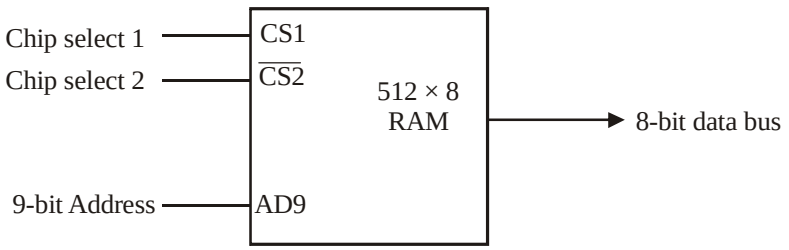


Figure 1.15 *Block Diagram of a ROM Chip*

The two chip select lines must be $CS1 = 1$ and $CS2 = 0$ for the unit to be operational. Otherwise, the data bus is in high impedance. There is no need for read and write input control because the unit can only read. Thus, when the chip is selected, the bytes selected by the address line appear to be in the data bus.

Memory Address Map

The memory address map is a pictorial representation of the assigned address space for each chip in a system.

The interconnection between the memory and processor is established from information on the size of memory required and the type of RAM and ROM chips available. RAM and ROM chips are available in a variety of sizes. If a memory needed for computer is larger than the capacity of one chip, it is necessary to combine a number of chips to get the required memory size. If the required size of the memory is $M \times N$ and if the chip capacity is $m \times n$, then the number of chips required, can be calculated as

$$k = \frac{M \times N}{m \times n}$$

Suppose, a computer system needs 512 bytes of RAM and 512 bytes of ROM. The capacity of a RAM chip is 128 × 8 and that of a ROM is 512 × 8. Hence, the number of RAM chips required will be

NOTES

$$k = \frac{512 \times 8}{128 \times 8} = 4 \text{ RAM chips}$$

One ROM chip will be required by the computer system. The memory address map for the system is given in Table 1.6, which consists of three columns. The first column specifies whether a RAM or a ROM chip is used. The next column specifies a range of hexadecimal addresses for each chip. The third column lists the address bus lines.

Table 1.6 *Memory Address Map*

Component	Hexadecimal address	Address Bus									
		10	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
RAM 1	0000–007F	0	0	0	×	×	×	×	×	×	×
RAM 2	0080–00FF	0	0	1	×	×	×	×	×	×	×
RAM 3	0100–017F	0	1	0	×	×	×	×	×	×	×
RAM 4	0180–01FF	0	1	1	×	×	×	×	×	×	×
ROM	0200–03FF	1	×	×	×	×	×	×	×	×	×

Table 1.6 lists only 10 lines for the address bus although it actually consists of 16 lines. These six lines are assumed to be zero. RAM chips have 128 bytes and need seven address lines, which are common to all four RAM chips. ROM chips have 512 bytes and need nine address lines. Thus, ×s are assigned to the low-order bus lines—lines 1 to 7 for the RAM chip and lines 1 through 9 for the ROM chip—× to represent a binary number, which is a combination of all possible 0 and 1 values. Also, there must be a way to distinguish between the four RAM chips. Lines 8 and 9 are used for this purpose. If these lines are 00, it is the RAM1 chip; if it is 01, it is the RAM2 chip; if it is 10, then it is RAM3 chip and if it is 11, it is the RAM4 chip. Also, the distinction between RAM and ROM is required. This distinction is made with the help of line 10. When line 10 is 0, the CPU selects one of the RAM chips, and when line 10 is 1, the CPU selects the ROM chip.

Memory Connection to CPU

The CPU connects the RAM and ROM chips with the data and address buses. Low-order lines inside the address bus choose the byte within the chip and other lines choose a particular chip through its chip chosen lines. The memory chip connection to the CPU is shown in Figure 1.16. Seven low-order bits of the address bus are connected directly to each RAM chip for selecting one of 128 possible bytes.

NOTES

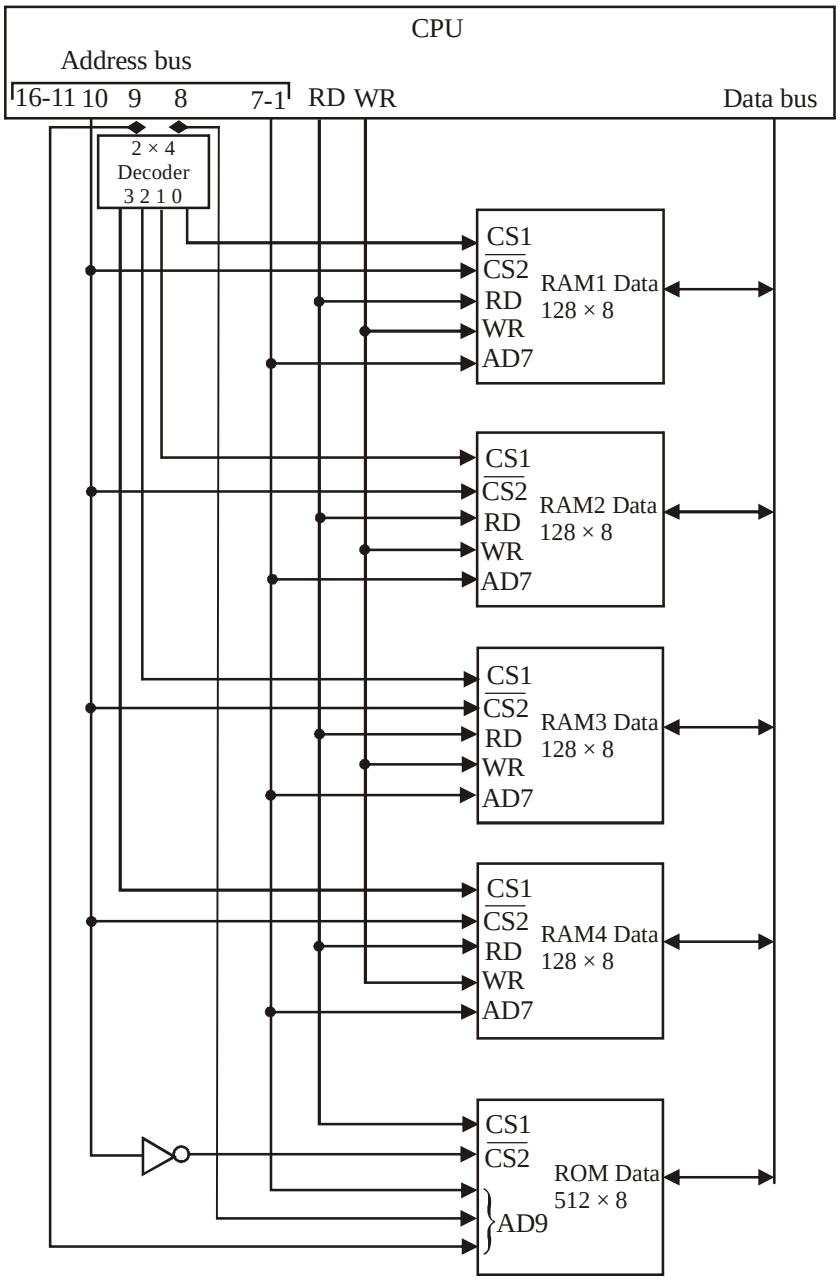


Figure 1.16 *Memory Connection to the CPU*

Lines 8 and 9 are connected as inputs to a 2 × 4 decoder, whose output are connected to the CS1 input of each RAM chip. Thus, when lines 8 and 9 are 00, the first RAM chip is selected. The RD and WR outputs from the CPU are connected to the inputs of each RAM chip. Line 10 is connected to $\overline{CS2}$ of each RAM chip, and with the help of an inverter to a ROM chip. The other chip select input in the ROM is connected to the RD control line. Address bus lines 1 to 9 are applied to the input address of ROM. The data bus lines of both RAM and ROM are linked to the data bus of the CPU where the RAM can transfer information in both directions, whereas the ROM has only one output capability.

Auxiliary Memory

Storage devices which help in backup storage are called auxiliary memory.

RAM is a volatile memory and, thus, a permanent storage medium is required in a computer system. Auxiliary memory devices are used in a computer system for permanent storage of information and hence these are the devices that provide backup storage. They are used for storing system programs, large data files and other backup information. The auxiliary memory has a large storage capacity and is relatively inexpensive, but has low access speed as compared to the main memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Now, optical disks are also used as auxiliary memory.

NOTES

Magnetic Disk

Magnetic disks are circular metal plates coated with a magnetized material on both sides. Several disks are stacked on a spindle one below the other with read/write heads to make a disk pack. The disk drive consists of a motor and all the disks rotate together at very high speed. Information is stored on the surface of a disk along a concentric set of rings called tracks. These tracks are divided into sections called sectors. A cylinder is a pair of corresponding tracks in every surface of a disk pack. Disk drives are connected to a disk controller.

Thus, if a disk pack has *n* plates, there will be 2*n* surfaces; hence, the number of tracks per cylinder is 2*n*. The minimum quantity of information which can be stored, is a sector. If the number of bytes to be stored in a sector is less than the capacity of the sector, the rest of the sector is padded with the last type recorded. Figure 1.17 shows a magnetic disk memory.

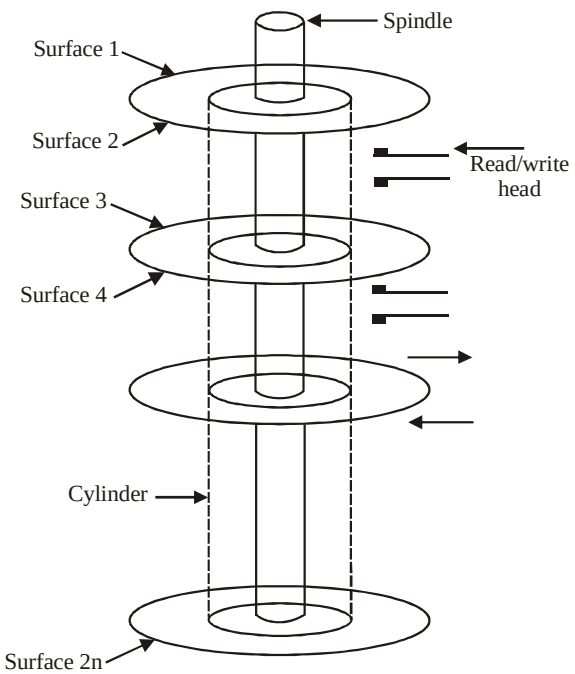


Figure 1.17 *Magnetic Disk*

NOTES

The subdivision of a disk surface into tracks and sectors is shown in Figure 1.18.

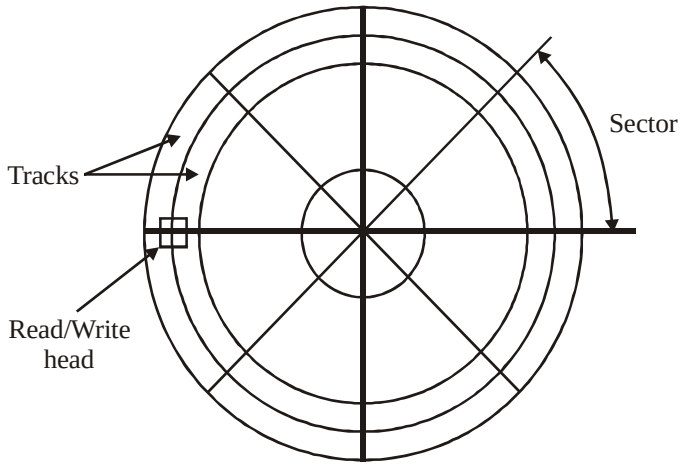


Figure 1.18 *Surface of a Disk*

Let *s* bytes be stored per sector; *p* sectors exist per track, *t* tracks per surface and *m* surfaces. Then, the capacity of the disk will be defined as:

Capacity = $m \times t \times p \times s$ bytes

If *d* is the diameter of the disk, the density of recording would be:

$$\text{Density} = \frac{(p \times s)}{(\pi \times d)} = \text{bytes/inch}$$

A set of disk drives is linked to a disk controller; the latter agrees to the instructions and keeps ready the read/write heads for reading or writing. When the read/write instruction is accepted by the disk controller, the controller first positions the arm so that the read/write head reaches the appropriate cylinder. An appropriate *seek time* (*Ts*) is the time needed to reach the proper cylinder. The maximum *Ts* is the time needed by the head to reach the innermost cylinder from the outermost cylinder or vice versa. The minimum *Ts* will be 0 if the head is already positioned on the appropriate cylinder. Once the head is positioned on the cylinder, there is further delay because the read/write head has to be positioned on the appropriate sector. This is rotational delay, also known as *latency time* (*Tl*). The average rotational delay equals half the time taken by the disk to complete one rotation.

Floppy Disk

A floppy disk, also known as a diskette, is a very convenient bulk storage device and can be taken out of the computer. It is of 5.25" or 3.5" size, the latter size being more common. It is contained in a rigid plastic case. The read/write heads of the disk drive can write or read information from both sides of the disk. The storage of data is in magnetic form, similar to that of the hard disk. The 3.5" floppy disk has a capacity of storage up to 1.44 Mbytes. It has a hole in the centre for mounting it on the drive. Data on the floppy disk is organized during the formatting

process. The disk is also organized into sectors and tracks. The 3.5" high density disk has 80 concentric circles called tracks and each track is divided into 18 sectors. Tracks and circles exist on both sides of the disk. Each sector can hold 512 bytes of data plus other information like address, etc. It is a cheap read/write bulk storage device.

Magnetic Tape

A magnetic disk is used by almost all computer systems as a permanent storage device. It is still the accepted low-cost magnetic storage medium, and is primarily used for backup storage purposes. The digital audio tape (DAT) is the normal backup magnetic tape tool used these days. A standard cartridge-size cassette tape offers about 1.2 GB of storage. These magnetic tape memories are similar to that of audio tape recorders.

A magnetic tape drive consists of two spools on which the tape is wound. Between the two spools, there is a set of nine magnetic heads to write and read information on the tape. The nine heads operate independently and record information on nine parallel tracks, all parallel to the edge of the tape. Eight tracks are used to record a byte of data and the ninth track is used to record a parity bit for each byte. The standard width of the tape is half an inch. The number of bits per inch (bpi) is known as *recording density*.

Normally, when data is recorded on to a tape, a block of data is recorded and then a gap is left and then another block is recorded, and so on. This gap is known as inter-block gap (IBG). The blocks are normally 10 times as long as IBG. The beginning of the tape (BOT) is indicated through a metal foil known as marker, and the end of the tape (EOT) is also indicated through a metal foil known as end of tape marker.

The data on the tape is arranged as blocks and cannot be addressed. It can only be retrieved sequentially in the same order in which it is written. Thus, if a desired record is at the end of the tape, earlier records have to be read before it is reached and hence the access time is very high as compared to magnetic disks.

Optical Disk

This (optical disk) storage technology has the benefit of high-volume economical storage coupled with slower times than magnetic disk storage.

Compact Disk-Read Only Memory (CD-ROM)

The CD-ROM optical drives are employed for storage of data that is circulated for read-only use. A single CD-ROM has the capacity to hold around 800 MB of data. This media can be used when software and huge reports are to be circulated to a large number of users. As compared to floppy disks or tapes, CD-ROM is more dependable for circulation. Nowadays, almost all software and documentations are circulated only on CD-ROM.

NOTES

In a CD-Rom, data is stored uniformly across the disk in parts made of equal size. So, as you go towards the outer surface of the disk, the data stored on a track increases. Thus, CD-ROMs are rotated at changing speeds for reading.

Information in a CD-ROM is written by creating pits on the disk surface by shining a laser beam. As the disk rotates, the laser beam traces out a continuous spiral. When 1 is to be written on the disk, a circular pit of around 0.8-micrometer diameter is created by the sharply focussed beam and no pit is created if a zero is to be written. The prerecorded information on a CD-ROM is read with the help of a CD-ROM reader, which uses a laser beam for reading. For this, the CD-ROM disk is inserted into a slot of CD drive. Then, a motor rotates the disk. A laser head moves in and out of the specified position. As the disk rotates, the head senses pits and converted to 1s and 0s by the electronic interface and sent to the computer. Figure 1.19 depicts a CD-ROM.

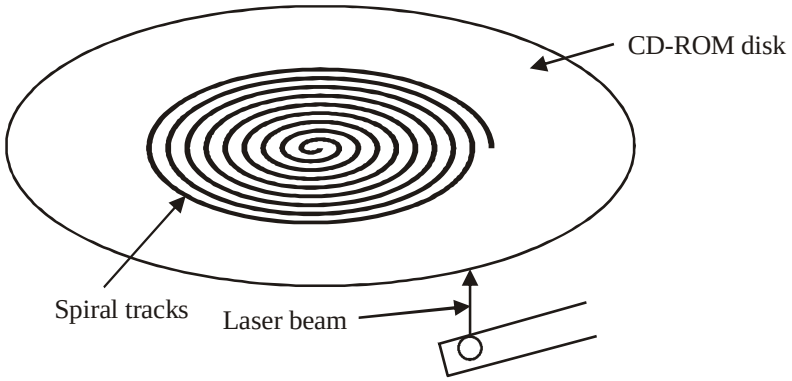


Figure 1.19 *Tracks on Disk Surface*

The speed of the disk is indicated by *nx*, where *n* is an integer indicating the factor by which the original nominal speed of 150 KB/S is multiplied. Thus, a 52X CD-ROM disk speed will be 52 × 150 = 7800 KB/S. The CD-ROM has a buffer size of 256 kilobytes to keep data temporarily. A small computer system interface (SCSI) adapter connects it to the computer system.

The following are the major benefits of CD-ROM:

- It has the capacity to store large amounts of data/information.
- Its mass replication is fast and inexpensive.
- It has removable disks.

The following are the demerits of CD-ROM:

- It is read-only and hence cannot be updated.
- Access time is longer than that of a magnetic disk.

Erasable Optical Disk

A recent development in optical disk technology is the erasable optical disk. The erasable optical disk can be a substitute for the standard magnetic disk, subject to the condition that the speed of access is not essential and the quantity of data

stored is large. The optical disk is used for multimedia, image and high-volume, low-activity backup storage. Data in these disks can be changed repeatedly as in a magnetic disk. The erasable optical disk is portable, highly reliable and has a longer life. It uses a format that makes semi-random access feasible.

Comparative Characteristics of Secondary Memories

Secondary memory is also known as storage media, It is the slowest and cheapest mode of memory than other storage media. It is not processed directly by the CPU. First, the content of secondary memory is copied into primary storage (RAM). The devices of secondary memory include magnetic disks, such as hard drives and floppy drives, optical disks, such as CDs and CD ROMs and magnetic tapes. They are known as the first form of secondary memory. Backing storage (also known as secondary memory) is used to store data that is not required all the time in the primary memory of the CPU. Often, such data is very large and cannot be held altogether in the primary memory. For these reasons, programs and data files are stored in secondary memory rather than on the primary memory. The data stored on the backing storage can be accessed in primarily two ways: (1) serial access and (2) direct access. For instance, the payroll program of a company serially accesses the data of employees, i.e. to read one record after another from a data file. This is known as serial access. To understand the direct access, let us take the example of a supermarket in which the item details are stored in a data file. Now, to access the details of an item, the computer locates the item code in the file and picks up the corresponding record. This is also known as random access. Direct access is only made possible by the use of an index.

The most commonly used secondary memory media are hard disks and CD-ROM. These media can be used to store operating system software, software application packages, data files, etc. Use of secondary memory is to overcome some of the limitations of primary memory. Primary memory is volatile, which means that its contents are gone the moment you switch off the CPU. This is not the case with secondary memory; hence it is used to hold data for a long term. The data remains in the secondary memory unless it is overwritten or deleted by the user. The storage capacity of the secondary memory is another important feature that differentiates it from primary memory. There is actually no limit to the capacity of secondary memory; it is only limited by technological advancement. It is used for mass storage. The data on secondary storage can be moved and, thus, shared by multiple users on their respective computers. Even after these advantages, secondary memory is not as fast as primary memory due to the use of slower interfaces such as Integrated Drive Electronics (IDE) and Small Computer System Interface (SCSI). A distinct feature of secondary memory is that the data stored in secondary memory cannot be accessed by the CPU directly. The computer system uses its input/output channels to transfer data from secondary memory to primary memory before it can be used.

Secondary memory forms a major part of today's PCs and hence its cost. Every modern computer comes with some form of removable media, such as CD

NOTES

drives, floppy drives, etc; however, the use of floppy drives has become almost obsolete. Backup devices also include magnetic tape devices, which are not very commonplace. These devices use Digital Audio Tape (DAT) and Digital Linear Tape (DLT) formats, which are slow and expensive and hold very less amount of data as compared to hard disks. The rapid advancement in computer processing power puts increasing pressure on computer scientists and engineers to bring a similar advancement in the secondary memory devices.

Secondary memory is intended for storing the data and information that is needed in future even if computer is switched off. Table 1.7 lists the comparative characteristics of secondary memory.

Table 1.7 *Characteristics of Secondary Memory*

Secondary Storage Media	Characteristics
Flash Memory	This type of memory belongs to low power, non-volatile, re-writable. It can carry programs and data into the USB. It transfers digital photos, large documents and back ups of important data. Its size is about 128 MB.
Disk	It is portable and swappable. It holds large capacity. It can be recovered fast.
Optical Media	Its memory capacity is larger than flash memory but smaller than hard disk. It is re-writable, rugged and portable. It has good back up medium than other storage media. Its typical size is 700 MB and DVD size is about 2.7 GB.
Tapes	It has larger capacity than all other types of media. It has some exception especially about hard disks. It is relatively cheap to manufacture. It has also good backup medium for networks. Its size is about 500GB.

CHECK YOUR PROGRESS

7. What are the two types of RAM?
8. What is the difference between a Static RAM and a Dynamic RAM?
9. What is auxiliary memory?

1.12 PROM AND EPROM

Read only memory (ROM) is a storage class specifically used in computers and electronic devices. The ROM memory is also called non-volatile memory because it is not erased even when the system is turned off. This type of memory helps in

NOTES

storing the data required to boot the computer system. The PROM and EPROM are non-volatile solid-state memory types and allow little modification. PROM permits the users to exactly program its contents one time by physically changing its structure applying the high-voltage pulses. The EPROM can be erased and re-programmed any number of times. Programmable read-only memory (PROM) or one-time programmable ROM (OTP) is configured and programmed using a specific device termed as PROM programmer. Characteristically, this device needs high voltages for permanently destroying or creating internal links (fuses or antifuses) within the chip. A PROM can be programmed only once. Erasable programmable read-only memory (EPROM) is erased if exposed to strong ultraviolet light and is again rewritten by applying high voltage. Repetitive exposure to UV light may sooner or later deteriorate an EPROM, but the strength of the majority of EPROM chips may be more than 1000 cycles to erase and reprogram. EPROM chips are frequently recognized by the leading quartz 'window' that permits the entry of UV light. Once the programming is complete, the window is uniquely covered by a label to protect it from any accidental erasure.

Programmable Read Only Memory (PROM)

A PROM is also termed as one-time programmable non-volatile memory (OTP NVM). It is a type of digital memory in which each bit setting is locked using a fuse or antifuse. These PROMs are specifically used to permanently store programs. It is different to a strict ROM because the programming is done once the device is constructed. These memory types are often used in video game consoles, mobile phones, radio-frequency identification (RFID) tags, implantable medical devices, high-definition multimedia interfaces (HDMI), automotive electronics products, etc. The PROM was invented by Wen Tsing Chow in 1956. The word 'burn' is termed as the original patent and was used to refer the programming process of a PROM.

Programming PROM

In a characteristic PROM all the bits read as 1. While programming, to burn a fuse bit changes the bit to read as 0. The memory is only programmed once when the fuses are manufactured using blowing process. It is an irreversible process. During programming the blowing fuse means to open a connection whereas an antifuse means to close a connection.

Erasable Programmable Read Only Memory (EPROM)

An Erasable Programmable Read-Only Memory (EPROM**)** is a memory chip which retains its information even if the power supply is turned off. It is also non-volatile memory. Basically, it is an arrangement of independently programmed floating-gate transistors using a high voltage electronic device. To erase the programming of EPROM, it is exposed to strong ultraviolet light of wavelength of 253.7 nm. Do not leave the chip under exposure for long time because an over-erased EPROM chip will not store data.

NOTES

As it was expensive to make the quartz window one-time programmable (OTP) chips were used in which the die was mounted such that it could not be erased once programmed. A programmed EPROM can retain the information for about ten to twenty years and can also be read any number of times. EPROMs are available in various sizes for different storage capacity as listed in Table 1.8.

Table 1.8 *EPROM Sizes*

EPROM Type	Size (bits)	Size (bytes)	Length (hex)	Last address (hex)
1702, 1702A	2 kbit	256	100	000FF
2704	4 kbit	512	200	001FF
2708	8 kbit	1 KB	400	003FF
2716, 27C16	16 kbit	2 KB	800	007FF
2732, 27C32	32 kbit	4 KB	1000	00FFF
2764, 27C64	64 kbit	8 KB	2000	01FFF
27128, 27C128	128 kbit	16 KB	4000	03FFF
27256, 27C256	256 kbit	32 KB	8000	07FFF
27512, 27C512	512 kbit	64 KB	10000	0FFFF
27C010, 27C100	1 Mbit	128 KB	20000	1FFFF
27C020	2 Mbit	256 KB	40000	3FFFF
27C040	4 Mbit	512 KB	80000	7FFFF
27C080	8 Mbit	1 MB	100000	FFFFF

Configuring an EPROM

An EPROM is an array of columns and rows and the intersection of row and column is called a cell. Each cell specifies a floating gate transistor and a control gate transistor which are separated by means of a fine oxide layer. All the cells of a blank EPROM have value 1. For configuring an EPROM chip one has to modify some cells from 1 to 0.

1.13 SUMMARY

In this unit, you have learned that:

- There are four systems of arithmetic which are used in digital systems – decimal, binary, hexadecimal and octal.
- The study of integers is termed as the number theory. Integers are the natural numbers 0,1,2,3..and their negatives -1, -2, -3….
- In its simplest form, the computer consists of five functionally independent components: input, output, memory, arithmetic and logic unit and control unit.
- The memory unit of the computer is mainly used to store programs and date at the time of operating the computer. Storage devices which help in backup storage are called auxiliary memory.

1.14 KEY TERMS

NOTES

- **Base or radix:** The base or radix of a number can be defined as the number of different digits which can occur in each position in the number system.
- **Binary number system:** A number system that uses only two digits, 0 and 1 is called the binary number system.
- **Octal number system:** A number system that uses eight digits, 0,1,2,3,4,5,6,7.
- **ASCII:** Short for American Standard Code for Information Interchange, it is a 7-bit code used to transfer coded information from keyboards to computer displays and printers.
- **EBCDIC:** Short for Extended Binary Coded Decimal Interchange Code. It is an 8-bit code where the decimal digits are represented by the 8421 BCD code precedeby by 1111.
- **RAM:** Short for Random Access Memory, it is the memory system that is easily read from and written to by the processor.
- **ROM:** Short for Read Only Memory, it is a secure memory where stored information is not lost even if power supply is cut off.
- **Memory address map:** A pictorial representation of the assigned address space for each chip in a system.

1.15 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The four arithmetic systems used in digitial systems are:

   Decimal, Binary, Octal and Hexadecimal.
2. An Octal odometer is a hypothetical device similar to the odometer of a car.
3. Integers are natural numbers which include the numbers 0,1,2,3,… and their negatives –1, –2, –3…..
4. The ASCII code is a 7-bit code used in transferring coded information from keyboards to computer displays and printers. It is used to represent numbers, letters and punctuation marks as well as control characters.
5. The EBCDIC code is an 8-bit code in which the decimal digits are represented by the 8421 BCD (Binary Coded Decimal) code preceded by 1111.
6. Computers are used in scientific calculations, commercial and business data processing, air traffic control and education.
7. Static RAM(SRAM) and Dynamic RAM (DRAM) are the two types of RAMs.

NOTES

8. A static RAM is made from an array of flip-flops, where each flip-flop maintains a single bit of data within a single memory address or location. A dynamic RAM keeps its data only if it is accessed by special logic on a continuous basis.
9. A storage device which helps in backup storage is called auxiliary memory.

1.16 QUESTIONS AND EXERCISES

Short-Answer Questions

1. Differentiate between the four systems of arithmetic used in the digital systems.
2. Write a short note on the input and output devices used by a computer.
3. Write a detailed note on the main memory unit in a CPU.
4. What are the different types of auxiliary memory devices?

Long-Answer Questions

1. Write a detailed note on the hexadecimal number system.
2. What is the use of auxiliary memory? Explain.
3. Explain the difference between magnetic tape and magnetic disk.
4. Explain the difference between PROM and EPROM.

1.17 FURTHER READING

Friedman, Daniel P. *Essentials of Programming Languages*. Boston, MA: MIT Press.

Manber, Udi. *An Introduction to Algorithms: A Creative Approach*. New York: Addison-Wesley.

Farrell, Joyce. *Computer Programming Logic Using Flowcharts*. New York: Boyd and Fraser.

UNIT 2 PROGRAMMING FUNDAMENTALS: ALGORITHMS AND FLOWCHARTS

Structure

2.0 INTRODUCTION

As Randall E. Stross puts, it: 'The best programmers are not marginally better than merely good ones. They are an order-of-magnitude better, measured by whatever standard: conceptual creativity, speed, ingenuity of design, or problem-solving ability.' For problem solving, programmers use algorithms. In computing, and related subjects, an algorithm is an effective method for solving a problem using a finite sequence of instructions. Each algorithm is a list of well-defined instructions for completing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating at a final state.

2.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Elaborate on the importance and use of algorithms in computing
- Understand the steps that transform input into output and create an algorithm
- Discuss techniques of problem solving
- Understand the graphical representation of a problem with the help of a flowchart

- Differentiate between the different algorithms
- Know sorting, searching and merging using arrays

NOTES

2.2 PROCEDURE AND ALGORITHMS

According to Niklaus Wirth, a computer scientist, programs consist of algorithms and data, i.e.,

Programs = Algorithms + Data

An algorithm is an important component of the blueprint or plan for a computer program. In other words, an algorithm may be defined as 'an effective procedure to solve a problem in a finite number of steps'. It means that an answer has been found and it involves a finite number of steps. A well-framed algorithm always provides an answer. It is not necessary that the answer will be the one you want. However, there must be some answer. Maybe, you get the answer that there is no answer. A well-designed algorithm is also guaranteed to terminate.

The term 'algorithm' is derived from the name of Al-Khwarizmi, a Persian mathematician. We can define an algorithm as a finite set of well-defined instructions to accomplish a task. We implement algorithms through computer programs. If there is an error in the algorithm, it results in the failure of program implementation.

Thus, an algorithm is a step-by-step problem-solving procedure, especially an established, recursive computational procedure for solving a problem in a finite number of steps.

Characteristics of an Algorithm

As already mentioned, an algorithm is a finite set of instructions that accomplishes a particular task. An algorithm must satisfy the following criteria:

- **Input:** Zero or more items to be given as input.
- **Output:** At least one item is produced as output.
- **Definiteness:** The instructions, which are used in algorithm, should be clear and unambiguous.
- **Finiteness:** The algorithm should terminate after a finite number of steps.
- **Effectiveness:** Each and every instruction should be simple and very basic.

Analysis

- An algorithm must be developed using a top-down development approach.
- The importance of the data representation to the final algorithm that is being developed should be clearly defined.
- An algorithm must be robust so that it works properly on all cases, not just the regular or expected cases.
- An algorithm is a procedure to solve a problem. It is composed of several commands whose order of execution is known, and it is always guaranteed to stop in a finite amount of time.

- There is a measure called the order of the algorithm that allows us to compare the relative efficiency of one algorithm with another algorithm that solves the same or similar problems.

It is essential that we distinguish the problem-solving phase of a task from what we term as implementation phase. The problem-solving phase relates to the development of an ordered sequence of rules. These rules describe the successive operations required for solving a specific type of problem. The effective implementation of an algorithm in a programming language largely depends on how well an algorithm is prepared; a well-formatted algorithm finds itself translated into an efficient program.

Key Features of an Algorithm

The following steps are an example of an algorithm to make a pot of tea:

Step 1: If the kettle is water, then the kettle is filled.

Step 2: The kettle is plugged into the power point, and then it is switched on.

Step 3: If the teapot contains something, then it is emptied

Step 4: Tea leaves are placed in the teapot.

Step 5: In case there is no boiling of water in the kettle, repeat the step 2

Step 6: The kettle is switched off.

Step 7: Water is poured from the kettle into the teapot.

It may be observed here that the algorithm involves a number of steps. Steps 1, 3 and 5 involve decision-making, while one step, i.e., step 2 (the process of waiting for the kettle to boil) involves repetition.

An algorithm has the following three features:

- *Sequence*, which is also known as *process*.
- *Decision*, which is also known as *selection*.
- *Repetition*, which is also known as *iteration* or *looping*.

In 1964, two great mathematicians, Corrado Bohm and Guiseppe Jacopini, showed that an algorithm can be stated by using sequence, decision and repetition. Their work was very significant as it finally led to the development of the discipline of structured program design ,which is very much used today, and the work you are studying now is also a part of software engineering.

A Strategy for Designing Algorithms

Now that we have understood the basic concepts of algorithms and data, we can devise a strategy to design algorithms. It would be helpful if our strategy for designing algorithms involves the following steps:

Step 1: Investigation step: This step should include the following:

- Identifying the processes
- Identifying the major decisions

- Identifying the loops
- Identifying the variables

Step 2: Preliminary algorithm step: This step should include the following:

NOTES

- Devising a 'high level' algorithm
- Stepping through the algorithm

Does this 'walk-through' reveal any major problems? If it does, correct the problems.

Step 3: Refining the algorithm step: This step should include the following:

- Incorporating any refinement indicated in step 2
- Grouping together processes wherever appropriate
- Grouping together variables wherever appropriate
- Testing the algorithm again by stepping through it

At the first step, i.e. investigation step, you have to read through the statement of the problem to be solved. Also, you have to analyse the terms used in the statement. As we have seen in the tea-making example discussed earlier, there are various processes involved, such as filling the kettle, plugging the kettle into the power point, and so on. There are decisions, variables and loops.

The second step —preliminary algorithm—is the first attempt to solve the problem. This step may be quite crude. However, but the second part of step 2 tests it and brings out the errors.

The most difficult step is the third step, i.e., refining. This step requires a lot of practice. In fact, unless you have some programming experience, you cannot learn some of the skills suggested in this step. It would not hurt though to give the refining step some thought in this course.

Simple Examples of an Algorithm

1. **Consider you want to pass an exam. A simple approach put in terms of what is called as algorithm is depicted as follows:**
   - `Start`
   - `Have a look at the university syllabus`
   - `Attend the classes regularly`
   - `Prepare notes on your own`
   - `Study the subject properly`
   - `Solve the previous year question papers`
   - `Appear for the university exam`
   - `Wait for the result`
   - `The result is pass the exam`
   - `End`

2. An algorithm for addition of two numbers

Step 1: Start
Step 2: Read First number
Step 3: Read Second number
Step 4: Sum = First number + Second number
Step 5: Write (sum)
Step 6: Exit

3. To find the largest of three numbers

- Start
- Read 3 numbers: num1, num2 , num3
- If num1 > num2, then go to step 5
- If num2 > num3, then
  print num2 is the largest; else
  print num3 is the largest; go to step 6
- If num1 > num3, then
  print num1 is the largest; else
  print num3 is the largest
- End

Algorithm Representation through Pseudocode

A pseudocode is neither an algorithm nor a program. It is an art of expressing a program in simple English that parallels the forms of a computer language. It is basically useful for working out the logic of a program. Once the logic seems right, you can attend to the details of translating the pseudocode to the actual programming code. The advantage of pseudocode is that it lets you concentrate on the logic and organization of the program while sparing you the efforts of simultaneously worrying how to express the ideas in a computer language.

A simple example of pseudocode:

```
set highest to 100
set lowest to 1
ask user to choose a number
guess ( highest + lowest ) / 2
while guess is wrong, do the following:
   {
   if guess is high, set highest to old guess minus 1
   if guess is low, set lowest to old guess plus 1
   new guess is ( highest + lowest) / 2
   }
```

Coding

In the field of computer programming, the term **code** refers to instructions to a computer in a programming language. The terms **'code'** and **'to code'** have different

NOTES

meanings in computer programming. The noun '**code'** stands for source code or machine code. The verb **'to code' ,** on the other hand**,** means writing source code to a program. This usage seems to have originated at the time when the first symbolic languages evolved and were punched onto cards as 'codes'.

It is a common practice among engineers to use the word 'code' to mean a single program. They may say 'I wrote a code' or 'I have two codes'. This inspires wincing among the literate software engineer or computer scientists. They rather prefer to say 'I wrote some code' or 'I have two programs'. As in English it is possible to use virtually any word as a verb, a programmer/coder may also say 'coded a program'; however, since a code is applicable to various concepts, a coder or programmer may say 'hard-coded it right into the program' as opposed to the meta-programming model, which might allow multiple reuses of the same piece of code to achieve multiple goals. As compared to a hard-coded concept, a soft-coded concept has a longer lifespan. This is the reason of soft-coding of concept by the coder.

While writing your code, you need to remember the following key points:

- **Linearity:** If you are using a procedural language, you need to ensure that code is linear at the first executable statement and continues to a final return or end of block statement.
- **If constructs:** You would better use several simpler nested 'if' constructs rather than a complicated and compound 'if' constructs.
- **Layout:** Code layout should be formatted in such a way that it provides clues to the flow of the implementation. Layout is an important part of coding. Thus, before a project starts, there should be agreement on the various layout factors, such as indentation, location of brackets, length of lines, use of tabs or spaces, use of white space, line spacing, etc.
- **External constants:** You should define constant values outside the code. It ensures easy maintenance. Changing hard-coded constants takes too much time and is prone to human error.
- **Error handling:** Writing some form of error handling into your code is equally important.
- **Portability:** Portable code makes it possible for the source file to be compiled with any compiler. It also allows the source file to be executed on any machine and operating system. However, creating a portable code is a fairly complex task. The machine-dependent and machine-independent codes should be kept in separate files.

Program Development Steps

The following steps are required to develop a program:

- Statement of the problem
- Analysis

- Designing
- Implementation
- Testing
- Documentation
- Maintenance

Statement of the problem: A problem should be explained clearly with required input/output and objectives of the problem. It makes easy to understand the problem to be solved.

Analysis: Analysis is the first technical step in the program development process. To find a better solution for a problem, an analyst must understand the problem statement, objectives and required tools for it.

Designing: The design phase will begin after the software analysis process. It is a multi-step process. It mainly focuses on data, architecture, user interfaces and program components. The importance of the designing is to get the quality of the product.

Implementation: A new system will be implemented based on the designing part. It includes coding and building of new software using a programming language and software tools. Clear and detailed designing greatly helps in generating effective code with less implementing time.

Testing: Program testing begins after the implementation. The importance of the software testing is in finding the uncover errors, assuring software quality and reviewing the analysis, design and implementation phases.

Software testing will be performed in the following two technical ways:

- **Black box tests** or **behavioral tests** (testing in the **large**): These types of techniques focus on the information domain of the software.
  Example: Graph-based testing, Equivalence partitioning, boundary value analysis, comparison testing and orthogonal array testing.
- **White box tests** or **glass box tests** (testing in the **small**): These types of techniques focus on the program control structure.
  Example: Basis path testing and condition testing.

Documentation: Documentation is descriptive information that explains the usage as well as functionality of the software.

Documentation can be in several forms:

- Documentation for programmers
- Documentation for technical support
- Documentation for end users

NOTES

Maintenance: Software maintenance starts after the software installation. This activity includes amendments, measurements and tests in the existing software. In this activity, problems are fixed and the software updated to make the system faster and better.

Programming is the process of devising programs in order to achieve the desired goals using computers. A good program has the following qualities:

- A program should be **correct** and designed in accordance with the specifications so that anyone can understand the design of the program.
- A program should be **easy to understand.** It should be designed that anyone can understand its logic.
- A program should be **easy to maintain and update.**
- It should be **efficient** in terms of the speed and use of computer resources such as primary storage.
- It should be **reliable.**
- It should be **flexible**; that is to say, it should be able to operate with a wide range of inputs.

Tracing of Algorithms

The tracing of algorithms gives a beginner a good understanding of them. It is very important to know when algorithms can be applied, as it is crucial for the correct and efficient working of software programs. It provides basic information on the basics of algorithms and data structures as well as on the performance characteristics of specific algorithms used in development and programming tasks. It also offers some fundamental data structures and explains various sorting and searching algorithms, leading to efficient practices for storing and searching by way of linear, binary and array maps.

Swapping Algorithms

A swapping action is an exchange operation in which two objects change positions. Suppose there are two variables *x* and *y*, that may be integer data type or character type. For swapping x and y, a temporary variable 'temp' is needed to retain the value temporarily. When data is stored, it is often necessary to swap data elements. Though the concept of swapping algorithm is the same, two swapping techniques are used in C programming. They are as follows:

- Swapping two values (using temporary variable)
- Swapping two references (using pointer along with temporary variable)

In swapping two values, you need a variable 'temp' to hold a data item and then swapping it with another variable of the same data type. In this technique, the actual values of variables are swapped so that first variable has the value of second variable and vice versa. However, in swapping of references, the same result is

achieved by simply making use of the pointer to the value in memory. The actual values of the variables are not swapped.

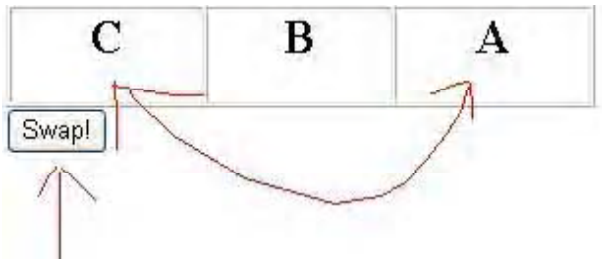


Figure 2.1 *Swap the Value*

The following algorithm is used to find out the swapping mechanism:

Step 1: integer a,b temp;

Step 2: a←2;

Step 3: b←3;

Step 4: print 'A =';

Step 5: print a;

Step 6: print 'B =';

Step 7: print b;

//Swapping the values of a and b

Step 8: temp←a; //temp variable keeps the value of a

Step 9: a←b; //value of b is shifted to a

Step 10: b←temp; //value of temp is shifted to b

//Values have been swapped

Step 11: print 'A ='; //Prints the value of A; earlier it was the value of B

Step 12: print a;

Step 13: print 'B ='; //Prints the value of B; earlier it was the value of A

Step 14: print b;

The preceding algorithm produces the following result:

```
A = 2
B = 3
A = 3
B = 2
```

The preceding algorithm is explained in the following way:

Input x (Let this variable be at location L_1)

Input y (Let this variable be at location L_2)

// Swap x and y

temp ← x

NOTES

```
x ← y
y ← temp
Output x      (Now x is at location L2)
Output y      (Now y is at location L1)
```

This swapping pseudocode can be written in assembly language code. In that case variables are moved to registers of the CPU with the MOV command.

Counting the Entered Values of a List of Numbers

The following algorithm is used to count the entered values of a list of numbers:

Algorithm to count the entered values of a list of numbers

```
Step 1: integer count, number;
//Declare variables as integer data types
Step 2: print 'Enter a value for 'for' loop;
Step 3: read n;
Step 4: count←0;
Step 5: for i←1 to n
{
print 'enter a value';
read number;
print 'Values you entered :'
print number;
count←count+i;
}
Step 6: print 'Number of entered items =';
Step 7: print count;
```

In this example, the integer variable `count` is first assigned a value of 0 and is initialized. Then the value of `count` is checked. Suppose it is less than 10, i.e. `<n` if `n` is taken as 10, then the loop body asks to enter a value for `number`. Then the value of `count` is incremented by `1` and again the count is checked for less than 10. If it is true the loop body is again executed and then the `count` is incremented by 1 and checked for < 10. It keeps on doing like this until the count is >= 10. Then the loop stops. So, here the loop is executed 10 times. The count variable prints the number of items as 10.

Finding the Sum of All the Entered Values of a List of Numbers

The following algorithm is used to count the entered values of a list of numbers:

```
Step 1: integer sum, number;
//Declare variables as integer data types
Step 2: print 'Enter a value for 'for' loop';
Step 3: read n;
Step 4: sum←0;
```

```
Step 5: for i←1 to n
{
print 'Enter a value';
read number;
print 'Values you entered :'
print number;
sum←sum+number;
}
Step 6: print 'Sum of entered items =';
Step 7: print sum;
```

In the preceding example the integer variable `sum` is first assigned a value of `0`. Then the values are entered up to `n` entered value. The `number` variable is used for input values to be added in the loop. Then the value of `sum` is incremented by adding the entered value for number. If it is true, the loop body is again executed and then the `sum` is incremented. Then the loop stops. The last value of all entered numbers will be summed up.

The result of the preceding algorithm is as follows:

```
Enter a value for 'for' loop 5
Enter a value
3
Values you entered: 3
Enter a value
12
Values you entered: 12
Enter a value
46
Values you entered: 46
Enter a value
91
Values you entered: 91
Enter a value
6
Values you entered: 6
Sum of entered items =158
```

Product of All the Entered Values of a List of Numbers

The following algorithm is used to find the product of entered values of a list of numbers:

```
Step 1: integer number;
//Declare variables as integer data types
```

NOTES

```
Step 2: long product,
//Declare variables as long data types
Step 3: print 'Enter a value for 'for' loop [up to 5]';
Step 4: read n;
Step 5: product←1;
Step 6: for i←1 to n
{
print 'Enter a value [From 1 to 20]';
read number;
print 'Values you entered :'
print number;
product←product*number;
}
Step 7: print 'Product of entered items =';
Step 8: print product;
```

In this algorithm, the long variable `product` is first assigned a value of `1` to be multiplied by enter values in the number variable. Then the values are entered up to `n` value. The product is multiplied by number which is being entered by user. After stopping the `for` loop, the product value is printed.

The result of the algorithm is as follows:

```
Enter a value for 'for' loop 4
Enter a value
3
Values you entered: 3
Enter a value
12
Values you entered: 12
Enter a value
19
Values you entered: 19
Enter a value
16
Values you entered: 16
Product of entered items =7344
```

Find Maximum and Minimum of a List of Numbers

The following algorithm is used to find the maximum and minimum of a given list of numbers:

Algorithm to find the largest value and second largest value of the given list of numbers

```
Step 1: integer number, counter, large, small;
//Declare variables As integer data types.
```

```
Step 2: print 'Enter a number';
Step 3: read number; //Accept the input values
Step 4: large←number;
//Assigning the very first number as maximum value
Step 5: small←number;
//Assigning the very first number as minimum value
Step 6: for counter←1 to 10
{
print 'Enter next number';
read number;
if large < number
large←number;
if small > number
small←number;
}
Step 7: print 'Maximum value in the list is';
Step 8: print large;
Step 9: print 'minimum value in the list is';
Step 10: print small;
```

In this algorithm, the 'for' loop is used to run the counter value till 10, which means 10 values are to be accepted for deciding the maximum and minimum value in the given list.

Implementation of finding the maximum and minimum value of the given list of numbers

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h>
#include <conio.h>
void main()
{
int number, counter, large,small;
clrscr(); //Clear the screen of previous
for(counter=1; counter<10; i++)
{
printf("Enter next number : ");
scanf("%d", number);
if(large<number)
large=number;
//The maximum value is assigned as the value of number
if(small>number)
small=number;
```

```
//The value itself is assigned as samll
}
printf("\nMaximum value in the list is = %d", large);
printf("\nMinimum value in the list is = %d", small);
getch();
}
The result of the above program is as follows:
Enter next number
12
Enter next number
56
Enter next number
3
Enter next number
90
Enter next number
468
Enter next number
25
Enter next number
54
Enter next number
25
Enter next number
67
Enter next number
38
Maximum value in the list is =468
Minimum value in the list is =3
```

2.3 ALGORITHM DEVELOPMENT

In general, an algorithm can be defined as a well-defined procedure for computation that takes an input of some values and produces an output. Hence, an algorithm is a sequence of steps which transform an 'input' into 'output'.

It can also be considered as a tool for solving a well-specified 'computational problem'. The problem statement defines the input/output relationship. The algorithm lists a specific computational procedure to reach that input/output relationship.

To put it simply, 'An algorithm is a step-by-step procedure for performing some task in a finite amount of time.'

Definition

NOTES

An algorithm is composed of a finite set of steps each of which may require one or more operations. Every operation may be characterized as either a simple or complex. Operations performed on scalar quantities are termed simple, while operations on vector data normally termed as complex.

Properties of Algorithm

The following are the five important properties (features) of algorithm:

- Finiteness
- Definitiveness
- Input
- Output
- Effectiveness
- **Finiteness:** An algorithm must always terminate after a finite number of steps. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **Definitiveness:** Each operation must have a definite meaning and it must be perfectly clear. All steps of an algorithm need to be precisely defined. The actions to be executed in each case should be rigorously and clearly specified.
- **Inputs:** An algorithm may have zero or more 'inputs' quantities. These inputs are given to the algorithm initially prior to its beginning or dynamically as it runs. An input is taken from a specified set of objects. Also, it is externally supplied to the algorithm.
- **Output:** An algorithm has one or more 'output' quantities. These quantities have a specified relation to the inputs. An algorithm produces at least one or more outputs.
- **Effectiveness:** Each operation should be effective, i.e. the operation must be able to carry out in a finite amount of time.

An algorithm is usually supposed to be 'effective' in the sense that all its operations needs to be sufficiently basic so that they can in principle be executed exactly in the same way in a finite length of time by someone using pencil and paper.

Criteria for Algorithm Design

Several active areas of research are included in the study of algorithms. The following four distinct areas can be identified:

1. Devising algorithms

The creation of an algorithm is an art. It may never be fully automated. A few design techniques are especially useful in fields other than computer science, such as operations research and electrical engineering. All of the approaches we consider have applications in a variety of areas including computer science. However, some important design techniques such as linear, non-linear and integer programming are not covered here as they are traditionally covered in other courses.

2. Validating algorithms

Once you have devised an algorithm, you need to show that it computes the correct answer for all possible legal inputs. This process is referred to as algorithm validation. It is not necessary to express the algorithm as a program. If it is stated in a precise way, it will do. The objective of the validation is to assure the user that the algorithm will work correctly and independently of the issues concerning the programming language in which it will eventually be written. After the validity of the method is shown, it is possible to write the program. On the completion of writing the program, a second phase begins. This phase is called program providing or sometimes as program verification. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program, which is annotated by a set of assertions about the input and output variables of the program. The second form is called specification and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.

3. Analysing algorithms

As an algorithm is executed, it uses computer's central processing unit (CPU) for performing operation. It also uses the memory for holding the program and its data. Analysis of algorithms is the process to determine the computing time and storage required by an algorithm.

4. Testing a program

Testing of a program comprises two phases: (i) debugging and (ii) profiling.

Debugging refers to the process of carrying out programs on sample data sets for determining if there are any faulty results. If any faulty result occurs, it is corrected by debugging. A proof of correctness is much more valuable than a thousand tests since it guarantees that the program will work correctly for a possible input.

Profiling refers to the process of the execution of a correct program on data sets and the measurement of the time and space it takes in computing the results. It is useful in the sense that it confirms a previously done analysis and points out logical places for performing useful optimization.

An example of profiling:

If we wish to measure the worst-case performance of the sequential search algorithm, we need to do the following:

- Decide the values of n for which the times are to be obtained
- Determine for each of the above value of n the data that exhibits the worst-case behaviour

Algorithm for sequential search:

```
1. Algorithm seqsearch (a, x, n)
2. //search for x in a[1: n] . a[0] is used as additional
   space
3. {
4. i := n; a[0] := x;
5. while(a[i] * x) do i := i " 1;
6. return i;
7. }
```

The decision on which the values of n to use is based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained. Assume that for algorithm, our interest is to simply predict how long it will take, in the worst case, to search for `x`, given the size `n` of `a`.

Algorithms as Technology

If computers were infinitely fast and computer memory was free, you would be in a position to adopt any correct method to solve a problem. In all likelihood, you would like your implementation to be adhering to good software engineering practice. However, you would use the method which is the easiest to implement.

However, computers may be fast, but they cannot be infinitely fast. Similarly, memory may be cheap, but it cannot be free. Thus, computing time and space in memory are bounded resources . You need to use these resources wisely. Such algorithms which are efficient in terms of time or space will be helpful.

Efficiency

It has been found that algorithm devices used for solving the same problem usually differ considerably in their efficiency. These differences are more significant than those due to hardware and software.

Algorithms and Other Technologies

Algorithms are indeed very important for contemporary computers considering other advanced technologies:

- Hardware with high clock rates, pipelining and super scalar architectures
- Easy to use, intuitive graphical user interfaces (GUIs)
- Local area networking (LAN) and wide area networking (WAN)

NOTES

A truly skilled programmer possesses a solid algorithmic knowledge and technique. It separates him/her from a novice. It is true that with modern computing technology, you can perform some tasks even if you do not have much knowledge of algorithms. However, if you are with a good background in algorithms, you can perform much more.

Types of Algorithms

- Approximate algorithm
- Probabilistic algorithm
- Infinite algorithm
- Heuristic algorithm

Approximate Algorithm

An algorithm is said to approximate if it is infinite and repeating., e.g. $\sqrt{2} = 1.414$, $\sqrt{3} = 1.732$, $\pi = 3.14$, etc.

Probabilistic Algorithm

If the solution of a problem is uncertain, then it is called probabilistic algorithm, e.g. tossing of a coin

Infinite Algorithm

An algorithm, which is not finite, is called infinite algorithm, e.g. a complete solution of a chessboard, division by zero.

Heuristic Algorithm

Giving less inputs and getting more outputs is called heuristic algorithm.

Design of an Algorithm

Example 2.1: Algorithm to pick the largest of three numbers.

```
Step 1: Read A, B, C.
Step 2: If A > B, go to Step 3.
        Else go to Step 5.
Step 3: If A > C
        Print A as the largest number.
        Else
        Print C as the largest number.
Step 4: Stop.
```

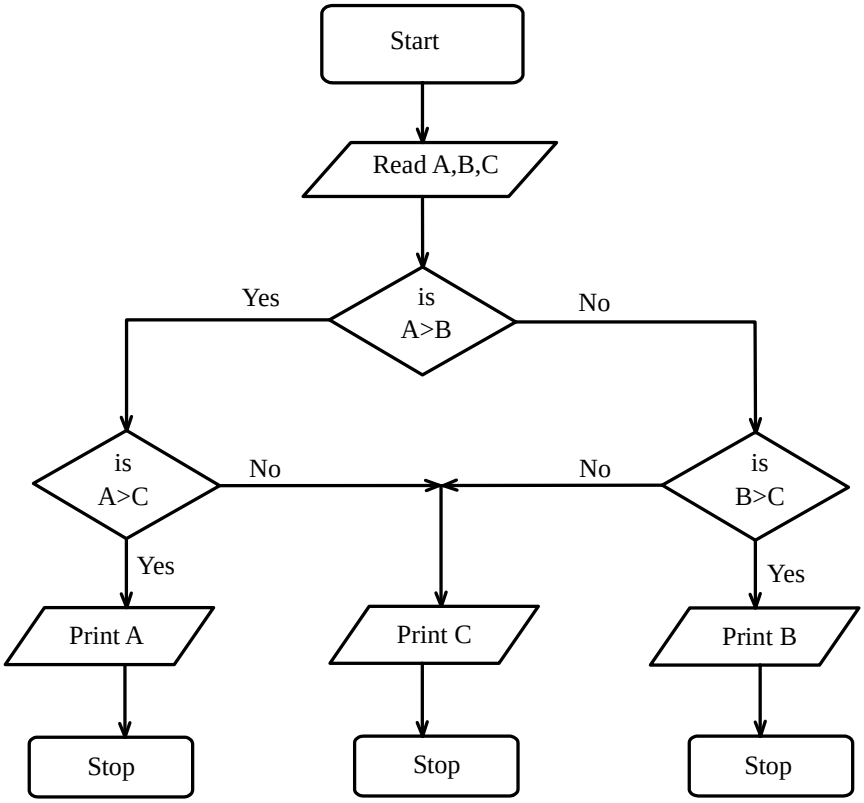
Step 5: If B > C

Print B as the largest number.

Else

Print C as the largest number.

Step 6: Stop.



Explanation: Read the three numbers A, B and C. A is compared with B. If A is larger, then it is compared with C. If A turns out to be the largest number again, then A is the largest number; otherwise, C is the largest number. If in the second step, A is less than or equal to be B, then B is compared with C. If B is larger, then B is the largest number; otherwise, C is the largest number.

Example 2.2: Algorithm to find the roots of a quadratic equation $ax^2 + bx + c = 0$ for all cases.

Step 1: Read a, b, c.

Step 2: disc ← b^2 – 4ac.

Step 3: If disc ← 0, go to Step 4.

Else, if disc > 0, go to Step 5.

Else, go to Step 6.

Step 4: root 1 ← – b/2a.

root 2 ← root 1.

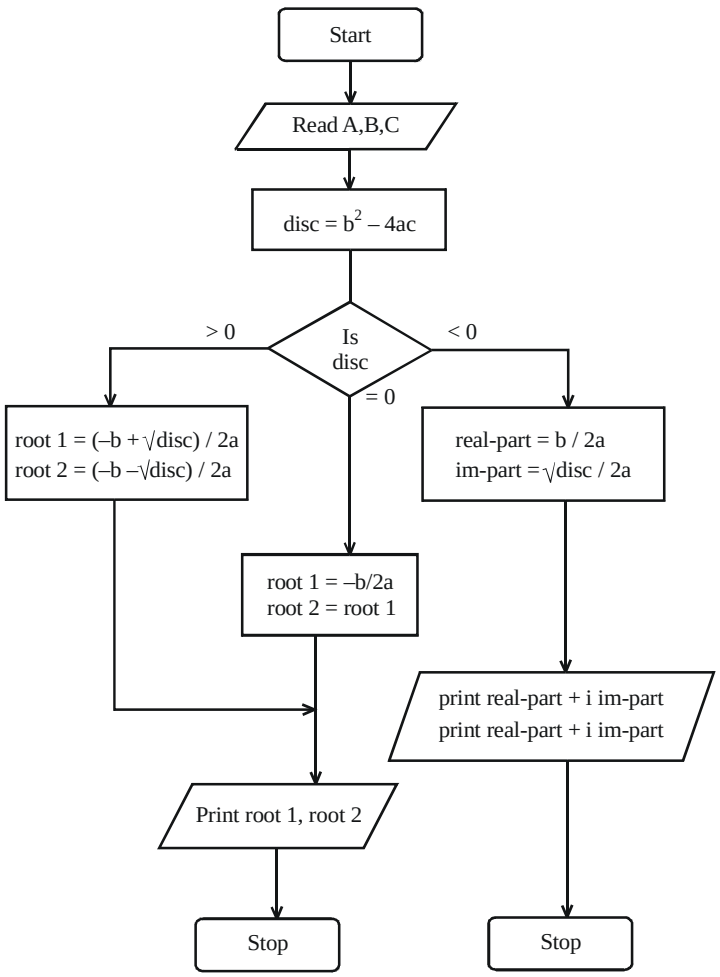
go to Step 7.

NOTES

Step 5: Root 1 ← (–b + sqrt (disc)) / 2a.
Root 2 ← (–b – sqrt (disc)) / 2a.
go to Step 7.

Step 6: real-part ← –b / 2a.
im-part ← sqrt(–disc) / 2a.
Print real-part + i im-part.
Print real-part – i im-part.
Stop.

Step 7: Print root 1, root 2.
Stop.



Example 2.3: Algorithm for finding maximum and minimum numbers.

Step 1: Read number.

Step 2: Maximum ← number. Minimum ← number.

Step 3: If (another number) go to Step 4.
Else go to Step 7.

NOTES

Step 4: Read number.

Step 5: If number > Maximum Maximum = number.

Else if number < Minimum

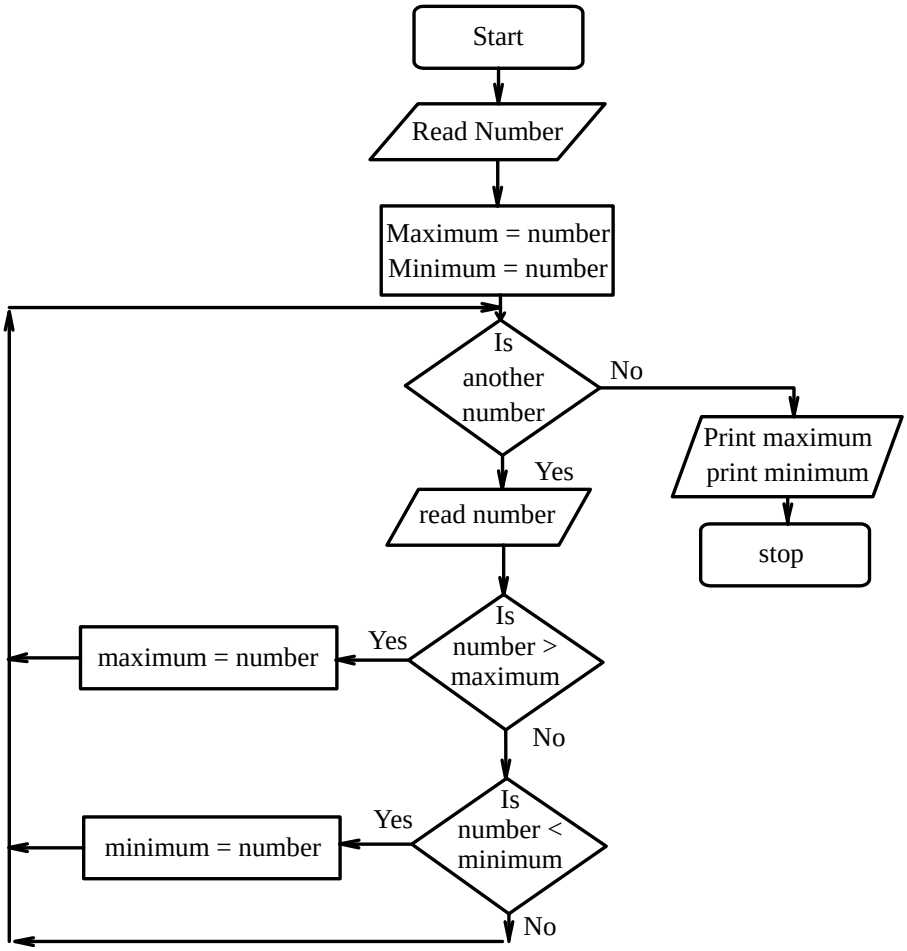
Minimum = number.

Step 6: go to Step 3.

Step 7: Print Maximum.

Print Minimum.

Step 8: Stop.



Example 2.4: Algorithm for finding maximum and minimum of given n numbers.

Step 1: Read N.

Step 2: Counter ← 1.

Read number.

Maximum ← number.

Minimum ← number.

NOTES

Step 3: If Counter < N go to Step 4.
Else go to Step 7.

Step 4: Read number.
Counter ← Counter + 1.

Step 5: If number > Maximum

Step 6: Maximum ← number.

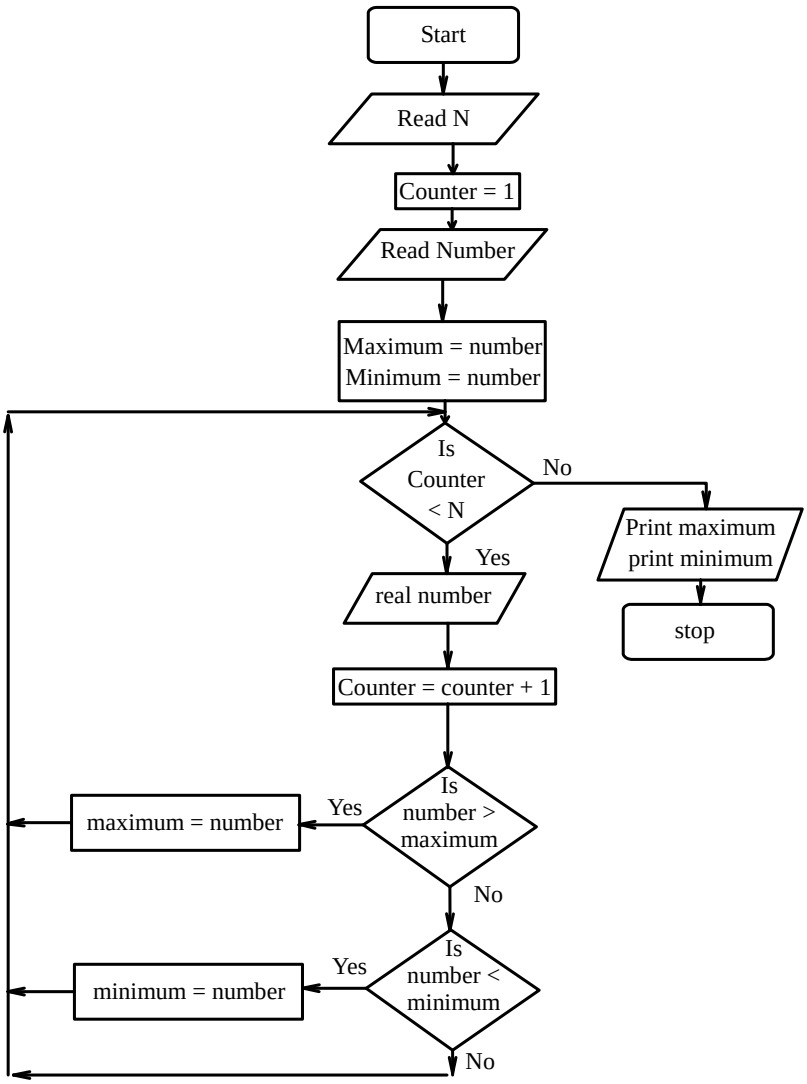
Step 7: Else If number < Minimum

Step 8: Minimum ← number.

Step 9: go to Step 3.

Step 10: Print Maximum. Print Minimum.

Step 11: Stop.



NOTES

Example 2.5: Algorithm for generating Fibonacci numbers up to N.

The first and second terms in the Fibonacci series are 0 and 1. The third and subsequent terms in the sequence are found by adding the preceding two terms in the series. The Fibonacci series is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

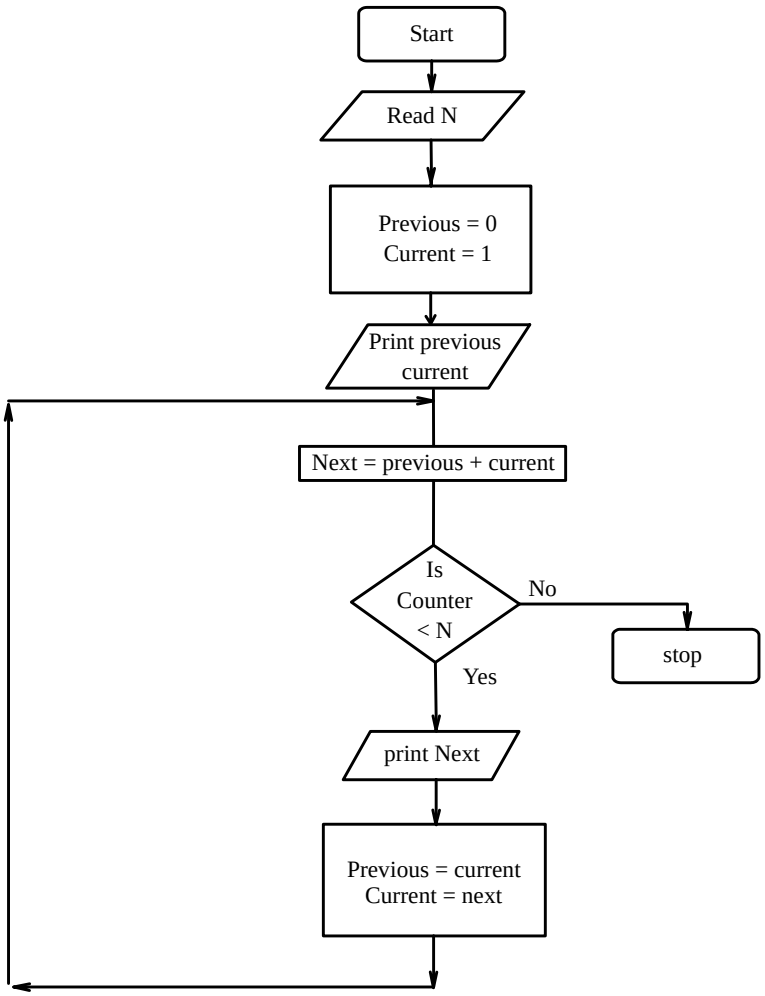
Step 1: Read N.

Step 2: Previous ← 0. Current ← 1.
Print Previous, Current.

Step 3: Next ← Previous + Current.

Step 4: If Next < N
Print Next.
Previous ← Current.
Current ← Next,
go to Step 3.
Else go to Step 5.

Step 5: Stop.



NOTES

Example 2.6: Algorithm for generating first k Fibonacci numbers.

Step 1: Read k.

Step 2: Counter ← 2.

Previous ← 0.

Current ← 1.

Print 'First k Fibonacci numbers are:'

Print Previous, Current.

Step 3: Next ← Previous + Current.

Counter ← Counter + 1.

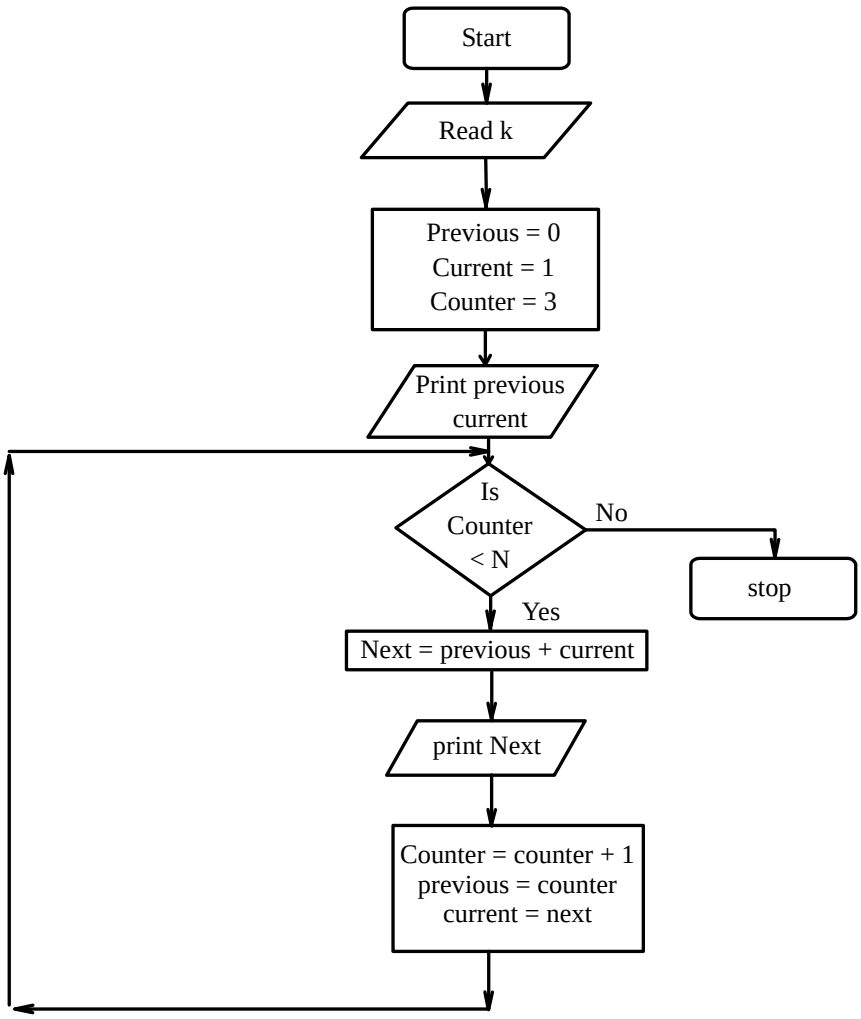
Print Next. Previous ← Current.

Current ← Next.

Step 4: If (Counter < k) go to Step 3.

Else go to Step 5.

Step 5: Stop.



Example 2.7: Sum of first n Factorials

The factorial of a non-negative integer n is the product of all positive integers less than or equal to n and is denoted by n! It is defined as follows:

```
n! = n(n-1) … 2*1
```

For example, 5!=5*4*3*2*1. Its older notation was $\underline{|n}$. In factorial number system where the denominations are 1, 2, 6, 24, 120, …, etc. the nth digit is in the range 0 to n. This identity works as shown in this example:

```
1*1!+2*2!+3*3!+ … +k*k! = (k+1)! – 1
Sum of 2!+3! = (2*1) + (3*2*1) = (2) + (6) = 8
```

The following algorithm is used to find the sum of n factorials:

Algorithm of sum of n factorials:

```
Step 1: integer n, factorial, i, j, sum;
Step 2: sum←0;
Step 3: print 'Enter the number';
Step 4: read n;
Step 5: for i←1 to n //Running outer loop till n value
{
factorial←1
for j←1 to i
//Inner loop to calculate the sum of n factorial values
{
factorial ←factorial*j;
//Calculating n factorial values
}
sum←sum+factorial;
//Calculating sum of n factorial values
}
Step 6: print 'Sum of n Factorials';
Step 7: print sum;
//Print the sum value of n factorials
```

Implementation to Find the Sum of First n Factorials

```
/*———— START OF PROGRAM ————*/
#include <stdio.h> //Declaration of Header files
#include <conio.h>
/*——-1/1! + 2/2! + 3/3! + 4/4! ...-——*/
void main()
{
int factorial,sum=0,i,j,n;
//Declaring and assigning the variables
```

```
printf("Enter a value for [n] value = ");
scanf("%d", &n);
//Accept input value for n term
for(i=1;i<=n;i++) //For outer loop till n value
{
factorial=1;
for(j=1;j<=i;j++)
//Using inner for loop to calculate the sum of n factorial
{
factorial = factorial *j;
}
sum=sum+ factorial;
}
Printf("\n sum of %d factorial = %d", n, sum);
}
getch();
}
```

The result of the preceding program is as follows:

```
Enter a value for [n] value = 4
sum of 4 factorial = 33
```

How this program works it is explained stepwise in the following:

```
1!+2!+3!+4! = 33
The value of 1! = 1
The value of 2! = 2
The value of 3! = 6
The value of 4! = 24
1+2+6+24= 33
```

Example 2.8: *To find largest value and the second largest value of the list:* The largest and second largest values in the given list are determined by array implementation. Array can contain the various elements of the list. The algorithm to find the largest and second largest of given list is as follows:

Algorithm to find the largest value and second largest value of the given list

```
Step 1: integer M, a[M], i, largest, t, second_largest;
Step 2: print 'Enter a value for array';
Step 3: read M;
Step 4: for i←1 to M
print 'Enter values:';
read a[i];
Step 5: if i==1
largest←t←second_largest←a[i];
```

```
Step 6: else if a[i]>largest
second_largest←largest;
largest←a[i];
Step 7: else if a[i]>second_largest && a[i]<largest
second_largest←a[i];
Step 8: else if a[i]<t
t←a[i];
Step 9: print 'Largest value in the given list =';
Step 10: print largest;
Step 11: 'Second largest value in the given list =';
Step 12: print second_largest;
```

The result of this algorithm is as follows:

```
Enter a value for array
5
```

Then `array A[M]` is assigned a value 5 as `A[5]`.

The input values are entered in the following way:

```
Enter values
45
90
112
4
35
Largest value in the given list = 112
Second largest value in the given list = 90
```

The first element of the array is `45` which is assumed to be the largest value and it is kept in the temporary location where it is temporarily stored in variable `t`. All the remaining values are checked from this number. Now, the `A[i]` value is assigned as `45`. At second step, the condition is satisfied so largest value is `90`. Now, `90` is checked with the next entered value `112`. As the condition is not satisfied, `112` is assumed as greater value. The values `4` and `35` are less than `90`, so the condition for less than largest is not satisfied. The checking process of second largest value `'45'` is done after checking the rest four values and declaring `112` as first largest value. Further, the statement `'second_largest=largest'` is used. The first element of the array is again taken as largest among the four values. Now, `45` is checked step-by-step in `if else if` conditional statement to find the second largest value.

Program to find the largest value and second largest value in a given list

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h>
#include <conio.h>
```

NOTES

```
#define M 5 //Define preprocessor directive that assigns
M = 5
void main()
{
int a[M], i, largest, t, second_largest;
clrscr(); //Clear the screen of previous
for(i=1; i<=M); i+)
{
printf("Enter %d value");
scanf("%d", &a[i]);
if(i==1)
largest=t=second_largest=a[i];
//The largest, t, second_largest values are assigned as
the value of a[i].
if(a[i]>largest)
{
second_largest=largest;
largest=a[i];
}
if(a[i]>second_largest && a[i]<largest)
second_largest=a[i];
if(a[i])<t)
t=a[i];
}
printf("\nLargest Vvalue in the given list = %d", largest);
printf("\nSecond largest value in the given list = %d",
second_largest);
getch();
}
```

The result of the preceding program is as follows:

```
Enter 1 value = 45
Enter 2 value = 90
Enter 3 value = 112
Enter 4 value = 4
Enter 5 value = 35
Largest value in the given list =112
Second largest value in the given list=90
```

In this program, `#define M 5` statement defines preprocessor directive that works as a macro. It means wherever `M` comes in the program, its value `5` is changed automatically. The `#define` statement can not be terminated by a

semicolon (;) because the preprocessor is a program that comes before `main()` statement.

Example 2.9: Determining *n*th root of a number.

The *n*th root of a number *a* is a number where *n* is positive integer. The nth roots are taken with the following iteration where *a* is the input values and *n* is the root value to be taken. The equation is arranged in the following ways:

$$b = \sqrt[n]{a}$$

where *a* is the number, *n* is the *n*th root and *b* is the value that retains the nth root of number a. For example, nth root is equal to 3 and number *a* is equal to 2 can be written as $2^3 = 8$. The following algorithm is used to find out the *n*th root of a given value:

Algorithm to find the *n*th root of a number:

```
Step 1:   double calculate_root(double,double);
//Declare a calculate_root function having two parameters
Step 2:   double Find_nth_Root(double,double,double);
//Declare a calculate_root function having three parameters
Step 3: double number(double,double);
//Declare a number function having two parameters
Step 4: double x←1, NUMBER_OF_ITERATIONS←40, n;
//Assign value 1 to x variable and NUMBER_OF_ITERATIONS=40
Step 5: double N;
//Declare a variable N as double data type
Step 6: double root;
//Declare a variable root as double data type
Step 7: x_label:
//Assign a label named as x_label
Step 8: print 'Enter root do want [2,3, …5] ?'
Step 9: read n;
//Accept input value n
Step 10: if n<=0
//Check the condition where n is less than 0
Step 11: print 'Number should be Greater than 0';
Step 12: print n;
Step 13: goto x_label;
//Go to label on x_label
Step 14: y_label:
//Assign a label named as y_label
Step 15: print 'Enter the number for Root';
Step 16: read N;
Step 17: if N<=0
```

NOTES

```
Step 18: print 'Number should be greater than 0';
Step 19: print 'PRESS ANY KEY TO ENTER AGAIN';
Step 20: goto y_label;
//Go to label on y_label
Step 21: x←calulate_root(n,N);
//x retains the returned value of function calculate_root
Step 22: print 'The first assumed root is',x;
Step 23: root←Find_nth_Root(N,n,x);
//root retains the Find_nth_Root returned value
Step 24: print 'Root of n',n;
Step 25: print N;
Step 26: print root;
Step 27: double calculate_root(double n,double N)
Step 28: integer i,xr;
//integer i and xr are declared
Step 29: xr←1;
//xr is assigned as 1
Step 30: double j←1;
//double j is assigned as 1
Step 31: while(1)
Step 32: for i←0 to n //Running for loop
    {
    xr←xr*j; //xr retains the value of xr*j
    }
Step 33: if xr>N
    Return j-1; //Returns j-1
Step 34: j←j+1; //j value is increased by 1
Step 35: xr←1; //xr value is increased by 1
Step 36: double Find_nth_Root(double NUM,double n,double
X0) //Function Find_nth_Root starts from here.
Step 37: int i;
Step 38: double d←1.0;
Step 39: double first_term, second_term, root←X0;
Step 40: for i←1 to NUMBER_OF_ITERATIONS
//Body of for loop starts that calculates first term and
second term value of enter values of NUMBER_OF_ITERATIONS
Step 41: d←number(root,n);
//d retains the n th value of given number.
Step 42: first_term←((n-1)/n)*root;
// first_term retains the value of let say 5 (5-1)/5)*
root value
```

```
Step 43: second_term←(1/n)*(NUM/d);
Step 44: root←first_term+second_term;
Step 45: print first_term,second_term,root;
Step 46: return root;
Step 47: double number (double x,double n)
Step 48: double d←1;
Step 49: integer i;
Step 50: for i←1 to n-1
Step 51:  d←d*x; //Printing the final nth root value of
given number n
Step 52: return d;
//Returns the resulted value to d
```

The preceding algorithm can work in the following way:

The odd nth root let say cube root of a real number b can not be identified with the fractional power a^{1/n}, although so has been done in the entries nth root and cube root. The fractional power with a negative base is not uniquely determined therefore, it depends not only on the value of the exponent but also on the form of the exponent, e.g.

(–1)^{1/3} = the 3rd root of –1, i.e. = –1

(–1)^(2/6) = the 6th root of (–1)^2, i.e. = 1

Implementation to Find the nth Root of a Number

```
/*————————— START OF PROGRAM —————————*/

#include<stdio.h>
#include<conio.h>
#define NUMBER_OF_ITERATIONS 40
//Preprocessor directive where NUMBER_OF_ITERATIONS is
defined as macro
double calculate_root(double,double);
double Find_nth_Root(double,double,double);
double number(double,double);
void main()
{
     double x=1,n;
     double N;
     double root;
x_label:
printf("\n Enter root value [2,3, …5] ?");
     scanf("%f",&n);
     if(n<=0)
```

NOTES

```
     {
     printf("\nNumber should be Greater than 0");
printf("Press any key to enter again");
     getch();
     goto x_label;
     }
y_label:
printf("\n\rEnter a number = ");
scanf("%f",&N);
     if(N<=0)
     {
     printf("\nNumber should be greater than 0");
printf("\n PRESS ANY KEY TO ENTER AGAIN …");
     getch();
goto y_label;
     }
x = calulate_root(n,N);
printf("\n\nThe first assumed root is calculated as
%f\n",x);
     root=Find_nth_Root(N,n,x);
printf("\n\n%f Root of %f = ",n,N);
     printf("Root value is = %f",root);
     getch();
    }

double calculate_root(double n,double N)
    {
     int i, xr=1;
     double j=1;
     while(1)
     {
     for(i=0;i<n;i=i+1)
     {
     xr=xr*j;
     }
     if(xr>N)
     {
     return(j-1);
     break;
     }
     j=j+1;
```

NOTES

```
     xr=1;
     }
     }
double Find_nth_Root(double NUM,double n,double X0)
{
     int i;
     double d=1.0;
     double first_term,second_term,root=X0;
for(i=1;i<=NUMBER_OF_ITERATIONS;i++)
{
d=number(root,n);
first_term=((n-1)/n)*root;
second_term=(1/n)*(NUM/d);
root=first_term+second_term;
printf("\n%f\t%f\t%f",first_term,second_term,root);
}
     return(root);
}
double number(double x,double n)
{
    double d=1;
    int i;
    for(i=1;i<=n-1;i++)
    d=d*x;
    return(d);
}
```

The result of the preceding program is as follows:

```
Enter root value [2,3, …5] ? 3
Enter a number = 64
Root value is = 4
```

In this program, the syntax of `#define` is as follows:

```
#define macro-name replacement-string
```

The `#define` command is used to make substitutions throughout the program file in which it is located. It causes the compiler to go through the file, replacing every occurrence of macro-name with replacement-string. The replacement-string stops at the end of the line. The above program calculates the nth root of any number a. This program uses the NEWTON_RAPTION_ ITERATION method for calculation. For example, you have to calculate the square root of 16, then n=2 (square root), a=16 (the number). The following examples show how nth root of the given number can be written:

Enter a Number = 32, Enter a Root = 5. The (n^{th}) 5^{th} root of 32 is 2.

Enter a Number = 11, Enter a Root = 4. The 4^{th} root of 11 is 1.82116.

Example 2.10: Greatest Common Divisor (GCD).

NOTES

The GCD of two integers is the largest integer value that divides both integer values where both the values are not zero. The basic identities of GCD are as follows:

```
GCD(A,B)=GCD(B,A)
GCD(A,B)=GCD(-A,B)
GCD(A,0)=ABS(A)
```

Both the integer values can be assumed as nonnegative integers. The GCD procedure extracts the greatest common divisor `A` because the common divisor `B` divides to get the remainder until finally `B` divides `A`. The result `A` is in fact a greatest common divisor because it contains every other common divisor `B`.

GCD Algorithm

```
Step 1: integer m, n, q, r; //Variables are defined
Step 2: print'Enter two values:';
Step 3: read m,n;
//Input two values for m and n variables
Step 4: if m==0 OR n==0
//Checking the condition whether m is equal to 0 or n is
equal to 0
print'One number is Zero';
else
reach: //Label reach is defined for loop
q←m/n;
//Get the value of q after dividing m by n
r←m – q*n; //Gets remainder value
Step 5: if r==0
print 'GCD Value is :'; //Prints message
print n; //Prints GCD value
goto end; //Got to end label
else
m←n←r;
//Assigning m is equal to n that is also equal to r
goto reach; //Go to reach label
end:; //Label end is defined
```

If the two given values are 10, 12 then the greatest common factor is the number that divides both the values 10 and 12.

NOTES

The GCD of two given integers (a and b) is the largest positive integer which divides both integers a and b, for example, gcd (10,12)=2. The following table shows the step-by-step procedure to get resultant GCD value:

Let the two values be, m =15 and n = 18.

div	quo	%Quo	%div	Resultant value
0				1
1	15	False	True	1
2	7	False	False	1
3	5	False	True	3
4	3			

The loop exits and returns 3. So, the resultant GCD value of the two given values 15 and 18 is 3.

Program to find GCD of given values:

```
/*————————— START OF PROGRAM —————————*/
#include <stdio.h>
#include <conio.h>
void main()
{
int m, n, q, r;
clrscr();
printf("Enter two values:");
scanf("%d%d", &m,&n);
if (m==0||n==0)
printf("One number is Zero");
else
reach:
{
q=m/n;
r=m - q*n;
}
if(r==0)
{
printf("GCD Value is : %d", n);
goto end; //Go to end label
}
else
{
m=n=r;
goto reach;
}
end:;
}
```

The GCD can also be calculated applying Euclidean algorithm. If the integers a and b are two positive integers and n is the remainder, then (a, b) = (b, r).

```
Euclidean_gcd(a,b)
Step 1: integer x, y, f, d;
Step 2: x←f; y←d;
Step 3: if y=0 return x
Step 4: r←x mod y;
Step 5: x←y;
Step 6: y←r;
Step 7: goto Step 2.
```

The above algorithm works in the following way:

Small value (x) = 10, Large value (y) =12.

Large	Small	Remainder
12	10	2
10	2	0

Result: 2 is the GCD of 10, 12.

The above algorithm is known as Euclid's GCD algorithm that extracts the greatest common divisor x. The common divisor y divides x and keeps remainder as value n. This process is continued until y divides x finally. Therefore, value assigned for x is the greatest common divisor if it contains every other common divisor y.

Example 2.11: Base Conversion (Decimal to Binary).

The base of a binary number is 2 and of decimal number is 10 (denary). Binary numbers have only two numerals (0 and 1), whereas decimal numbers have 10 numerals (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). An example of a binary number is 10011100 and decimal number is 0.012345679012. The decimal numeral system is the one that is the most widely used. Computer operations are performed with number base conversion.

The following algorithm is an example of printing an integer value into binary format:

Algorithm

```
Step 1: integer number, binary_val,temp_val, counter,d_val;
Step 2: binary_val←0; //Assigning value 0 to binary_val
Step 3: temp_val←number; // Assigning temp_val is equal
to number
Step 4: counter←0; //Assigning value 0 to counter
Step 5: print 'Enter the number';
Step 6: read number; //Accept input values to number
Step 7: if temp_val>0
{
```

```
d_val ← mod(temp_val,2)
binary_val ← binary_val + d_val*10^counter;
//10^counter means power(10,counter)
d_val← d_val + a_val*p_val;
temp_val ← int(temp_val/2)
//Change the fraction values as integer data types.
counter ← counter + 1;
//Increase the counter value by one
}
Step 7: print 'Binary Value';
Step 8: print binary_val; // Prints resultant binary
value
```

How this algorithm works is explained in the following.

Let us take a decimal value 6.

```
d_val = 6 mod 2 that returns 0
binary value=0+3*10^0 returns 0
d_val = 3 mod 2 returns 1
counter = 0 + 1 = 1
```

The decimal number **6** is equal to binary number **110**. This conversion is explained in the following way:

number	number/2	number % 2
6	3	0
3	1	1
1	0	1

Implementation of Base Conversion (Decimal to Binary)

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h> //Declaring Header files
#include <conio.h>
#include <math.h>
void main() //Start main() function
{
int number, binary_val, temp_val, counter, d_val, p_val;
binary_val=0;
// Declaring integer data types variables
temp_val=number; //Assigning temp_val is equal to number
counter=0; //Initailizing 0 to counter
printf("\n Enter a number");
scanf("%d", &number); //Accept input value
if (temp_val>0)
{
```

```
d_val = temp_val%2;
//Returns remainder to d_val
p_val=power(10,counter);
binary_val= binary_val+ d_val*p_val; //The value of binary_val is added to d_val by d_val by 10 'raise to the power' counter value
temp_val = int(temp_val/2)
//if temp_val contains fraction value, int() function changes the integer type value
counter = counter +1;
//Counter variable is increased
}
printf("Binary Value = %d", binary_val); //Printing the binary value
getch();
}
```

Base Conversion (Binary to Decimal)

Algorithm

Step 1: integer number,d_val,temp_val,counter,a_val;

Step 2: d_val←0; //Assigning 0 to d_val

Step 3: temp_val←number; //Assigning temp_val is equal to number

Step 4: counter←0; //Assigning 0 to counter

Step 5: print 'Enter the number';

Step 6: read number; //Accept input value for number

Step 7: if temp_val>0 //Body of if control statement

{

a_val ← mod(temp_val,10);

p_val←power(2,counter);

d_val← d_val+ a_val*p_val;

temp_val ← int(temp_val/10);

counter ← counter +1;

}

Step 8: print 'Decimal value';

Step 9: print d_val;

How the preceding algorithm works is explained in the following:

Let us take binary number 1011.

$=1*2^3+0+2^2+1*2^1+1*2^0$

$=8+0+2+1 = 11$

NOTES

The binary number **1011** is equal to decimal number **11**.

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h> //Declaring Header files
#include <conio.h>
#include <math.h>
void main() //Start main() function
{
int number,d_val,temp_val,counter,a_val;
// Declaring integer data types variables
temp_val=number;
//Assigning temp_val is equal to number
d_val=0;
counter=0; //Initailizing 0 to counter
printf("\n Enter a number");
scanf("%d", &number); //Accept input value
if (temp_val>0)
{
a_val = temp_val%10;
//Returns remainder to d_val
p_val = pow(2,counter);
//Returns counter value raise to the power 2 to p_val
variable
d_val=d_val+ a_val*p_val;
//The value of d_val is added to multiplied value of
a_val and p_val
temp_val = int(temp_val/10)
//if temp_val contains fraction value, int() function
changes the integer type value
counter = counter +1;
//Counter variable is increased
}
printf("Decimal Value = %d", d_val); //Printing the binary
value
getch();
//Pressing key to return the program
}
```

This program is able to convert the binary number into decimal number. The result of the program is as follows:

```
Enter a number = 1011
Decimal Value = 11
```

When a theoretical algorithm design is combined with the real-world data, it is called **algorithm engineering**. When you take an algorithm and combine it with a hardware device that is connected to the real-world, you can verify and validate the algorithm results and behaviour more precisely and accurately. A simple data acquisition or stimulus device may be considered as the real-world device. Alternatively, you can implement an algorithm on some embedded platform, such as a **field-programmable gate array** (**FPGA)** or microprocessor which can be similar to the final system design.

The first specific use of the term, 'algorithm engineering' was at the inaugural Workshop on Algorithm Engineering (WAE) in 1997.

It has of late been used for describing the steps in a graphical system design: 'A modern approach to design, prototype and deploy the embedded systems which combine open graphical programming with the **commercial off-the-shelf** (**COTS**) hardware for dramatically simplifying development, bringing higher-quality designs with a migration to custom design.'

With the help of algorithm engineering, you can transform a pencil-and-paper algorithm into a robust, efficient, well-tested and easily usable implementation. It covers various topics, from modelling cache behaviour to the principles of good software engineering. However, experimentation is its main focus.

2.4 TECHNIQUES OF PROBLEM SOLVING

In computing, problem means trouble with the system or software that is difficult to deal with and is solved to make it operational. The problem can be solved using various approaches of problem solving. The traditional and typical approach includes clear explanation of the problem, analyzing causes, identifying alternatives, assessing every alternative, selecting and implementing any one and final evaluation for solution. Problem solving functions when any system requires reaching the end goal state from the current or given state. Problem can be encountered when you start your system or when you define any incorrect algorithm.

If Your System is Not Starting

Restart your computer. Various software file related problems will be automatically corrected because the system will reload the required files when the system is restarted.

Check all the cable connections of the computer system. If no display on monitor, keyboard not responding, mouse not clicking, etc., check that all the respective cables are plugged in.

Check the electric power supply. Be confident that the SMPS of computer system is functioning properly.

Check for any unusual sounds. For example, the hard drive creating sound when the computer is switched on.

Start the computer using an external booting or start-up disk.

If the problem persists then contact the hardware engineer to resolve it.

Problem Solving for Incorrect Algorithms

An algorithm is used in mathematics, computing, linguistics and other allied subjects to efficiently solve a problem with the help of set order of instructions. It is used for data processing, calculation, etc. Flowcharts are also frequently used to express algorithms graphically. The computer performs a job with the help of an algorithm which defines the steps to be performed because every algorithm contains well-defined directives to complete a task. Beginning from an initial position, the directives/ instructions explain the steps of computation that proceeds via well-defined sequences of successive states and finally terminate in end state. Thus, an algorithm must be correct because it is a process which performs some specific sequences of instructions. If an algorithm is incorrect then either it will not produce the desired result or it may not function. The following are the methods of problem solving:

Problem Solving

- Using the programming process for problem solving.
- Solving the problem by:
  - analysing the problem by systematically and carefully understanding it;
  - outlining the necessities of the problem;
  - designing algorithm steps for the problem;
  - implementing the algorithm;
  - modifying the program when there is change in problem domain.

Assumptions can be identified behind the problem which play important role in problem solving. Sometimes these are ignored because efforts are not made to identify the hidden and unknown assumptions which may be the real cause of problem. Assumptions are essential because they set limits to the problem by providing structure to work, reflect preferred values to be maintained all over the solution and make simpler the problem by building it more controllable.

Problem Solving Techniques

The basic terminology related to problem solving includes purpose, situation, problem, cause, solvable cause, issue and solution. All these are explained in the following:

Purpose: Purpose is considered as the first step in problem solving, because if the purpose is not clear one cannot resolve the problems.

NOTES

Situation: Situation is the circumstance which can be good or bad. The problematic situations are recognized because all the situations are not problematic.

Problem: Problem is the specific part of a situation. The true problems are identified by specifying the purpose.

Cause: Cause is the effect of a problem. The causes must be distinguished from problems so that there is correct judgment of situation and basic problem. Though problems are part of a situation, hence problems are more common than causes are. Basically, causes are the precise facts which cause problems. Problem solving can only be specific if specific facts of causes are distinguished from problems.

Solvable cause: In a problem some causes can be solvable while some may be unsolvable. While solving a problem, the solvable causes are selected, because trying to solve unsolvable causes will simply be wastage of time. To extract solvable causes is an important step in problem solving.

Issue: Issue is the opposite expression of a problem. It specifies that the cause of the problem must be solved.

Solution: Solution is the precise act to solve any problem. It is similar to an exact action required to solve an issue. Issues are not the exact solutions but it specifies actions to be solved.

Other Problem Solving Techniques

Rational thinking: Rational thinking method is also used for problem solving. It includes the following points:

- Setting the perfect situation
- Identifying the current situation
- Comparing the perfect situation and the current situation
- Identifying the problem situation
- Breaking down the problem into various causes
- Visualizing the alternate solutions to the causes
- Evaluating and selecting the logical alternate solutions
- Implementing the solutions

The rational thinking method is used to solve almost all types of problems.

Systems thinking: Systems thinking is the scientific approach for problem solving. The system causing problem is analysed on the basis of systems functioning. The following are the types of system that are included in system thinking:

- Purpose
- Input
- Output

- Function
- Inside cause (Solvable)
- Outside cause (Unsolvable)
- Result

To understand the purpose of the problem the input is prepared using functions to get the desired output. The output may or may not define the purpose and the end result of the function can be different from the purpose which is produced by outside cause and inside cause. The inside cause can be solved whereas the outside cause is not. Systems thinking is considered as the most comprehensible and practical method for problem solving.

Cause and Effect Thinking: Conventionally, this defines cause and effect relations. First the cause is checked for the specific problem. Checking a cause and effect relation and then using problem solving method is a traditional and successful approach.

Hypothesis thinking: After collecting all relevant information the problem can be efficiently solved. Sometimes all the relevant information is not available and in order to collect all one may need longer time frame. In such cases, hypothesis thinking is very effective because it does not depend on all collected information. The hypothesis is developed on the basis of information available. Once the hypothesis is developed, minimum information is collected to prove this hypothesis. If the analysis proves that the first hypothesis is correct, then further information is not collected, but if the first hypothesis is incorrect then the next hypothesis is developed on the basis of available information. This method is considered as an efficient problem-solving method, because no time is wasted in collecting pointless information.

2.5 FLOWCHARTING

In the beginning, the use of flowcharts was restricted to electronic data processing for representing the conditional logic of computer programs. The1980s witnessed the emergence of structured programming and structured design. As a result of this, in database programming, data flow and structure charts began to replace flowcharts. With the widespread adoption of such ALGOL-like computer languages as Pascal, textual models like pseudocode are being used frequently for representing algorithms. Unified modelling language (UML) started the synthesis and codification these modelling techniques in the 1990s.

A flowchart refers to a graphical representation of a process which depicts inputs, outputs and units of activity. It represents the whole process at a high or detailed (depending on your use) level of observation. It serves as an instruction manual or a tool to facilitate a detailed analysis and optimization of workflow as well as service delivery.

Flowcharts have been in use since long. Nobody can be specified as the 'father of the flowchart'. It is possible to customize a flowchart according to need or purpose. This is why flowcharts are considered a very unique quality improvement method for representing data.

Symbols

A typical flowchart has the following types of symbols:

- **Start and end symbols**: They are represented as ovals or rounded rectangles, normally having the word 'Start' or 'End'.
- **Arrows:** They show the 'flow of control' in computer science. An arrow coming from one symbol and ending at another symbol shows the transmission of control to the symbol the arrow is pointing to.
- **Processing steps**: They are represented as rectangles.

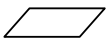
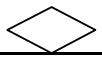
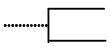
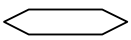
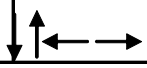
  Example: Add 1 to X.
- **Input/output symbol**: It is represented as a parallelogram.

  Examples: Get X from the user; display X.
- **Conditional symbol**: It is represented as a diamond (rhombus). It has a Yes/No question or True/False test. It contains two arrows coming out of it, normally from the bottom and right points. One of the arrows corresponds to Yes or True, while the other corresponds to No or False. These two arrows make it unique.

There are also other symbols in flowcharts may contain, e.g. connectors. Connectors are normally represented as circles. They represent converging paths in the flowchart. Circles contain more than one arrow. However, only one arrow goes out. Some flowcharts may just have an arrow point to another arrow instead. Such flowcharts are useful in representing an iterative process, what is known as a loop in terms of computer science. A loop, for example, comprises a connector where control first enters processing steps, a conditional with one arrow exiting the loop, and another going back to the connector. These are listed in Table 2.1.

NOTES

Table 2.1 *Symbols Used in Flowcharts*

Shape	Symbol	Symbol Name	Purpose
Oval		Terminator	To represent the begin/end or start/stop of a flow chart
Rectangle		Process	To represent calculations and data manipulations
Parallelogram		Data	To represent Input/Output data
Diamond		Decision	To represent a decision or comparison control flow
Double sided Rectangle		Predefined Process	To represent Modules or set of operations or a function
Bracket with broken line		Annotation	To represent descriptive comments or explanations
Document		Print out	To represent output data in the form a document
Multiple documents		Print outs	To represent output data in the form of multiple documents
Circle		Connector	To connect different parts of the flow chart
Hexagon		Repetition/ Looping	To represent a group of repetitive statements
Trapezoid		Manual Operation	To represent an operation which is done manually
Card		Card	To represent a card, e.g., punched card
Arrows		Flows of control	To represent the flow of the execution

It is now used at the beginning of the next line or page with the same number. Thus, a reader of the chart is able to follow the path.

Instructions

The following is the step-by-step process for developing a flowchart:

Step 1: Information on how the process flow is gathered. For this, the following tools are used:

- Conservation
- Experience
- Product development codes

NOTES

Step 2: The trial of process flow is undertaken.

Step 3: Other more familiar personnel are allowed to check for accuracy.

Step 4: If necessary, changes are made.

Step 5: The final actual flow is compared with the best possible flow.

Construction/Interpretation Tips for a Flowchart

- The boundaries of the process should be defined unambiguously.
- The simplest symbols should be used.
- It should be ensured that each feedback loop contains an escape.
- It should be ensured that there is only one output arrow out of a process box. Otherwise, it would require a decision diamond.

Types of Flowcharts

A flowchart is common type of chart representing an algorithm or a process and showing the steps as boxes of different kinds and their order by connecting these with arrows. We use flowcharts to analyse, design, document or manage a process or program in different fields.

There are many different types of flowcharts. On the one hand, there are different types for different users, such as analysts, designers, engineers, managers or programmers. On the other hand, those flowcharts can represent different types of objects. Sterneckert (2003) divides four more general types of flowcharts:

- Document flowcharts showing a document flow through system
- Data flowcharts showing data flows in a system
- System flowcharts showing controls at a physical or resource level
- Program flowchart showing the controls in a program within a system

However, there are several of these classifications. For example, Andrew Veronis named three basic types of flowcharts: the system flowchart, the general flowchart, and the detailed flowchart. Marilyn Bohl (1978) stated 'in practice, two kinds of flowcharts are used in solution planning: system flowcharts and program flowcharts...'. More recently, Mark A. Fryman (2001) stated that there are more differences. Decision flowcharts, logic flowcharts, systems flowcharts, product flowcharts and process flowcharts are just a few of the different types of flowcharts that are used in business and government.

Interpretation

- Analyse flowchart of the actual process.
- Analyse flowchart of the best process.
- Compare both charts looking for areas where they are different. Most of the time, the stages where differences occur are considered to be the problem area or process.

- Take appropriate in-house steps to correct the differences between the two separate flows.

Example: Process flowchart—finding the best way home.

This is a simple case of processes and decisions in finding the best route home at the end of the working day.

A flowchart provides the following:

- **Communication:** Flowcharts are excellent means of communication. They quickly and clearly impart ideas and descriptions of algorithms to other programmers, students, computer operators and users.
- **An overview:** Flowcharts provide a clear overview of the entire problem and its algorithm for solution. They show all major elements and their relationships.
- **Algorithm development and experimentation:** Flowcharts are a quick method of illustrating program flow. It is much easier and faster to try an idea with a flowchart than to write a program and test it on a computer.
- **Check program logic:** Flowcharts show all major parts of a program. All details of program logic must be classified and specified. This is a valuable check for maintaining accuracy in logic flow.
- **Facilitate coding:** A programmer can code the programming instructions in a computer language with more ease with a comprehensive flowchart as a guide. A flowchart specifies all the steps to be coded and helps to prevent errors.
- **Program documentation:** A flowchart provides a permanent recording of program logic. It documents the steps followed in an algorithm.

Advantages of Flowcharts

- Clarify the program logic.
- Before coding begins, a flowchart assists the programmer in determining the type of logic control to be used in a program.
- Serve as documentation.
- Serve as a guide for program coding of program writing.
- A flowchart is a pictorial representation that may be useful to the businessperson or user who wishes to examine some facts of the logic used in a program.
- Help to detect deficiencies in the problem statement.

Limitations of Flowcharts

- Program flowcharts are bulky for the programmer to write. As a result many programmers do not write the chart until after the program has been completed. This defeats one of its main purposes.

NOTES

- It is sometimes difficult for a business person or user to understand the logic depicted in a flowchart.
- Flowcharts are no longer completely standardized tools. The newer structured programming techniques have changed the traditional format of a flowchart.

Differences between Flowcharts and Algorithms

Flowchart

- It is the graphical representation of the solution to a problem.
- It is connected with the shape of each box indicating the type of operation being performed. The actual operation, which is to be performed, is written inside the symbol. The arrow coming out of symbol indicates which operation to perform next.

Algorithm

- It is a process for solving a problem.
- It is constructed without boxes in a succession of steps.

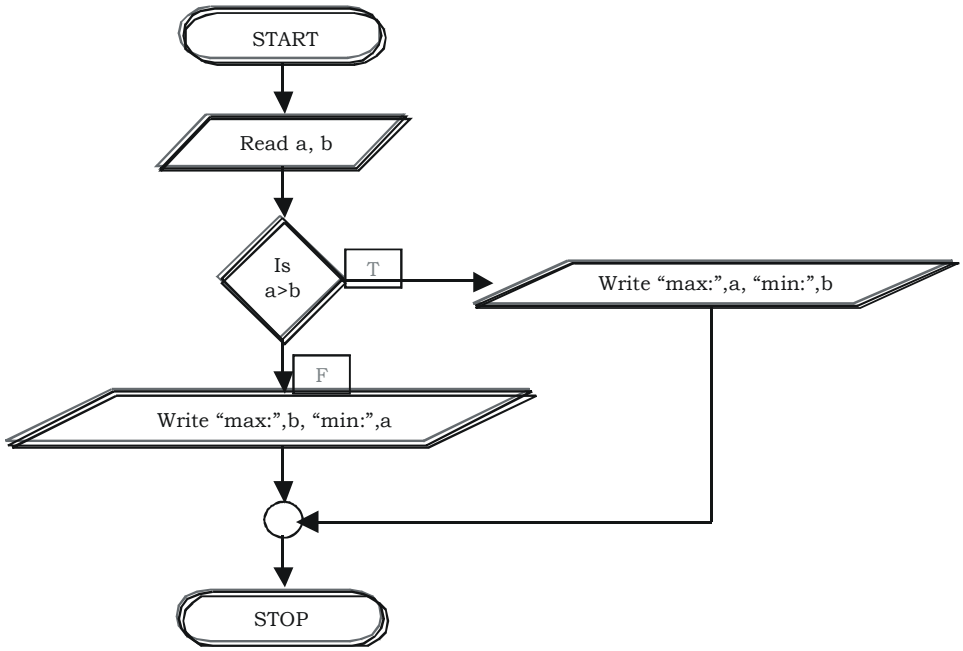
An algorithm can be written in the following three ways:

- **Straight sequential:** A series of steps that can be performed one after the other.
- **Selection or transfer of control:** Making a selection of a choice from two alternatives of a group of alternatives
- **Iteration or looping:** Performing repeated operations.

The following are the examples of algorithms and flowcharts for some different problems:

Examples of Straight Sequential Execution

Example 2.12: Write a flowchart to find the maximum and minimum of given numbers.



Example 2.13: Write the various steps involved in executing a 'C' program and illustrate it with the help of a flowchart.

Executing a program written in C involves a series of steps. They are as follows:

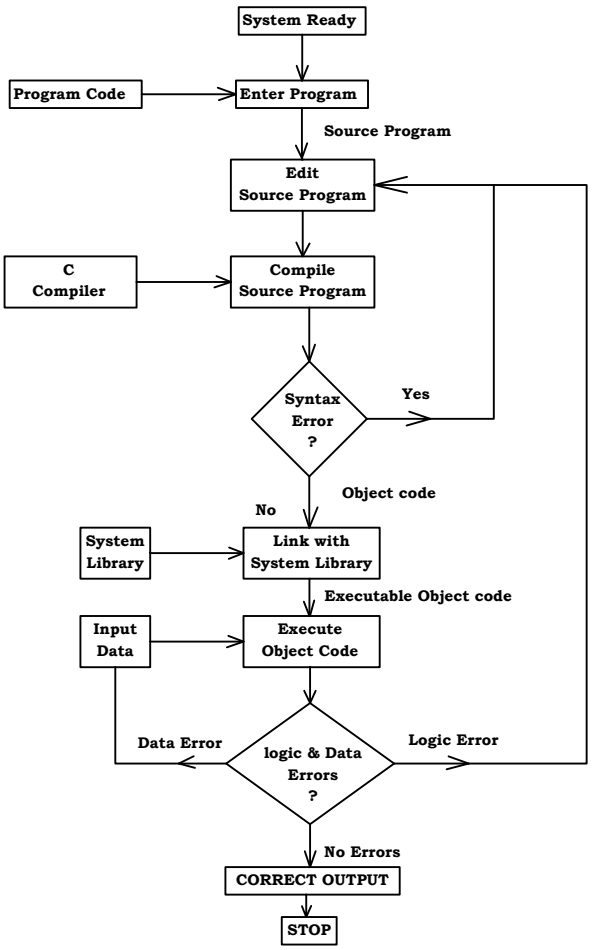
- Creating the program.
- Compiling the program.
- Linking the program with functions that are needed from the C library.
- Executing the program.

Although these steps remain the same irrespective of the operating system, system commands for implementing the steps and conventions for naming files may differ on different systems.

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

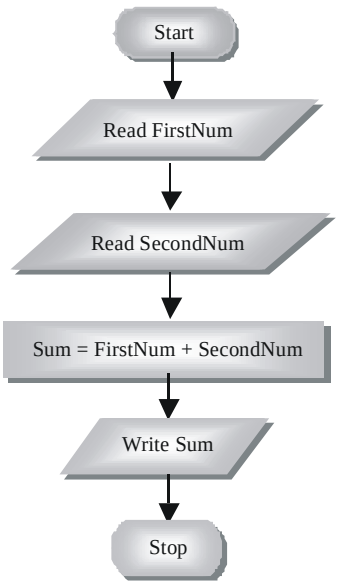
The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers).

NOTES



Examples for Flowcharts with Algorithms

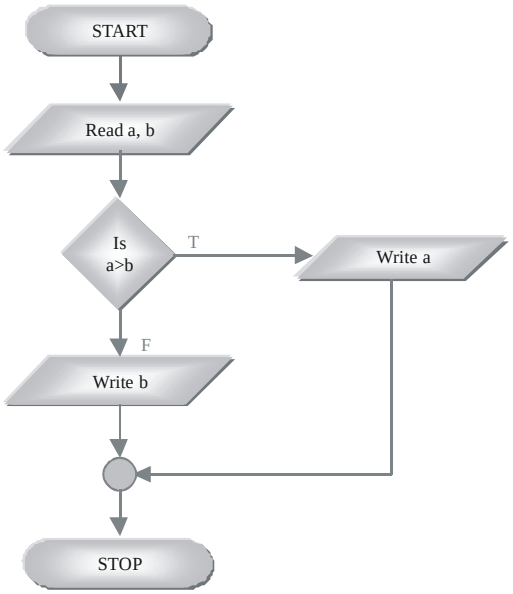
a. Draw a flowchart for adding two numbers and write an algorithm for it.



Algorithm for addition of two numbers:

```
Step 1: Start
Step 2: Read FirstNumber
Step 3: Read SecondNumber
Step 4: Sum= FirstNumber + SecondNumber
Step 5: Write (Sum)
Step 6: Exit
```

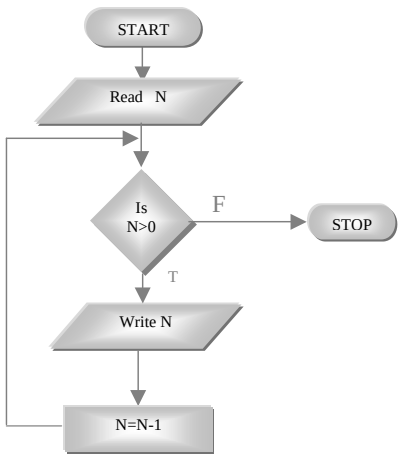
b. Draw a flowchart to find the larger number between two numbers and write an algorithm for it.



Algorithm for finding large number between two numbers

```
Step 1: Start
Step 2: Read a and b
Step 3: IF a > b THEN Write (a)
        ELSE Write(b)
Step 5: Stop
```

c. Draw a flowchart to display natural numbers between 1 and N in reverse order.



NOTES

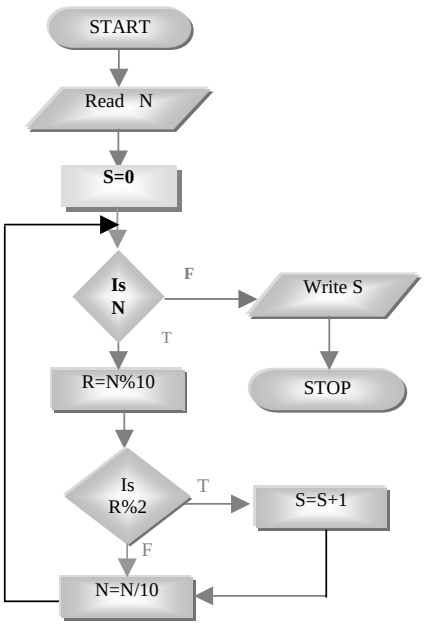
d. Draw a flowchart to display number of odd digits in a given number.

Algorithm for displaying Natural numbers between 1 and N in Reverse Order.

```
Step 1: Start
Step 2: Read  N
Step 3: Repeat while N>0
                Write (N)
                 N=N-1
Step 4:Exit
```

Algorithm to display number of odd digits exist in a given number.

```
Step 1: Start
Step 2: Read  N
Step 3: S=0
Step 4: REPEAT while N>0
               R=N mod 10
               IF  R mod 2 THEN
                      S=S+1
               N =N/10
Step 5. Write(s)
Step 6: Exit
```

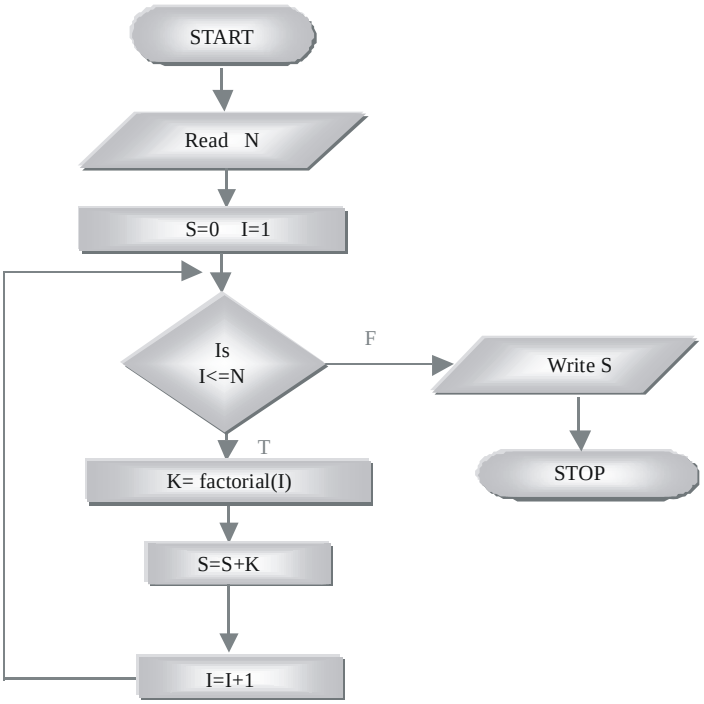


NOTES

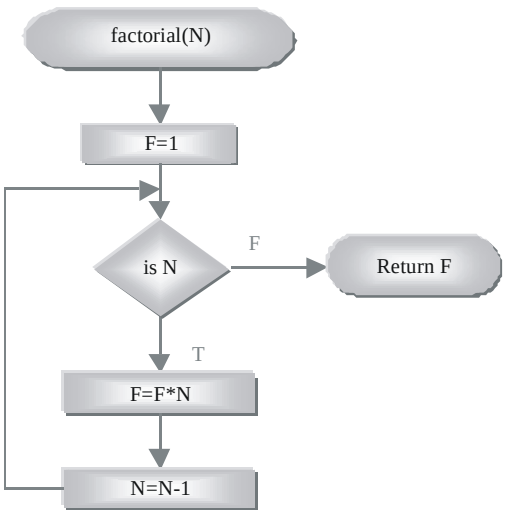
e. Draw a flowchart to evaluate the series 1! + 2!+ 3!+ ··· +N!

Algorithm for evaluating the series 1!+2!+…..+N!

```
Step 1: Start
Step 2: Read  N
Step 3: S=0,I=1
Step 4: Repeat while I<=N
                K=factorial(I)
                S=S+K
                I=I+1
Step 5. Write(S)
Step 6: Exit
```



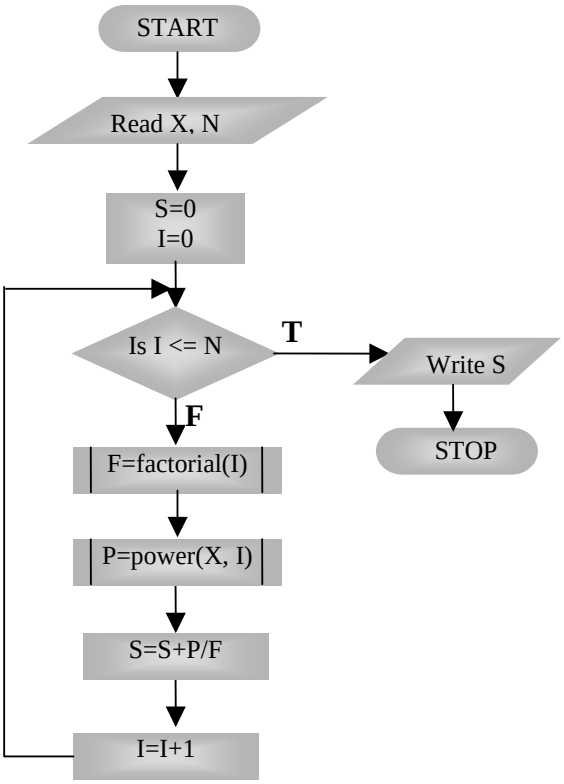
f. Flowchart to evaluate N!



Algorithm to find factorial(N). Where N is a value and function returns Factorial value for N

```
Step 1: F=1
Step 2: Repeat while N <> 0
                F=F*N
                N=N-1
Step 3: Return F
```

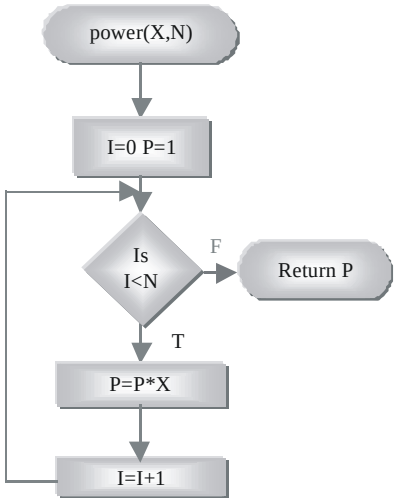
g. Draw a flow chart to evaluate the series $1+x+ x^2/2! +\cdots+x^n /N!$



Algorithm to evaluate $1 + X + X^2/2! +\ldots+ X^n/N!$

```
Step 1: Read X,N
Step 2: S=0,I=0
Step 3: Repeat while I<=N
                F=factorial(I)
                P=power(X,I)
                S=S+P/F
                I=I+1
Step 4: Writes(S)
Step 5: Exit
```

h. Flowchart to evaluate `Power(X, N)`



Algorithm to evaluate Power(X, N). Where X and N are values

```
Step 1: I=0,P=1
Step 2: Repeat while I<N
                P=P*X
                I=I+1
Step 3: Return P
```

CHECK YOUR PROGRESS

1. What is the basic terminology related to problem solving?
2. Define 'flowchart'.
3. List two advantages and disadvantages of a flowchart.
4. What is the difference between a flowchart and an algorithm?

2.6 STEPWISE REFINEMENT

The program is broken into smaller subproblems and the required algorithm is specified for each subproblem is known as stepwise refinement. It supports the breaking of complicated task into smaller task. The word refinement includes the process of high level procedure specifications. The stepwise refinement mechanism is used in programming concepts. A refinement process in writing a program divides the problem from the top most segment. It segments into series of small tasks. These tasks perform a sequence of smaller tasks. These tasks perform an order of performance. For example take an example of calculating sum of two given values. The steps taken in refinement concept are as follows:

NOTES

```
Initialize variables
Input the values
Assign third variable that keeps the sum value of variable
one and variable two
Print sum
```

Refinement basically uses sequencing. If pseudocode statement is started with initializing the variables, the refinement scheme defines the whole process as follows:

```
Initializing the variables
Keep one variable that keeps the sum of two given values
```

All the variables are to be initialized before using them. The sum value is decided as per user input value and is not initialized instead calculated as per provided values. The pseudocode '**Print sum'** can be refined as follows:

```
Input for two values
Set a variable that counts the summation of two values
Print "Sum", sum
```

The given problem can be broken down into components which can be executed directly into the components and associated with implementation language, such as PROLOG. For example, a procedure is written as follows:

```
Biggest(X, Y) :- (X=0; X=1), Y is X+1
```

The Biggest is considered as function that defines the values of `X` and `Y` and `Y` are dependent on the value of `X`. The required specifications in programming are transformed into code by the steps of correctness preserving steps. These steps are considered as refinement. For example, command `S` is refined by the command `T`. If `T` is terminated naturally for all inputs then `S` also terminates normally.

This technique is used to write modules or subprograms which are efficient, correct and easy to modify. In this, the main task is written first then BIG task is broken into smaller tasks that are referred to as methods or functions defined in highlevel languages, such as C, C++, Java, etc. Each method is taken one at a time and broken it up and this process is continued until each method focuses on cohesion. For example, in pseudocode of sorting, the top-level function is written to reduce the programming complexity and repeat this function along with stepwise refinement until it is called successfully. This provides reuse of coding. This technique basically involves operations and data structures representing structured programming. For example, if a given integer number `n` is determined whether it is prime number or not is determined for the outline of the solution. This problem can be broken out as per Fermat's theorem that follows the techniques of stepwise refinement. This can be assessed by breaking down the following steps:

- If `n` is defined as prime number and `a` is taken as positive integer which is less than n, then `a` raise to the `n`th power is congruent to `a` modulo `n`.

- The two given numbers are congruent modulo `n` and both have the same remainder if it is divided by `n`. It is also referred to as `a modulo n`.
- If `n` is not considered as prime then most numbers do not satisfy `a<n` condition. A random number is picked if value of `a<n` and remainder comes as `a`n modulo `n`. If value is not equal to `n` then the defined number is not considered as prime number.

The very first step is to be defined as a functional algorithm which includes iterative functions as follows:

$$prime(n,\ q) = \begin{cases} true & \text{if } n = 2 \\ prime\_test(n,\ q,\ false) & \text{otherwise} \end{cases}$$

The iterative function `prime(n,q)` is declared as `prime: P × N →` `{true, false}`, where `n` is the defined prime number and `q` is the maximum number of elements.

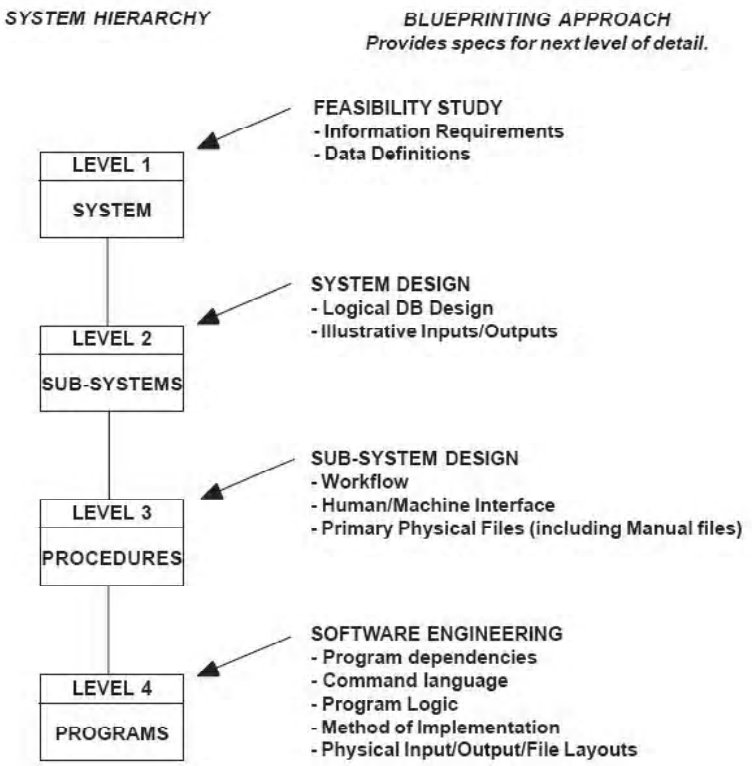


Figure 2.2 *Programming by Stepwise Refinement*

In Figure 2.2, the four levels are set as SYSTEM, SUB-SYSTEM, PROCEDURES and PROGRAMS respectively. The level 1 is SYSTEM that determines feasibility study including information requirements and data definitions of the required system, whereas level 2 is SUB-SYSTEM that determines logical design and input/output values. The level 3 is PROCEDURES containing workflow, machine interface and primary physical files and level 4 is SOFTWARE ENGINEERING that contains all aspects of program dependencies, command language, program logic, method of implementation, and layouts of input and output.

Stepwise refinement (SWR) is considered as fundamental aspect of programming sequence. This technique is started with module defining that writes top level, specifies step, proves structure and then writes step. The specifying step and writing step are encircled because of depending on user's input values or if the values are defined within the program write step depends on the values (Figure 2.3).

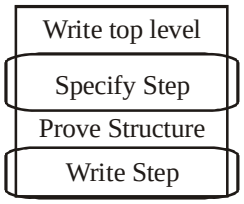


Figure 2.3 *Stepwise Refinement Program*

The resulting program structure deals with the stepwise refinement. It follows structured programming. A program is developed to calculate class averaging program. This problem is divided the top into smaller tasks and syntax is written in a specified order which is followed by the declared program. The refinement scheme supports the following steps:

First Refinement

This type of refinement keeps the following sequence of pseudocode:

Initializing the required variable→Input the values→ sum the total and count exam grades→calculating and then printing class average of given values

The second level of refinement needs variables and repetitive structure. A variable is used to receive each grade's value and a loop is needed to calculate the total grades before calculating the average value.

Second Refinement

This type of refinement keeps the following sequence of pseudocode:

Input a value for first grade→Add the grade to running total→ Input another value for next grade→Calculate and print the class average

This pseudocode can be refined as follows:

If counter is not considered as zero, set the average value to the total divided by counter value. It prints the average else a message as **"No grades are provided."**

Note: A fatal error occurs if a number is divided by zero. So grade is determined if the values for subjects are provided properly.

Third Refinement

The third refinement keeps the following sequence of pseudocode:

Initialize zero for total value→Initialize zero for counter variable→Input a value for first grade→Add this grade for running total→Add one to the grade counter→Input the very next grade→If counter value is not equal to zero→Average is set to the total value divided by counter→Print the average→Print the message "No grades are provided."

The top down stepwise refinement algorithm supports the following steps:

- Divide and conquer strategy
- Change the programming part from general to specific
- Divide the problems into levels that makes strategy
- Do not allow storage concept, for example, put the number into a specified variable X.
- Determine the process of division. Divide by task is possible what to do?

A general algorithm specifies in stepwise refinement so that it is suitable for `while` loop. This is to be continued until and unless number is not equal to 0. The required pseudocode is necessary to get the result for positive integer to get the output in number of terms N in the following way:

```
get a positive integer N from the user;
while N is not 1:
compute N = next term;
output N;
count the term;
output the number of terms;
```

The next term is computed and checked whether number N is odd or even. For this, two extra steps including if-else statement will be taken place that decides even or odd. The required pseudocode is as follows:

```
get a positive integer N as input value asked from user;
while N is not 1:
if N is even;
calculate N = N/2;
else
calculate N = 3 * N +1;
```

```
count the term;
output the number of terms;
```

In the preceding pseuducode, counting represents that the calculation must be started with 0.

NOTES

2.7 ALGORITHM FOR SORTING AND SEARCHING

If you are a movie buff and a fan of old detective movies, it will be very easy for you to recall how Mr Chan, when the occasion demanded, used to introduce his sons to his clients. He would say, 'This is my number 1 son, my number 2 son, etc.' It shows that although computers were not heard of and were to be invented long after that time, Mr Chan used the method of referring to his sons by their subscripts. You will see the essence of this method while using arrays in the following example.

Suppose your company has plans to form a new metal alloy and you have been entrusted with the responsibility of developing a computer simulation program that provides heat statistics of this new substance to show how heat passes through it. For the purpose, a thin rod of this new material is to be used, which is to be placed between two heat sources, with each source maintaining a constant temperature. It is not required that both heat sources maintain the same temperature. The initial temperature of the rod is kept uniform and the rod is fully insulated from its surroundings.

This problem can be solved using a method known as 'finite difference method'. To ascertain the temperature at different points along the rod, the rod is divided into *n* equal segments of length *h* each.

Now assume that for your program you divide the whole rod into 1000 equal segments. At the initial stage, the leftmost segment of the rod has the same temperature as the heat source at the left end, and the same is the case with the rightmost segment of the rod. The rest of the rod has the initial temperature. You are required to write a program that simulates the heat flow along the rod with the passage of time by calculating each segment's temperature.

Let us leave aside the actual heat dynamics of this given problem and concentrate on something that is the main focus of this section. Consider the 1000 segments the temperature of which is to be measured. For this purpose, you require 1000 variables; just imagine the complexity of writing such a program. In real-life situation the number can go much higher, say 10,000 segments.

The problem of using such a great number of variables can be easily resolved by the use of arrays. Whether there are 1 million variables or 10,000, with the use of arrays both situations can be handled with any additional complexity. Let us understand arrays in detail.

An array is an ordered collection of elements that share the same name. We use declarations to tell the compiler when we want an array. The compiler needs to know the same things about an array that it needs to know about an ordinary variable (known as a scalar variable in the trade): the type and the storage class. In addition, it needs to know how many elements the array has. Arrays can have the same types and storage classes as ordinary variables, and the same default rules apply.

An **array** is a collection of elements of the same data type and it can be referred to with a single name. The statement for declaring an array includes the name of the array and specifies the data type and number of array elements. A variable of the array type is a pointer to the type of array elements.

Syntax:

```
<type>  <variable_name>[index1]{[index2]....};
```

- An array can be of single dimensional or two-dimensional (multi-dimensional) type.
- The memory allocation for an array type depends on the memory required to store all its elements.

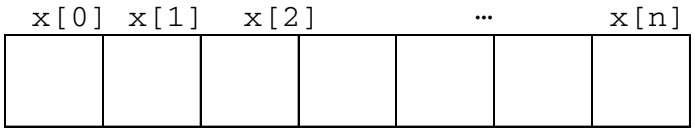
The following conditions must be satisfied by the elements of an array:

- All elements in an array are of same data type.
- The first element in the array is numbered; so the last element is 1 less than the size of the array.
- The first element is referred to as array `[o]` and any element at `x` position is referred to as array `[x]`.
- Each member in the array is referred to by its position in arrays using array name followed by one or more subscripts, with each subscript enclosed in square brackets.
- Array elements are always stored in contiguous memory locations.
- The value of each subscript must be expressed as a non-negative integer constant, variable or an expression.

Subscripts are used to refer to array elements. Each array element is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets.

The number of subscripts determine the dimensionality of the array; they are written as `x[i]` and refers to an element in the one-dimensional array x. In the `n`-element array `x`, the array elements are `x[o], x[1], x[2], ....., x[n – 1], x [n].`

It can be represented as follows:



x is an array with n elements.

The array elements are x[0], x[l], x[n – l]. The value of each subscript can be expressed as an integer constant, an integer variable or an integer expression. The number of subscripts determines the dimension of an array that is x[i] and refers to an element in the one-dimensional array 'x' in the ith location.

Defining an Array

Arrays are defined just as an ordinary variable. The only difference is that each array name should necessarily be accompanied by a size specification. In a single-dimensional array, the size can be specified by positive integer expression enclosed in square brackets.

The general form of defining one-dimensional array is:

storage - class data type name[expression];

Where storage class refers to the storage class of the array [static, auto, extern, register], data type refers to the type of the data that can be stored in the array, expression is a positive integer expression that indicates the number of arrays elements.

A storage class is optional. Default values are automatic for arrays defined within a function or block and external for arrays defined outside of a function. To define an array, you need to define an array size in terms of a symbolic constants and not fixed-integer quality.

Declaration of Arrays

Example:

```
int marks [30]
int data type of an array
marksname of an array
```

The number 30 tells the number of elements of type int in the array. This number is often termed as the dimension of the array. The brackets inform the compiler that they are dealing with arrays.

The dimension used to declare an array must always be a positive integer constant or an expression that can be evaluated to be a constant when the program is compiled.

The method of subscribing arrays is 'C' is different from that used in many other programming languages. The first element in the array is numbered zero. It means that the last element is 1 less than the size of the array. However big the array, its elements are always stored in contiguous memory locations.

The first element in the array is numbered 0. The reason for this unusual numbering scheme is that 'C' was developed to perform lower level operations than most programming languages were designed to deal with. When the first element of an array starts with zero, it becomes easier to perform the address calculation corresponding to a subscript.

Memory Map of An Array

One-Dimensional Array

`int iarrMarks[100]` Declares array `iarrMarks[]` with 100 integers

`float farrNum[100]` Declares array `farrNum[]` with 100 floating point numbers

`char carrName[100]` Declares array `carrName[]` with 100 characters

In the general case, consider an array as $a[l_1..u_1]$

Obviously the above array contains $(u_1 - l_1 + 1)$ elements. In C, the lower index l_1 for an array is 0.

Two-Dimensional array

`Int iarrMatrix[20][30]` Declares two dimensional array with order 20×30 (i.e., 20 rows and 30 columns).

Now consider a two dimensional array in general as $a[l_1..u_1][l_2..u_2]$

Then the above array contains $(u_1 - l_1 + 1)$ rows and $(u_2 - l_2 + 1)$ columns. So the total numbers of elements in the array is $(u_1 - l_1 + 1)(u_2 - l_2 + 1)$

N-Dimensional Array

If we interpret the indices to be N-dimensional $(i_1, i_2, i_3..., i_n)$, then it is called an N-dimensional array.

In the N-dimensional array, if the array is declared as $a[l_1..u_1]$ where l_1 and u_1 are the lower bound and upper bound respectively of its dimension, then the total number of elements $= (u_2 - l_2 + 1)(u_2 - l_2 + 1) \cdots (u_n - l_n + 1) = \prod_{i=1}^{n} (u_2 - l_2 + 1)$

Memory Representation of Array

There are two ways to store an array into memory (1) row major ordering and (2) column major ordering. They are described in this section. Here it is considered that each element takes only byte to store its value.

Row Major Ordering

In row major ordering the rows of the array are stored first. Consider an one dimensional array a[l_1 ... u_1]. Suppose its base address (address of the first element) is λ.

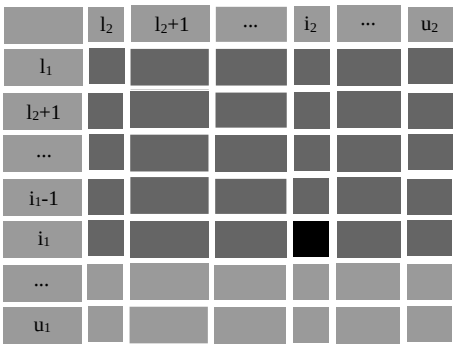
NOTES

Then the address of the element a[i_1] is $\lambda + i_1 - l_1$.

Now consider a 2-dimensional array a [0 .. 3] [0 .. 3]. Its rows and elements in each row are depicted below.

a[0][0]	Row 1
a[0][1]	
a[0][2]	
a[1][0]	Row 2
a[1][1]	
a[1][2]	
a[2][0]	Row 3
a[2][1]	
a[2][2]	
a[3][0]	Row 4
a[3][1]	
a[3][2]	

For a general case suppose the declaration of a 2-dimensional array be a[l_1 …u_1][l_2 …u_2] and the base address is λ



Now to calculate the address of the element a[i_1][i_2], first traverse the i_1-l_1 rows (each row contains $u_2 - l_2 + 1$ elements) and then $i_2 - l_2$ elements. Hence the address of $a[i_1][i_2] = \lambda + (i_1 - l_1)(u_2 - l_2 + 1) + (i_2 - l_2)$.

Now consider a 3-dimensional array a[0 ... 2] [0 ... 2] [0 ... 1].

a[0][0][0]	Row 1	Page 1
a[0][0][1]		
a[0][1][0]	Row 2	
a[0][1][1]		
a[0][2][0]	Row 3	
a[0][2][1]		
a[1][0][0]	Row 1	Page 2
a[1][0][1]		
a[1][1][0]	Row 2	
a[1][1][1]		
a[1][2][0]	Row 3	
a[1][2][1]		
a[2][0][0]	Row 1	Page 3
a[2][0][1]		
a[2][1][0]	Row 2	
a[2][1][1]		
a[2][2][0]	Row 3	
a[2][2][1]		

If a declaration of a 3-dimensional array be $a[l_1 ... u_1][l_2 ... u_2][l_3 ... u_3]$ and the base address be λ then the address of the element

$$a[i_1][i_2][i_3] = \lambda + (i_1 - l_1)(u_2 - l_2 + 1)(u_3 - l_3 + 1) + (i_2 - l_2)(u_3 - l_3 + 1) + (i_3 - l_3)$$

Hence for an N-dimensional array $a[l_1 ... u_1][l_2 ... u_2]...[l_n ... u_n]$ if the base address be λ (i.e., the address of $a[i_1][i_2]..[i_n]$) then the address of

$$\begin{aligned} a[i_1][i_2]..[i_n] = \lambda &+ (i_1 - l_1)(u_2 - l_2 + 1)(u_3 - l_3 + 1)...(u_n - l_n + 1) \\ &+ (i_2 - l_2)(u_3 - l_3 + 1)(u_4 - l_4 + 1)...(u_n - l_n + 1) \\ &+ \quad .... \\ &\quad + (i_n - l_n) \\ &= \lambda + \sum_{j=1...n} (i_j - l_j) a_j \text{ with } a_j = \prod_{\substack{k=j+1 \\ a_n = 1}}^{n} (u_k - l_k + 1) \end{aligned}$$

Column Major Ordering

In this technique we store the columns of an array first. Consider an one dimensional array $a[l_1 ... u_1]$. Suppose its base address (address of the first element) is λ.

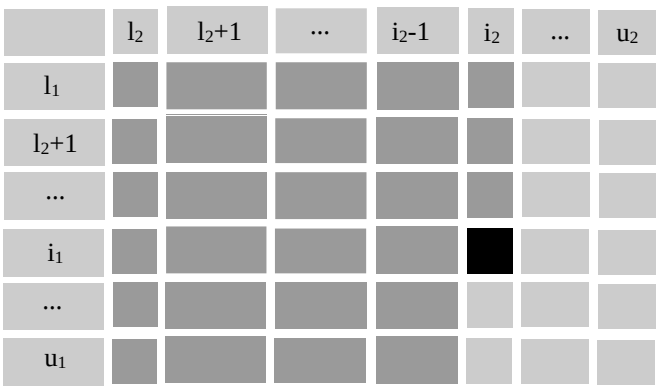
Then the address of the element $a[i_1]$ is $\lambda + i_n - l_n$

NOTES

Consider a 2-dimensional array a[0 … 3] [0 … 2]. Its columns and elements in each column are depicted below.

a[0][0]	Column 1
a[1][0]	
a[2][0]	
a[3][0]	
a[0][1]	Column 2
a[1][1]	
a[2][1]	
a[3][0]	
a[0][2]	Column 3
a[1][2]	
a[2][2]	
a[3][2]	

Suppose the declaration of a 2-dimensional array be $a[l_1 ... u_1][l_2 ... u_2]$ and the base address be λ



Now to calculate the address of $a[i_1][i_2]$ first traverse the $i_2 - l_2$ columns (each column has $u_1 - l_1 + 1$ elements) and then $i_1 - l_1$ elements. Hence the address of $a[i_1][i_2] = \lambda + (i_1 - l_1) + (i_2 - l_2)(u_1 - l_1 + 1)$

Now consider a 3-dimensional array a[0 … 2] [0 … 2] [0 … 1].

a[0][0][0]	Column 1	Page 1
a[0][1][0]		
a[0][2][0]		
a[0][0][1]	Column 2	
a[0][1][1]		
a[0][2][1]		
a[1][0][0]	Column 1	Page 2
a[1][1][0]		
a[1][2][0]		
a[1][0][1]	Column 2	
a[1][1][1]		
a[1][2][1]		
a[2][0][0]	Column 1	Page 3
a[2][1][0]		
a[2][2][0]		
a[2][0][1]	Column 2	
a[2][1][1]		
a[2][2][1]		

If a declaration of a 3-dimensional array be $a[l_1...u_1][l_2...u_2][l_3...u_3]$ and the base address be λ then the address of

$$a[i_1][i_2][i_3] = \lambda + (i_1 - l_1) + (i_2 - l_2)(u_1 - l_1 + 1) + (i_3 - l_3)(u_2 - l_2 + 1)(u_1 - l_1 + 1)$$

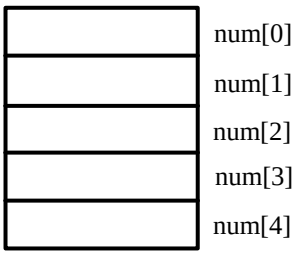
Hence for an N-dimensional array $a[l_1...u_1][l_2...u_2]...[l_n...u_n]$, if the base address be λ (i.e. the address of $a[l_1, l_2, ..., l_n]$ then the address of

$$\begin{aligned} a[i_1][i_2]...[i_n] = \lambda &+ (i_1 - l_1) \\ &+ (i_2 - l_2)(u_1 - l_1 + 1) \\ &+ \quad .... \\ &+ (i_n - l_{2n})(u_{n-1} - l_{n-1} + 1)...(u_1 - l_1 + 1) \end{aligned}$$

$$= \lambda + \sum_{j=1...n} (i_j - l_j) a_j \text{ with } a_j = \prod_{\substack{k=j-1 \\ a_1=1}}^{1} (u_k - l_k + 1)$$

Let us assume you have an array declaration as follows: `int num[5];`

Since we know that array elements are stored in one continuous memory location. This is how the memory map looks:



Example 2.14: To find the average marks obtained by class of 30 students in a test

```
main( ) {
float avg,sum = 0;
int marks [ 30 ], i;
for ( i = l ; i <= 29; i++)
{
    printf( "Enter the marks");
    scanf("%d", &marks[i]);
    for ( i = 0 ; i <= 2 9; i++ )
    sum += marks [ i ];
    avg = sum/30; printf ( "Average is %f ", avg);
}
}
```

Insertion from One-Dimensional Array

Algorithm

```
1. Algorithm fnlnsertion_into_ 1D_Array(arrData, n, k, item)
2. // Purpose : This algorithm inserts an element into
   // one-dimensional array.
3. // Input : arrData[] is a one-dimensional array with n
   //number of elements. Element item is to be inserted into
   // the kth position in the array.
4. // Output : None.
5. {
6. for(i = n – l; i >= k – l; i– –)
7. arrData [i + l]= arrData [i];
8. arrData[k " l] = item; // Insert the item.
9. n = n + l; // Set size of the array.
10. }// End of Algorithm.
```

Deletion from One-Dimensional Array

Algorithm describes the process of deletion from one dimensional array.

NOTES

Algorithm

```
1. Algorithm fnDeletion_from_lD_Array (arrData, n, k)
2. // Purpose: This algorithm deletes an element from one-
   dimensional array.
3. // Input: arrData[] is an one-dimensional array with n
   number of elements. Element item is to be deleted from
   the kth
   // position of the array.
4. // Output : Deleted element item.
5. {
6. item = arrData[k – l]; //Item deleted.
7. for(i = k – l;i < n – l; i++)
8. arrData[i] = arrData[i + l];
9. n = n – l; //Set size of the array.
10. return item;
11. }//End of Algorithm
```

A program to copy one string to other:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int i, j;
    char str[20],s[20];
    clrscr();
    printf("enter the string:");
    gets(str);
    for(i = 0;str[i]!= '\0';i ++)
       s[i] = str[i];
    s[i]= '\0';
    printf("string after copying is: ");
    puts(s);
    getch();
}
```

OUTPUT

```
enter the string: good morning
string after copying is: good morning
```

A program to extract a portion of a string:

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

```
#include<string.h>
void main()
{
    int i, j, pos, n;
    char str[20];
    clrscr();
    printf("enter the string: ");
    gets(str);
    printf("enter the position of extraction: ");
    scanf("%d",&pos);
    printf("how many characters: ");
    scanf("%d",&n);
    printf("extracted string is :");
    pos—;
    for(i = pos;i < pos + n; i++)
    printf("%c ",str[i]);
    getch();
}
```

OUTPUT

```
enter the string: good morning
enter the position of extraction: 3
how many characters: 5
extracted string is: od mo
```

A program to count characters, words and lines in a text:

```
#include<stdio.h>
#include<conio.h>
#include<string,h>
void main()
{
    char str[20];
    int i = 0, w = 0, n = 0, l = 0, ch;
    clrscr();
    printf("enter the text(use @ to end the text): ");
    while((ch = getchar())!='@')
    {
       str[i]=ch;
       i++;
    }
    str[i] = '\0';
    while(str[i]!= '\0')
```

```
    {
    if(str[i]==' '\\ str[i]=='\t')
       w++;
    else
    if(str[i]=='\n')
       l=l+1;
    else
       n++;
    }
    printf("no of words : %d",w);
    printf("no of lines: %d",l);
    printf("no of characters : %d ",n);
    getch();
    }
```

OUTPUT

```
Enter the text (use @ to end the text):
welcome to SMN
welcome to CSE @
no of words: 6
no of lines: 2
no of characters: 28
```

Write a program to check whether a given number is prime or not:

```
#include<stdio.h>
main()
{
 int flag,num,a;
 char ch;
 do
 {
    printf("\nEnter a value:");
     scanf("%d", &num);
     a = 2;
     while(a <= num/2)
 {
 if(num%a == 0)
 {
      printf("Not ");
      break;
 }
 }
```

NOTES

```
 printf("Prime number");
 printf("\nAnother calculation (y/n)?...");
 fflush(stdin);
 scanf("%c",&ch);
 }while(ch!='n');
}
```

OUTPUT

```
Enter a value: 2
Prime number
Another calculation (y/n)?...y
Enter a value: 8
Not Prime number
Another calculation (y/n)? ...n
```

Explanation: A prime number is a number, which is divisible by one and itself only. To find out whether a given number is a prime or not, the given number will be divided with first half elements of the number. If the remainder is zero for any of the divisions, then the given number is not a prime number; else, it is a prime number.

Array Manipulation (Removing the Duplicates)

The array manipulation algorithm works with replicating arrays, inserting, replacing, extracting and shifting the elements. The following algorithm describes how to remove duplicate elements in a given array. For this, the three 'for' loops are used, and the 'if' statement is used within for loops to search duplicate values. The algorithm looks at each element in the array and checks if it matches the given values in the 'if' expression. It also compares the very first value to the rest values. The process is continued till finally all the smallest array elements are checked.

Algorithm to remove the duplicates from array:

```
Step 1: integer i, j, k, num, ans;
Step 2: ans←0
Step 3: integer array_elem[25]; //Array is defined
Step 4: print 'Enter size of array [maximum 25]';
Step 5: read num; //Accept input values for num
Step 6: print 'Enter the elements of array';
Step 7: for i←0 to num
read array_elem[i];
//Accept input values for array_elem[i]
Step 8: for i←0 to num
print array_elem[i];
//Print the inputted values for array_elem[i]
Step 9: for i←0 to num-1
For j←i+1 to num
```

```
Step 10: if array_elem[i]== array_elem[j]
//Assign array_elem[i] is equal to array_elem[j]
num←num-1;
//Decrease the num value by one
for k←j to num
//Inner for loop starts
array_elem[k] ← array_elem[k+1];
//Array elements of k is assigned and increased by one
and check the duplicate values
ans←1;
//ans variable is increased by 1 if duplicate value is
found.
j←j-1; //j variable is decreased by 1
Step 11: if ans==0 //Check the value of ans variable as 0
print 'Array is without duplicates';
else
print 'Array after deleting duplicating';
Step 12: for i←0 to num
print array_elem[i];
//Print array elements without duplicates
```

In the preceding algorithm, the size of `array_elem` array is defined as `25` of integer data type. This means the maximum array elements that can be accepted are `25`. The three 'for' loops `i, j, k` are run. The conditional statement `if array_elem[i]==array_elem[j]` checks according to incrementing for loop `i` and `j`. The statement `num=num-1` is used to reach the last to first value for checking the duplicate values. If any duplicate values are found, it goes to `ans` variable to store the duplicate values. The statement `j¬j-1`; under `k` for loop is used to control the inner `k` loop because the `k` loop checks the values as the values assigned for `k` loop is started from `j`.

Implementation of removing the duplicates from array elements:

```
#include <stdio.h> //Declaring header files
#include <conio.h>
void main() //main() function starts
{
clrscr(); //clears the screen
int i,j,k,num,ans=0; //variables are declared
int array_elem[25]; //array size is declared
printf("\nEnter size of array [maximum 25]");
scanf("%d",&num); //Accept input value for running for
loops
```

NOTES

```
printf("\nEnter the elements of array");
for(i=0;i<num;i++)
scanf("%d",& array_elem[i]); //Accept array input elements
for(i=0;i<num;i++)
printf("\n%d", array_elem[i]);
for(i=0;i<num-1;i++)
for(j=i+1;j<num;j++)
{
if(array_elem[i]==array_elem[j])
//Assumed the array_elem[i] is equal to array_elem[j]
and the body of if control statement starts.
{
num=num-1;
for(k=j;k<num;k++)
array_elem[k]= array_elem[k+1];
```

//Assign `array_elem[k]` is equal to `array_elem[k+1]`. This means checking the k value from the very next value. For example, let us say `k=2`, then `array_elem[2]` value is checked from `array_elem[3]`, and if the previous value is duplicate the same value it goes to ans variable otherwise the third value is checked for the very next fourth value of the array.

```
ans=1;
//Increase the value of variable ans if duplicate value
comes
j=j-1;
```

//Decrease the j value as duplicate values come (for example, 2 duplicate values can decrease the k loop for processing only for 3).

```
}
}
if(ans==0)
//Check the ans variable as 0 that means no duplicate
value comes in array.
printf("\n Array is without duplicates\n");
else
{
printf("\n Array after deleting duplicating");
for(i=0;i<num;i++)
printf("\n", array_elem[i]); //Print the array elements
without duplicate values.
}
getch();
}
```

The result of the preceding program is as follows:

```
Enter size of array [maximum 25]
5
Enter the elements of array
3 56 7 7 12
3
56
7
7
12
Array after deleting duplicating
3
56
7
12
```

If no duplicate values come, the above program prints as 'Array is without duplicates'.

Partitioning of an Array

The classical process of partitioning an array into subarrays can be extended to a more useful array language operation. There are different modes of partitioning defined for different types of arrays and thus subarrays may vary over the original array in an arbitrary manner. Such partitions have dynamic tree structures to derive and maintain the array control information.

An array is a list. To sort out a list in an array we employ a divide and conquer strategy in quick sort, to divide a list into two sublists. This is a partition operation.

The steps followed are:

1. Choose an element from the list. This element is called a *pivot*.
2. Reorder the list. Elements less than the pivot come before it and elements greater than the pivot come after it. If there are equal values, they can go either way. After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Sort both the sublists (of lesser elements and greater elements) recursively.

Base case of the recursion is; lists of size zero or one.

In simple pseudocode, the algorithm might be expressed as this:

```
function quicksort(array)
var list less, greater
if length(array) ≤ 1
return array
select and remove a pivot value pivot from array
```

NOTES

```
for each x in array
if x ≤ pivot then append x to less
else append x to greater
return concatenate(quicksort(less), pivot, quicksort
(greater))
```

Here, elements are examined by comparing them to other elements. Thus, this can also be described as a comparison sort. The partition algorithm is based on the following two facts:

- After iteration, elements processed are in the desired position, before the pivot if less than or equal to the value of the pivot and more if after the pivot.
- On iteration, fewer elements are left to be processed.

The disadvantage of the above version is that it needs Ù(*n*) extra storage space. There is another version more complex that uses an in-place partition algorithm. It has *O*(*logn*) space requirement that does not include the input. The algorithm is:

```
function partition(array, left, right, pivotIndex)
pivotValue := array[pivotIndex]
swap array[pivotIndex] and array[right] // Move pivot to
end
storeIndex := left
for i from left to right –1
if array[i] d" pivotValue
swap array[i] and array[storeIndex]
storeIndex := storeIndex + 1
swap array[storeIndex] and array[right] // Move pivot to
its final place
return storeIndex
```

This algorithm is an 'in-place partition algorithm'.

Suppose there are *n* items in an array and it has to be partitioned into *p* intervals in a way that the maximum weight of the intervals is minimized. This problem has bound $O(n+p^{1+}\varepsilon)$ for $\varepsilon<1$. However, there is an algorithm that has run time of $O(n \log n)$. This algorithm is very fast for an arbitrary value of *p*. We generalize the case of partitioning to two dimensions. We partition an $n \times n$ array of items into p^2 blocks. For this, we partition rows and columns, each into *p* intervals and consider these blocks that result due to this partition. The problem of partitioning requires us to find a partition that minimizes the maximum weight among these blocks. This problem is NP-hard.

These problems are found in load balancing for parallel machines and data partitioning in parallel languages. These find applications estimation of motion by block, matching in video and image compression cause dual problem. One problem is of minimizing the number of dividers *p* in a way that maximum weight of a block

is at most d. This problem has an $O(\log n)$ approximation algorithm. Results of two dimensional array partitioning extend to a higher dimension which is fixed.

Complexity of Algorithms

NOTES

An algorithm is a stepwise procedure for performing some task in a finite amount of time. Sometimes we need to know how much time and space (computer memory) a computer algorithm requires, i.e. how efficient it is. This is termed as **time** and **space** complexity. Typically, the complexity refers to a function of the values of the inputs, and we would like to know what is that function. The best, average and worst cases can also be considered.

The big *O* notation: The big O notation provides a convenient way to compare the speed of algorithms. This is a mathematical notation used in the priori analysis. If an algorithm is said to have a computing time of $O(g(n))$, then it implies that if the algorithm is run on some computer on the same type of data put for increasing the values of n, the resulting times will always be less than same constant times $|g(n)|$.

The best algorithm runs in $O(1)$ times. Good algorithm runs in $O(\log N)$ times. Fair algorithm runs in $O(N)$ times. Worst algorithm runs in $O(N^2)$ times.

Note: If $A(n) = a_m n^m + \cdots + a_1 n^1 + a_0$ is a polynomial of degree m, then $f(n) = O(n^m)$. Thus, if the frequency of execution of a statement is in the form of $A(n)$, then the statements computing time will be $O(n^m)$.

Formally, $O(g(n))$ is the set of functions, f, such that for some $c > 0$, $f(n) < cg(n)$ for all positive integers, $n > N$, i.e. for all sufficiently large N. It can be represented as $\lim_{n\to\infty} \frac{f(n)}{g(n)} \le c$.

Informally, we say the $O(g)$ is the set of all functions, which grows no faster than g. The function g is an upper bound to functions in $O(g)$.

We can analyse any algorithm by the O notation irrespective of the programming language and machine.

Consider two other functions: $\Omega(g)$ and $\Theta(g)$.

$\Omega(g)$ is the set of functions $f(n)$ for which $f(n) \ge cg(n)$ for all positive integers, $n>N$, and $\Theta(g) = \Omega(g) \cap O(g)$

Properties of the Big *O* Notation

1. Constant factors may be ignored: For all $k > 0$, kf is $O(f)$,

   e.g. cn^2 and kn^2 are both $O(n^2)$.

2. Higher powers of n grow faster than lower powers: n^r is $O(n^8)$ if $0 \le r \le s$.

3. The growth rate of a sum of terms is the growth rate of its fastest growing term: If f is $O(g)$, then $f+g$ is $O(g)$.

   e.g. an^3+bn^2 is $O(n^3)$.

NOTES

4. The growth rate of polynomial is given by the growth rate of its leading term: If f is a polynomial of degree d, then f is $O(n^d)$.
5. If f grows faster than g, which grows faster than h, then f grows faster than h.
6. The product of upper bounds of functions gives an upper bound for the product of the functions: If f is $O(g)$ and h is $O(r)$, then fh is $O(gr)$,

   e.g. If f is $O(n^2)$ and g is $O(\log n)$, then fg is $O(n^2 \log n)$.
7. Exponential functions grow faster than powers: n^k is $O(b^n)$, for all $b > 1$, k,

   e.g. n^4 is $O(2^n)$ and n^4 is $O(\exp(n))$.
8. Logarithms grow more slowly than powers: $\log_b n$ is $O(n^k)$ for all $b > 1$, $k > 0$

   e.g. $\log_2 n$ is $O(n^{0.5})$.
9. All logarithms grow at the same rate: $log_b n$ is $\Theta(\log_d n)$ for all $b, d > 1$.
10. The sum of the n rth powers grows as the $(r + 1)$th power: $\sum_{k=1}^{n} k^r$ is $\Theta(n^{r+1})$,

    e.g. $\sum_{k=1}^{n} i = \frac{(n+1)n}{2}$ is $\Theta(n^2)$

General Rules

1. **Simple statement sequence:** It is to be noted first that a sequence of statements executed once only is $O(1)$. It is immaterial as to how many statements are in the sequence; only that the number of statements (or the time that they take to execute) is constant for all problems.
2. **Simple loops:** If a problem of size n can be solved with a simple loop. For example,

```
for (i = 0;i < n; ++i)
{
   Statement(s);
}
```

   Where Statement(s) is an $O(1)$ sequence of statements, then the time complexity is $nO(1)$ or $O(n)$.
3. **Nested loops:**

```
for(j = 0;j < n; ++j)
       for(i = 0;i < n; ++i)
       {
```

```
    Statement(s);
  }
```

when we have n repetitions of an $O(n)$ sequence, then the complexity is $nO(n)$ or $O(n^2)$.

NOTES

4. **Loop index does not vary linearly:** Where the index jumps by an increasing amount in each iteration.

```
i = 1;
while(i ≤ n)
{
 Statement(s);
 i = 2*i;
}
```

in which i takes values 1, 2, 4,… until it exceeds n. This sequence has $1 + \lfloor \log_2 n \rfloor$ values, so the complexity is $O(\log_2 n)$.

5. **If the inner loop depends on an outer loop index:**

```
for(j = 0;j < n; j++)
   for(i = 0;i < j; i++)
   {
   Statement(s);
   }
```

The inner loop $i = 0, 1, 2 \ldots n$ gets executed n times, so the total is:

$\sum_{1}^{n} i = \frac{n(n+1)}{2}$ and the complexity is $O(n^2)$.

Note that the two nested loops also have the same complexity, so the variable number of iterations of the inner loop does not affect the 'big picture'. However, if the number of iterations of one of the loops decreases by a constant factor with every iteration as shown here:

```
i = n;
while(i > 0)
{  for(i = 0;i < n; ++i)
 {
    Statement(s);
   }
 h = h/2;
}
```

Then there are $\log_2 n$ iterations of the outer loop and the inner loop is $O(n)$. So the overall complexity is $O(n \log n)$.

NOTES

The most common computing times of algorithms in the big *O* notation are:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) <= O(n!)$$

Finding Prime Factor of a Given Number

Finding a prime factor begins with the lowest prime number 2. If 2 divides the number completely and leaves no remainder it is marked as the very first prime factor. It continues dividing until it longer divides evenly. Then the control flow moves to the next lowest prime numbers. The step is repeated until the next prime factor comes. The following algorithm is used to find the prime factor of a given number:

Algorithm to find prime factor of a given number:

```
Step 1: integer input, divisor, count;
Step 2: print 'Enter a value:',
Step 3: read input;
Step 4: count←0;
Step 5: Do
Step 6: divisor←0;
Step 7: if input mod 2==0 OR input==1
//To remove all the factors of 2
break;
count←count+1; //Increase counter value by 1
print count, divisor;
input←input/2; //Remove this factor from input
Step 8:End Do
Step 9: divisor←3;
Step 10: Do
Step 11: if divisor>input
break;
Step 12: Do //Remove the factors repeatedly
Step 13: if input mod divisor ==0 OR input==1
break;
count←count+1;
print count, divisor;
input←input/divisor;
//Remove factors from input
Step 14: End Do
Step 15: divisor← divisor+2;
//Move to next odd number
Step 16: End Do
```

The preceding algorithm lists out all prime factors of an `n integer >=2`. First it sides back all factors of `2`. Then, all factors, such as `3, 5, 7` and so on can be removed. This process is run until all the prime factors are sided back and kept in a temporary location. According to the above algorithm, if the input value is 53, the prime factors of 53 are 1 and 53 itself.

Implementation of finding prime factors of a given number:

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h>
#include <conio.h>
void main()
{
int number, i, j, k;
clrscr();
printf("Enter a number:");
scanf("%d", &d);
while(i<=number)
{
k=0;
if (number%i==0)
{
j=1;
while(j<=i)
{
if(i%j==0)
k++; //Value k is increased by 1
j++; //Value j is increased by 1
}
if(k==2)
printf("\nPrime factors are:");
printf("%d",i);
i++;
}
getch();
}
```

The result of the above program is as follows:

```
Enter a number: 123
Prime factors are: 3 41
```

In the above program, finding a prime factor of a given number can be performed in the following step. You first enter a number lets say '123' as input value. The prime factors of 123 are 3 and 41 (prime numbers). If you multiple 3 and 41, it returns 123, that is a prime number.

NOTES

List of Prime Numbers

Prime numbers are numbers that can be divided only by 1 or by themselves. Below, in white, are the prime numbers between 1 and 100.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

The following algorithm is used to find the list of prime numbers:

Algorithm to find the list of prime numbers:

Step 1: integer N,D;

// Declare the variables the integer being considered needed for the integer divison

Step 2: integer N_is_prime;

// N is equal to 1 (default) when N is prime and N = 0 when N is not prime

Step 3: for N←3 to 30 //Running for loop

// This loop considers all prime integers between 3 and 30

N_is_prime←1;

// assume N is prime

Step 4: for D←2 to (N-1)

if N%D == 0

//Returns remainder if N is divided by D

N_is_prime = 0;

// if the remainder is 0 then N is prime

Step 5: if N_is_prime == 0

break; //Exit from loop

// if N is prime do not do any more integer divisions

Step 6: if N_is_prime == 1

print N;

Implementation to find the list of prime numbers:

```
/*———————— START OF PROGRAM ————————*/
#include<stdio.h>
#include <conio.h>
```

```
void main()
{
int N; // the integer being considered
int D; // needed for the integer divison
int N_is_prime; // = 1 (default) when N is prime and = 0
when N is not prime
for (N=3;N<=30;N++)
// This loop considers all prime integers between 3 and
100
{
N_is_prime = 1; // assume N is prime
for (D=2;D<=N-1;D++)
{
if ( N%D == 0 ) N_is_prime = 0;
// if the remainder is 0 then N is prime
if (N_is_prime == 0)
break;
// if N is prime don't do any more integer divisions
}
if (N_is_prime == 1)
printf("%d\n",N);
}
getch();
}
```

The result of the preceding program is as follows:

```
2
3
5
7
11
13
17
19
23
29
```

Problem of Search

In computer science, a search algorithm refers to an algorithm taking a problem as input and returning a solution to the problem, usually after it evaluates of the various possible solutions. Most problem-solving algorithms studied by computer scientists

are search algorithms. Search space refers to the set of all possible solutions to a problem. In the Brute-force search, commonly known as naïve or uninformed algorithms, the simplest method of the searching through the search space is used, whereas in informed search algorithms, heuristic functions are used for applying knowledge about the structure of the search space so that the amount of time spent in searching is reduced.

Searching refers to an operation of finding the location of an item in a table or a file. Depending on the location of the records to be searched, searching techniques are classified into the following two types:

External searching: When the records are stored in files or disk or tape or any secondary storage, then the searching is known as external searching.

Internal searching: When the records to be searched are stored entirely within computers memory, then it is known as internal searching.

The locating of a particular element in a data structure is called *searching*. Generally, the methods are used for searching are as follows:

1. Linear search
2. Binary search
3. Fibonacci search

Linear Search or Sequential Search

Linear search is the easiest and least efficient searching technique. In this technique, the given list of elements are scanned from the first one till either the required element is found or the list is exhausted. This technique is used in direct access media such as magnetic tapes.

Example of linear search: Find an element 77 from the given list using linear search. The list of elements is: 10, 25, 77, 16, 47, 98

Linear search starts by checking the target element (i.e. 77) with the first element of the list, i.e. 10, which is not equal to the target element; search continues with the second element, i.e. 25, which is also not equal to the target element and search continues with the third element, i.e. 77, which is equal to the target element (=77). So, the search is stopped.

Algorithm for linear search:

```
LINEAR_SEARCH (L, N, E)
    1. [Initialization]
      Loc = 1
      L[N + 1] = E
    2. [Search the element in the vector]
      REPEAT WHILE ( K[Loc]<> E ) DO
         Loc = Loc + 1
```

```
3. [Check whether the search is successful or not?]
  IF Loc = N + 1, THEN WRITE ('UNSUCCESSFUL SEARCH')
     RETURN(0)
  ELSE WRITE('SUCCESSFUL SEARCH')
     RETURN(Loc)
```

Analysis of Linear Search Algorithm

For N total number of elements, the search time T is proportional to half of N:

```
T = K * N/2 where K is a constant
If K = 2, then T = K*N
```

The average linear search times are proportional to the size of the array, i.e. *O*(*N*)

Note: If an array is twice as big, it will take twice as long to search.

Implementation of Linear Search to Find a String from a String Vector/ Array

Program for linear search of strings:

```
/*——————START OF PROGRAM——————*/
   #include<stdio.h>
   #define MAXROWS 10
   #define MAXCOLS 20
   #define NOTFOUND -1
   typedef char STRINGS[MAXROWS][MAXCOLS];
   typedef char STRING[MAXCOLS];
   int LSearch(STRINGS s,STRING target,int n)
   {
    int loc=0;
    strcpy(s[n],target);
    while(strcmp(s[loc],target))
      loc++;
    if(loc==n)
      return NOTFOUND;
    else
      return loc;
   }
   void main()
   {
   STRINGS  a={"MON","TUE","WED","THU","FRI",
"SAT","SUN"};
   int index;
   index=LSearch(a,"WED",7);
   if(index==NOTFOUND)
         printf("Record not found");
```

NOTES

```
    else
          printf("Record found at Location:%d",index+1);
    }
    /*————————END OF PROGRAM————————*/
```

Output: Record found at Location 3

Implementation of Linear Search to Find a Value in a Vector or Array

```
/*————————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#define MAXROWS 10
#define MAXCOLS 20
#define NOTFOUND -1
typedef int VECTOR[MAXCOLS];
int LSearch(VECTOR s,int target,int n)
{
 int loc=0;
 s[n]=target;
 while(s[loc]!=target)
     loc++;
 if(loc==n)
  return NOTFOUND;
 else
 return loc;
}
void main()
{
VECTOR a={5,4,3,2,7};
int index;
clrscr();
```

Program for linear search of numbers:

```
index=LSearch(a,2,5);
if(index==NOTFOUND)
printf("Record not found");
else
printf("Record found at Location:%d",index+1);
}
/*————————END OF PROGRAM————————*/
```

Output: Record found at Location:4

Binary Search

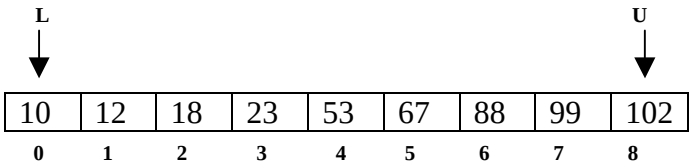
This search is used to search for an element in a sorted list.

The search method:

NOTES

- First compare the key with the item in the middle position of the array.
- If any match is found, return it immediately.
- If the key is less than the middle key, then the item to be found must lie in the lower half of the array; if it is greater, then the item to be found must lie in the upper half of the array.
- Repeat the procedure on the lower (or upper) half of the array.

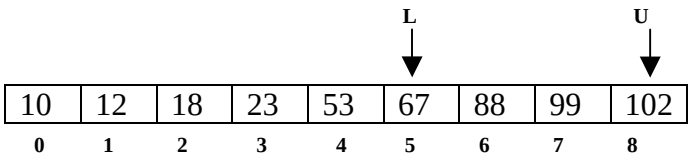
Example of binary search: Find an element 88 in an array of elements given below where **L** is lower bound of the array and **U** is the Upper bound of the array.



Calculate middle by **M**=FLOOR((**L**+**U**)/2), where **L**=0 and **U**=8

M=4

Since value in **Vector[M]** is 53, which is less than the target value (=88), search in the second half of the array.



Calculate middle by **M=**FLOOR((**L**+**U**)/2), where **L**=5 and **U**=8

M=6

Since value in **Array[M]** is equal to target value (=88), then it is a successful search and record found at location 6.

Algorithm for Binary Search

```
BINARY_SEARCH(B,N,E)
1. [Initialization]
   L=1
   H=N
2. [Start the searching process]
   REPEAT THRU STEP 4 WHILE L<=H DO
3. [Get the index of midpoint of interval]
   M=FLOOR(L+H)/2
4.[Comparison to get the element]
  IF E < B[M] THEN
```

NOTES

```
    H=M-1
  ELSE
    IF E > B[M] THEN
       L = M +1
    ELSE
        WRITE('SUCCESSFUL SEARCH')
     RETURN(M)
5. [Unsuccessful search]
  WRITE('UNSUCCESSFUL SEARCH')
6.[Finished]
  RETURN(0)
```

Analysis of Binary Search Algorithm

For *N* total number of elements, the search time *T* is proportional to log*(N) T=K* * $\log_2(N)$.

The average searching time for binary search is $O(\log N)$.

Implementation of Binary Search to Find an Element in a Sorted Vector/ Array

Program for binary search for numbers:

```
*————————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
#define NOTFOUND -1
typedef int VECTOR[MAXCOLS];
int BSearch(VECTOR str,int target,int n)
{   int s,e,m,cmp;
s=0;
e=n-1;
while(s<=e)
{   m=(s+e)/2;
if(target<str[m])
          e=m-1;
          else
          if(target>str[m])
             s=m+1;
          else  return m;
 }
 return NOTFOUND;
 }
```

```
void main()
{
VECTOR a={1,2,3,4,5};
int index;
clrscr();
index=BSearch(a,4,5);
if(index==NOTFOUND)
printf("Record not found");
else
       printf("Record found at Location:%d",index+1);
}
/*————————————END OF PROGRAM————————————*/
Output:
Record found at Location:5
```

Implementation to Search a String in a Vector/Array Having Strings in Sorted Order

```
/*————————————START OF PROGRAM————————————*/
#include<stdio.h>
#include<conio.h>
#define MAXROWS 10
#define MAXCOLS 20
#define NOTFOUND -1
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
int BSearch(STRINGS str,STRING target,int n)
{
    int s,e,m,cmp;
s=0;
    e=n-1;
while(s<=e)
{
m=(s+e)/2;
cmp=strcmp(target,str[m]);
if(cmp<0)
          e=m-1;
 else
          if(cmp>0)
            s=m+1;
          else
          return m;
 }
```

NOTES

```
 return NOTFOUND;
 }
void main()
{
STRINGS str={"AB","ABC","BB","BCA","CC","CCC"};
int index;
clrscr();
index=BSearch(str,"CC",6);
if(index==NOTFOUND)
printf("Record not found");
else
    printf("Record found at Location:%d",index+1);
}
/*————————END OF PROGRAM———————————*/
Output:
Record found at Location:5
```

Note: In the preceding two programs, the array should contain sorted values; otherwise, use any sorting algorithm before calling **BSearch.**

Algorithm for Binary Search Using Recursive Technique

Function BSearch(Vector,First_Index,Second_Index, Target)

```
1.[Search vector between First_Index and Second_Index
       for target]
 IF First_Index>Second_Index)
     Loc=0
 ELSE
  Middle_Index=(First_Index+Second_Index)/2;
  IF Target > Vector[Middle_Index]

 Loc=Bsearch(Vector,Middle_Index+1,Second_Index,Target)
     ELSE
      IF Target < Vector[Middle_Index]
       Loc=Bsearch(Vector,First_Index,Middle_Index-1,
Target)
       ELSE
        Loc=Middle_Index
2.[Finished]
 RETURN(Loc)
```

Implementation of Binary Search to Find an Element in a Sorted Vector/ Array Using the Recursion Technique

NOTES

Program for binary search for numbers using recursion:

```
#include <stdio.h>
#define MAXCOLS 20
#define NOTFOUND -1
typedef int VECTOR[MAXCOLS];
int bsearch(VECTOR vector,int findex,int sindex,int
target)
{
 int mindex,loc;
 if(findex>sindex)
    loc=NOTFOUND;
 else
 {
  mindex=(findex+sindex)/2;
    if(target>vector[mindex])
       loc=bsearch(vector,mindex+1,sindex,target);
    else
       if(target<vector[mindex])
         loc=bsearch(vector,findex,mindex-1,target);
       else
         loc=mindex;
 }
return(loc);
}
void main()
{
 VECTOR a={10,20,30,40,50,60,70,80,90};
 int loc;
 loc=bsearch(a,0,8,40);
 if(loc==NOTFOUND)
 printf("Target string not found");
 else
  printf("Starting from 0th location target is at
location:%d",loc);
}
```

Output

```
Starting from 0th location target is at Location:3
```

NOTES

Implementation to Search a String in a Vector/Array Having Strings in Sorted Order Using the Recursion Technique

Program for binary search for strings using recursion:

```
#include <stdio.h>
#include <string.h>
#define MAXCOLS 20
#define MAXROWS 10
#define NOTFOUND -1
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
int bsearch(STRINGS str,int findex,int sindex,STRING
target)
{
 int mindex,loc,cmp;
 if(findex>sindex)
     loc=NOTFOUND;
 else
 { mindex=(findex+sindex)/2;
     cmp=strcmpi(target,str[mindex]);
     if(cmp<0)        /* if target greater than middle
string */
      loc=bsearch(str,mindex+1,sindex,target);
     else
     if(cmp>0) /* if target less than middle string */
       loc=bsearch(str,findex,mindex-1,target);
       else
          loc=mindex;
 }
return(loc);
}
void main()
{
 STRINGS str[]={"aa","bb","cc","dd"};
 int loc;
 loc=bsearch(str,0,3,"bb");
 if(loc==NOTFOUND)
 printf("Target string not found");
 else
```

```
 printf("Starting from 0th location target is at
Location:%d",loc);
}
```

Output

```
Starting from 0th location target is at Location:1
```

NOTES

Fibonacci Search

The Fibonacci progression is a numeric progression such that $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1}+F_{n-2}$ for n_2. The Fibonacci search splits the given list of elements according to the Fibonacci progression unlike splitting in middle as in the binary search.

Algorithm for Fibonacci Search

```
Function Fibonacci_search(Array, Target, N)
1. [Initialize I with 0]
  I = 0
2. [Check ?]
   WHILE(Fib(I) < N)
    I = I + 1
3. [Assignments]
  A = Fib(I – 2)
  B = Fib(I – 3)
4. [Calculate middle element]
 Middle=N"-A"-1
5. [Search process]
 WHILE Array[Middle]<>Target DO
    IF Array[Middle] > Target THEN
        IF b < 0 THEN
           RETURN NOTFOUND
         T = A – B
         Middle = Middle – B
         A = B
         B = T
      ELSE
        IF A < 1 THEN
          RETURN NOTFOUND
         MIDDLE = MIDDLE + B
         A = A – B
         B = B – A
 6.[Finished]
RETURN Middle
```

NOTES

Algorithm for Fibonacci Function

```
Function Fib(N).
1.[Generate number]
 IF N = 0 THEN
  RETURN 0
 ELSE
  IF N = 1 THEN
   RETURN 1
  ELSE
   RETURN Fib(n – 1)+Fib(n – 2).
```

Analysis of Fibonacci search algorithm: Fibonacci numbers grow exponentially, it immediately follows that any node with *N* descendants that has rank at most *O*(log*N*)

The average searching time for Fibonacci search is *O*(log *N*).

Implementation to Search a String in a Vector/Array Having Strings in Sorted Order Using Fibonacci Search

Program for Fibonacci search for strings

```
/*————————STARTING THE PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAXCOLS 20
#define MAXROWS 10
#define NOTFOUND –1
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
int fib(int n)
{ if(n==0)
  return 0;
 else
  if(n==1)
   return 1;
  else
   return fib(n – 1) + fib(n – 2);
}
int fsearch(STRINGS str, STRING target, int n)
{ int i,a,b,middle,t;
 i = 0;
 while(fib(i) < n)
       i++;
 a=fib(i – 2);
```

NOTES

```
 b=fib(i – 3);
 middle = n – a – 1;
 while(strcmpi(str[middle],target)!=0)
 { if(strcmpi(str[middle],target)>0)
     {
        if(b < 0)
           return NOTFOUND;
        t = a – b;
        middle = middle – b;
        a = b;
        b = t;
     }
   else
       { if(a < 1)
           return – 1;
    middle = middle + b;
        a = a – b;
        b = b – a;
       }
 }
 return(middle);
}
void main()
{
 int i,n;
 STRINGS str[]={"aa","bb","cc","dd","ee","ff","gg"};
 i=fsearch(str,"gg",7);
 if(i==NOTFOUND)
 printf("\nRecord not found");
 else
  printf("\nStarting from 0th location record found
at:%d",i);
}
```

Output

```
Starting from 0th location record found at:6
```

CHECK YOUR PROGRESS

5. Which are the two ways to store memory in an array?
6. How does one determine if a number is a prime number?
7. What are the methods used for searching?

Mergesort

The Mergesort algorithm basically works on a divide and conquer strategy in which the sequence is divided into two halves. Each half is independently sorted and then both halves are merged to make a combine sequence. In this process, the validity of input data required in `mergesort` is as follows:

- Check the input sequences. If there is only one element then the `mergesort` operation is not performed.
- The input sequences are separated into two halves.
- Sort the input sequences.
- Merge both sorted input sequences to generate the result.

In the merging process, the elements of two arrays are combined, creating a new array. The algorithm is based on the merging process where all the elements are copied in one array and kept in the separate new array. Then it adds the second array to the new array. After combining the sorted array a `mergesort` array is created. For example, the two arrays `A[5]` and `B[3]` are manipulated and then merged to create a new array. The newly-created array, namely C, will have 5+3=8 elements. The required steps are as follows:

- Compare the very first elements of both `A[0]` and `B[0]`. If `A[0] < B[0]` then the value of `A[0]` is shifted to `C[0]`. Then the size of both arrays `[Arrays A and C]` current pointers are increased by one.
- The elements of array `A` and array `B` are compared where the pointers are pointing, that is, the first element of array A and the null element of `B`, i.e., `A[1]` and `B [0]`.
- If `B[0]<A[1]` then `B[0]` is moved to `C[1]`. The current pointer of `B` is incremented to point the next element in array `B`.

Algorithm to check the sequences of validation of arrays

```
Function Mergesort(M1, M2)
{
list A ← Empty
while (neither M1 nor M2)
{
compare first items of M1 and M2
remove smaller of the M1 and M2 from the list
add to end of A
}
catenate remaining list to end of A
return A
}
```

NOTES

Mergesort problem: Sort a sequence of given n elements in a non-decreasing way. It follows the **DCC** mechanism that represents **D**ivide, **C**onquer and **C**ombine:

Divide: Divides the n-element sequence that is sorted into two subsequences of n/2 elements.

Conquer: Sorts by using `mergesort` the two recursive subsequences.

Combine: Merges both subsequences to produce the sorted result.

The required steps in the `mergesort` algorithm are as follows:

Input: Sort a sequence of n numbers that is stored in an array.

Output: Produce an ordered sequence of n numbers.

The following algorithm is applied in merge sort mechanism:

```
Mergesort(A,m,n) //It sorts A[m...n] by divide and conquer
method
Step 1: if m<n
Step 2: then r←[(m+n)/2]
Step 3: Mergesort (A,m,r)
Step 4: Mergesort (A, r+1, n)
Step 5: Merge(A,m,n,r) //This step merges A[m...n] with
A[r+1...n]
Merge (A,m,n,p)
Step 1: n1←n - m+1
Step 2: n2←p - n
Step 3: for i ← 1 to n1
Step 4: do L[i] ←A[m+i-1]
Step 5: for j ←1 to n2
Step 6: do R[j] ←A[n+j]
Step 7: L[n1+1] ←∞
Step 8: R[n2+1] ←∞
Step 9: i←1
Step 10: j←1
Step 11: for k←m to p
Step 12: do if L[i]<=R[j]
Step 13: then A[k]←L[i]
Step 14: i← i+1
Step 15: else A[k] ←R[j]
Step 16: j← j+1
```

In the above algorithm, `L[i]` and `R[j]` are the smallest elements of `L` and `R` that are not copied back into `A`. Figure 2.4 shows the `mergesort` process that is based on above mentioned algorithm:

NOTES

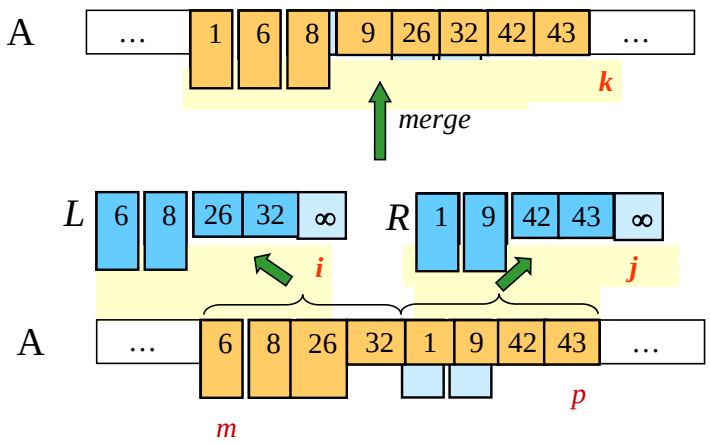


Figure 2.4 *Sort the Elements [1, 6, 8, 9, 26, 32, 42, 43] Using Mergesort*

Analysis of Mergesort Algorithm

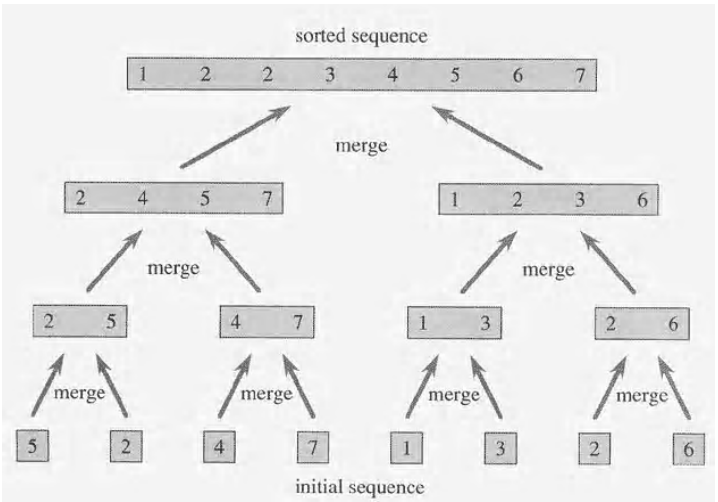


Figure 2.5 *Mergesort Algorithm*

In Figure 2.5 an array A is taken in which there are 8 elements. The operation of merge sort on the array A is [5, 2, 4, 7, 1, 3, 2, 6]. The length of the sorted sequences is merged as the steps required in algorithm from bottom to top.

Implementation of Mergesort for Two Vectors of Seven Elements

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h>
#include <conio.h>
void mergesort(int [], int [], int [], int, int);
void main()
{
    int A_Array[50], B_Array [50], C_Array [100], m, n,
i;
    printf("\n Enter the array elements for first array
[max 50]: ");
```

NOTES

```
    scanf("%d", &m);
     printf("\mEnter the array elements in ascending
order:");
for (i=0; i<m; i++)
        scanf("%d", &A_Array[i]);
    printf("\nEnter the array elements for second array
     [max 50]: ");
    scanf("%d", &n);
    printf("Enter the array elements in ascending order:");
    for (i=0; i<n; i++)
        scanf("%d", &B_Array[i]);
    mergesort(A_Array, B_Array, C_Array, m, n);
    printf("\n The sorted array is : ");
    for (i=0; i<m+n; i++)
        printf("%d\n", C_Array[i]);
   }
void mergesort(int A_Array[], int B_Array[], int C_Array[],
int m, int n)
{
    int a_ele=0, b_ele=0, c_ele=0;
    for (a_ele =0, b_ele=0, c_ele =0; a_ele<m && b_ele<n;)
    {
        if (A_Array[a_ele]< B_Array[b_ele])
//Check the elements of A_Array are less than elements of
B_Array
           C_Array[c_ele++] = A_Array[a_ele++];
//Assign the values of C_Array in A_Array otherwise B_Array
        else
           C_Array[c_ele++] = B_Array [b_ele++];
    }
    if (a_ele<m)
        while (a_ele<m)
           C_Array[c_ele++] = A[a_ele++];
    else
        while (b_ele<n)
           C_Array[c_ele++] = B_Array[b_ele++];
}
```

The arrays `A_Array` and `B_Array` are the input arrays that contain elements in ascending order. Their sizes are `m` and `n` respectively. The `C_array` is the output array containing the elements from the two combined arrays in sorted order.

The result comes in the following way:

```
Enter the array elements for first array [max 50]: 3
Enter the array elements in ascending order:
4
8
10
Enter the array elements for second array [max 50]:4
Enter the array elements in ascending order:
3
5
7
9
```

The sorted array is :

3
4
5
7
8
9
10

Problem of Sorting

In the fields of computer science and mathematics, a sorting algorithm refers to an algorithm whose function is to put elements of a list in a certain order. The numerical and lexicographical orders are the most used orders. In order to optimize the use of other algorithms, such as search and merge algorithms, efficient sorting is essential, as these algorithms require sorted lists to work correctly. Sorting is also often used to canonicalize data and to produce human-readable output. The output must meet the following two conditions:

- The output should be in non-decreasing order (each element should not be smaller than the previous element according to the desired total order).
- The output should be a permutation or reordering of the input.

Since the beginning of computing, the sorting problem has greatly attracted the attention of researchers, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, the analysis of bubble sort was done as early as 1956. Many consider it a solved problem. However, the invention of new sorting algorithms has not stopped. Library sort, for example, was first published in 2004. Sorting algorithms are taught in introductory computer science classes. Students are introduced to a variety of core algorithm concepts, such as big *O* notation, divide and conquer algorithms, data structures, randomized

algorithms, best, worst and average case analysis, time-space tradeoffs and lower bounds.

Sorting is a method of arranging keys in a file in the ascending or descending order. Sorting makes handling of records in a file easier.

Sorting can be classified into the following two types:

Internal sorting: Sorting of records in a file, which is stored in the main memory.

External sorting: Sorting of records in a file, which is stored in the secondary memory. Some sorting techniques are as follows:

- Bubble sort
- Insertion sort
- Selection sort
- Quick sort
- Tree sort
- Arrangement of elements in a list according to the increasing (or decreasing) values of some key field of each element.
- Sorting will be useful to search, insert or delete a data item in a list.

There are various methods for sorting:

Bubble Sort

It comes under the category of exchange sort technique:

- Consider an array `A` has `n` elements `A[0]` to `A[n – 1]`. The array is to be sorted in the ascending order.
- Compare `A[0]` and `A[1]` and arrange such that `A[0] < A[1]`. Then compare `A[1]` and `A[2]` and arrange such that `A[1] < A[2]` and repeat this process till the largest element is bubbled to the nth position.
- Since the largest value is now in the last position as required for the ascending order, consider the first `(n – 1)` elements. Repeat the above process as to bubble the next largest value to `(n – 1)` th position. Then consider the first `(n – 2)` elements and in this way proceed to bubble till all the elements are bubbled to their respective positions. Then sorting will be completed.

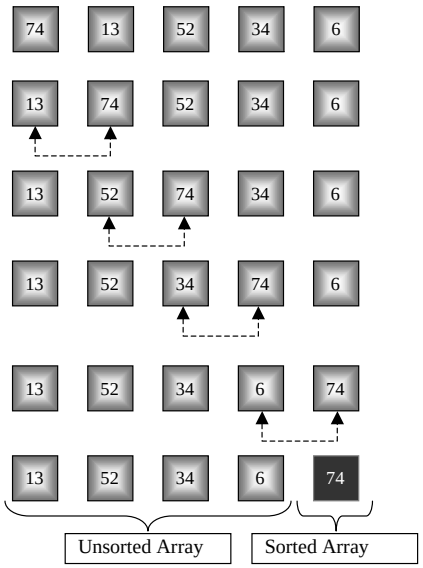
Algorithm for bubble sort or exchange sort

```
BUBBLE_SORT(B,N). Where B is a vector having N elements
1. [Initialization]
  Last = N (entire list assumed unsorted at this point)
2. [Loop on I index]
  REPEAT THRU STEP 5 FOR I = 1 TO N – 1 DO
3. [Initialize exchanges counter for this pass]
  EXS = 0
```

NOTES

```
4. [Compare the unsorted pairs]
  REPEAT FOR J = 1 TO Last – 1 DO
   IF B[J] < B[J+1] THEN
     B[J] ßà B[J+1]
  EXS = EXS + 1
5. [Check whether any exchanges occur or?]
  IF EXS = 0 THEN
RETURN (Sorting finished)
  ELSE
Last = Last – 1(reduce the size of unsorted list)
6. [maximum number of passes finished]
  RETURN
```

Example 2.15: Sort the elements 74, 13, 52, 34, 6 using bubble sort.



Apply the same procedure for the unsorted array and repeat the same process until the elements are not exchanged in any of the pass, then result will be the sorted list: 6, 13, 34, 52, 74.

Implementation of Bubble Sort to Sort Strings of Vector/Array

Program for bubble sort of numbers:

```
/*————————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
#define MAXROWS 10
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
void bub_sort(STRINGS a,int n)
```

```
{
    int i,j;
for(i = 0;i <n – 1; ++i)
{
int pass = 0;
for(j = 0; j < n – 1 – i; ++j)
{
if(strcmp(a[j], a[j + 1]) > 0)
      {
    STRING temp;
    strcpy(temp,a[j]);
          strcpy(a[j], a[j + 1]);
          strcpy(a[j + 1],temp);
          pass = 1;
      }
 }
      if(pass == 0)
    break;
    }
}
void main()
{
    STRINGS a = {“EE”,”BA”,”AB”,”CD”,”AA”};
int i;
clrscr();
bub_sort(a,5);
for(i = 0; i < 5; ++i)
printf(“%s “,a[i]);
}
/*————————END OF PROGRAM————————*/
```

OUTPUT: AA AB BA CD EE

Implementation of Bubble Sort to Sort Integers of a Vector/Array

```
/*————————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
typedef int VECTOR[MAXCOLS];
void bub_sort(VECTOR a,int n)
{
    int i, j;
    for(i = 0; i < n – 1; ++i)
```

NOTES

```
{
        int pass = 0;
        for(j = 0; j < n – 1 – i; ++j){

if(a[j] > a[j + 1])
{
int temp;
  temp = a[j];
  a[j] = a[j + 1];
  a[j + 1] = temp;
  pass = 1;
}
    }
if(pass == 0)
break;
}
}
void main()
{
VECTOR a = {5, 4, 3, 2, 1};
int i;
bub_sort(a, 5);
for(i = 0; i < 5; ++i)
printf("%d ", a[i]);
}
/*————————END OF THE PROGRAM————————*/
```

OUTPUT: 1 2 3 4 5

Selection Sort

Selection sort is a simple sorting technique to sort a list of elements. In this method, first find the smallest value in the array. Exchange it with the first element. Find the next smallest and exchange it with the second element. Continue in this manner till all elements are completed. A disadvantage of selection sort is that its running time depends only slightly on the amount of order already in the given list of elements.

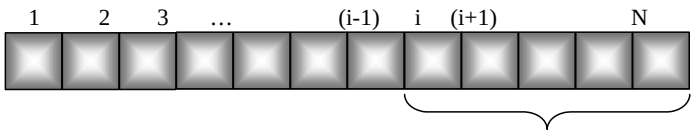
SELECTION_SORT(A,N)

```
[Where A is a vector having N elements]
1.[Loop on I index]
   REPEAT THRU Step 4 FOR I = 1, 2,..., N " 1
2.[Initially assume minimum index is in I]
   Mindex = I
3. [For each pass, get small value]
  REPEAT FOR J = I + 1 to N
```

```
        IF A[MIndex] >A[J] THEN
            Mindex = J
4.[Interchange Elements]
   IF Mindex <> I THEN
      A[I]A[Mindex]
5.[Sorted values will be returned]
   RETURN
```

Explanation: In the above algorithm, for each `I` to `N – 1`, exchange `A[I]` with the minimum element in the array `A[I],…,A[N]`. As the index `I` travels from left to right, the elements to its left are in their final position in the array and will not be touched again, so the array is fully sorted when `I` reaches the right end.

Example 2.16: Sort the elements 16, 19, 4, 1, 20, 2 using selection sort



In the ith pass select the lowest between A[i] and A[N] and swap it with A[I].

Set of elements	**1st Iteration**	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration
16	1	1	1	1	1
19	**19**	2	2	2	2
4	**4**	**4**	4	4	4
1	**16**	**16**	**16**	16	16
20	**20**	**20**	**20**	**20**	19
2	**2**	**19**	**19**	**19**	20

Implementation of Selection Sort to Sort Values of a Vector/Array

Program for selection sort of numbers:

```
/*————————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 10
typedef int VECTOR[MAXCOLS];
void sel_sort(VECTOR a, int n)
{
    int i, j, flag, index;
for(i = 0; i < n – 1; ++i)
{
  Index = i;
  Flag = 0;
for(j = i + 1; j < n; ++j)
    {
```

NOTES

```
if(a[index] > a[j])
       {
   Index = j
   flag = 1;
}
 }
   if(flag)
   {
 int temp;
 temp = a[i];
    a[i] =a[index];
    a[index] = temp;
   }
}
void main()
{
    VECTOR a = {5, 4, 3, 2, 1};
    int i;
    sel_sort(a, 5);
for(i = 0; i < 5; ++i)
       printf("%d ", a[i]);
}
/*————————END OF PROGRAM————————*/
```

OUTPUT: 1 2 3 4 5

Implementation of Selection Sort to Sort Strings of Vector/Array

```
/*————————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
#define MAXROWS 10
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
void sel_sort(STRINGS a, int n)
{
int i, j, flag, index;
for(i = 0; i < n – 1; ++i)
{
  flag = 0, index = i;
       for(j = i + 1; j < n; ++j){
```

NOTES

```
        if(strcmp(a[index], a[j]) > 0)
        {
           Index = j;
           flag = 1
        }
     }
    if(flag)
    {
     STRING temp;
 strcpy(temp, a[i]);
     strcpy(a[i], a[j]);
     strcpy(a[j], temp);
     }

}
}
void main()
{
STRINGS a = {"EE", "BB", "EA", "DD", "AA"};
    int i;
    sel_sort(a, 5);
    for(i = 0; i < 5; ++i)
       printf("%s ", a[i]);
}
/*————————END OF THE PROGRAM————————*/
```

OUTPUT: AA BB DD EA EE

Insertion Sort

Insertion sort refers to a simple sorting algorithm. In it, the sorted array (or list) is built one entry at a time. As compared to more advanced algorithms, such as quick sort, heap sort or merge sort, it is less efficient on large lists. However, insertion sort has many advantages, such as:

- Its implementation is simple.
- It is efficient for every small data sets.
- It is effective for data sets that are already considerably sorted. The time complexity is O($n + d$), where d is the number of inversions.
- Practically it is more effective as compared to most other simple quadratic (i.e. O(n^2)) algorithms, such as selection sort or bubble sort. The average running time of insertion sort is $n^2/4$. Further, in the best case scenario, the running time is linear.

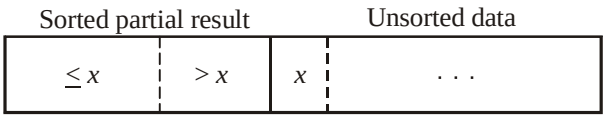
NOTES

- It is stable. In other words, it does not alter the relative order of elements with equal keys.
- It is in-place, i.e. it only requires a constant amount $O(1)$ of additional memory space.
- It is online, i.e. it sorts a list as it receives it.

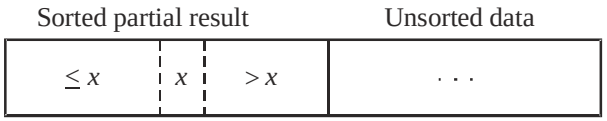
Most people while sorting—ordering a deck of cards, for example—use the insertion sort like method.

Conceptually, each iteration of insertion sort deletes an element from the input data and inserts it into the correct position in the list that is already sorted. The process continues till all input elements are inserted. The element to be removed from the input is chosen arbitrarily. Almost any choice algorithm can be used for this.

Insertion sort is carried out in-place. After *k* iterations, the resultant array has the first *k* entries sorted. In every iteration, the first remaining entry in the input is deleted from the input to be inserted into the result at its correct position, hence extending the resultant array:



becomes



with each element greater than *x* copied to the right.

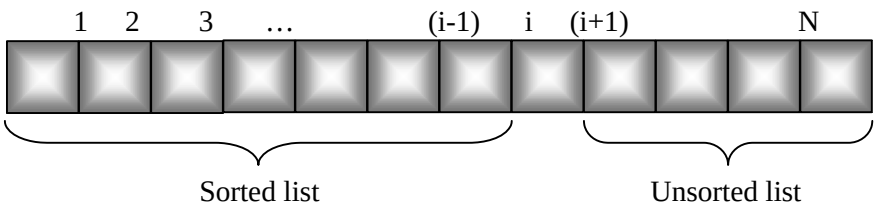
Consider a function called *Insert,* which is designed for inserting a value into a sorted sequence at the beginning of an array. It starts operating at the end of the sequence and shifts each element one place to the right unless an appropriate position becomes available for the new element. This function has a problem. It can overwrite the value that is stored just after the sorted sequence in the array.

For performing an insertion sort, you need to begin at the leftmost element of the array and invoke *Insert* in order to insert each element which is encountered into its correct position. The ordered sequence of inserted elements is stored at the beginning of the array. These elements are stored in the set of indices already examined. Each insertion overwrites a single value, i.e. the value which is being inserted.

Algorithm for insertion sort:

```
Procedure InsSort(A,N).
[Where A is a vector and N denotes number of elements in
the vector.
I,J acts as indices of vector A and Max].
```

```
1. [Initialize I]
    I = 0
2. [Perform sort]
    REPEAT THRU Step 6 until I < N
3. [Initialize Max,J]
    Max = A[I]
      J = I
4. [Backtrack and change]
    REPEAT WHILE J > 0 AND Max < A[J – 1]) /*backtrack */
       A[J] = A[J – 1]
       J = J – 1
5. [Assign Max]
     A[J] = Max
6. [Increment I]
    I = I + 1
7. [Finished]
    RETURN.
```



Example 2.17: Sort the elements 16, 19, 4,1,20,2 using Insertion sort.

Set of elements	2nd Iteration	3rd Iteration	4th Iteration	5th Iteration	6th Iteration
16	**16**	**4**	**1**	**1**	**1**
19	19	**16**	**4**	**4**	**2**
4	**4**	19	**16**	**16**	**4**
1	**1**	**1**	19	**19**	**16**
20	**20**	**20**	**20**	20	**19**
2	**2**	**2**	**2**	**2**	20

From the insertion sort algorithm, sorting is achieved by each iteration as shown in the above diagram. In each row the elements are in sorted order relative to each other above the element within a block; below this element, the elements are not affected.

Analysis of insertion sort: The time complexity of the insertion sort is $O(N^2)$, where N is the number of elements in the array. On an average, the number of interchanges required is $(N^2/4)$ and in worst cases about $(N^2/2)$. The insertion sort is highly efficient if the array is already in almost sorted order.

NOTES

Implementation of Insertion Sort for a Vector Having Numbers as Its Elements

```
#include<stdio.h>
#define MAX 100
typedef VECTOR[MAX];
void InsSort(VECTOR a, int n)
{int i, j, max;
 for(i = 0; i < n; ++i)
 {
     max = a[i];
     j = i;
     while(j > 0 && max < a[j – 1]) /*backtrack */
     {
     a[j] = a[j – 1];
     j = j – 1;
     }
     a[j] = max;
 }
}
void main()
{VECTOR a = {5, 4, 3, 2, 1};
 int i;
 InsSort(a, 5);
 for(i = 0; i < 5; ++i)
     printf("%d ", a[i]);
}
```

Output: 1 2 3 4 5

Implementation of Insertion Sort for a Vector Having Strings as Its Elements

```
#include<stdio.h>
#include<string.h>
#define MAXROWS 10
#define MAXCOLS 20
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
void InsSort(STRINGS A,int N)
{
 int I, J;
 STRING MaxStr;
 for(I = 0; I < N; ++I)
 {
```

```
    strcpy(MaxStr, A[I]);
    J = I;
    while(J > 0 && strcmp(MaxStr, A[J – 1])<0) /*backtrack
*/
    {
      strcpy(A[J], A[J – 1]);
      J = J – 1;
    }
    strcpy(A[J], MaxStr);
 }
}
void main()
{
 STRINGS A = {"EE", "AA", "BB", "DD", "CC"};
 int i;
 InsSort(A, 5);
 for(i = 0; i < 5; ++i)
    printf("%s", A[i]);
}
```

OUTPUT: AA BB CC DD EE

The array which is already sorted is considered the best case input. In the given case, insertion sort has a linear running time, i.e. $O(n)$. During each iteration, the first remaining element of the input would only be compared with the rightmost element of the sorted subsection of the array.

An array sorted in the reverse order is the worst case input. In the given case, insertion sort has a quadratic running time, i.e. $O(n^2)$. Every iteration of the inner loop scans and shifts the entire sorted subsection of the array before the next element is inserted. The average case is also quadratic. That is why the insertion sort is not practical for sorting large arrays. However, for sorting arrays having less than ten elements, insertion sort is one of the fastest algorithms.

2.8 MERGING OF ORDERED LIST

Ordered list is maintained by some predefined order, for example, ascending or descending order or ordered values preserving a specified series. This order can be of numerical or alphabetical order. For this mechanism, implementation of linked list is used. Routines are needed to maintain the order and delete an element from the defined list. A lookup is maintained to continue a routine for extra information from the specified list. A list is considered as numerically ordered for every item `x` in the specified list. Every item after `x` in the list is greater than `x` or equal to `x`. Merging is also known as collating which pushes up a process by which the two given ordered lists are merged or combined into one ordered list. The pseudocode used in merging the ordered list is written in the following way:

NOTES

```
function merge_sort(m)
var list left, right, result
    if length(m) d" 1
        return m
    var middle = length(m) / 2 - 1
    for each x in m up to middle
         add x to left
    for each x in m after middle
         add x to right
    left = merge_sort(left)
    right = merge_sort(right)

    if left.last_item > right.first_item
         result = merge(left, right)
     else
         result = append(left, right)
    return result
```

Some of the sorting algorithms are required to merge the ordered list. One popular sorting algorithm is taken as two-way merging. The two-way merging process is used to merge the two ordered lists. For example,

List_one={a_1,a_2,…a_i…a_n}a_1d"a_2d"…d"a_id"…a_n, a_nd

List_two={b_1,b_2,…b_j…b_m}b_1d" a_2d"…d"b_jd"… b_m,

The merging process combines two given lists into one single list let say L. The following consequences are made by the case of comparison between the keys a_i and b_j both belongs to List_one **and** List_two specified lists respectively:

A_one: If (a_i < b_j) then a_i is dropped into the given list L.

A_two: If (a_i > b_j) then b_j is dropped into the given list L.

A_three: If (a_i = b_j) then a_i and b_j both are merged into the specified list L.

In this case **A_one**, if ai is dropped into the given list L and the very next comparison is possible with b_j proceeding a_{i+1}. In case of A_two, if bj is dropped into the specified list L, the next comparison ai goes with aj+1. At the end of merging process, final list contains (n+m) ordered elements. The series of comparisons from the lists List_one **and** List_two and dropping elements from the smaller elements into the list L proceeding the following consequences :

B_One: List_one uses up List_two. The remaining element in the list List_two are dropped into the given list L occurring L2 and merge process is done.

`B_two: List_two` uses up earlier to that `List_one`. The remaining list into `List_one` is dropped into the specified list `L` occurring `List_one` and merging process is done.

`B_three: List_one` **and** `List_two` are depleted and both merge cases are utilized.

Basic Approach of Merging of Ordered List

The basic approach behind merging of ordered list is an effective method used to sort data and prepare a final list. It contains some concepts behind regulating it properly. If one list starts with small ordered list and other one have more elements then merging process is done to combine small ones into bigger list. Finally all the items generate a final list after merging together. Figure 2.6 shows how two given ordered lists, are merged into a new ordered list.

```
Contents of listObject1:
0 2 4 6 8

Contents of listObject2:
1 3 5 7 9

listObject2 is now empty
Contents of listObject1 after merge:
0 1 2 3 4 5 6 7 8 9
```

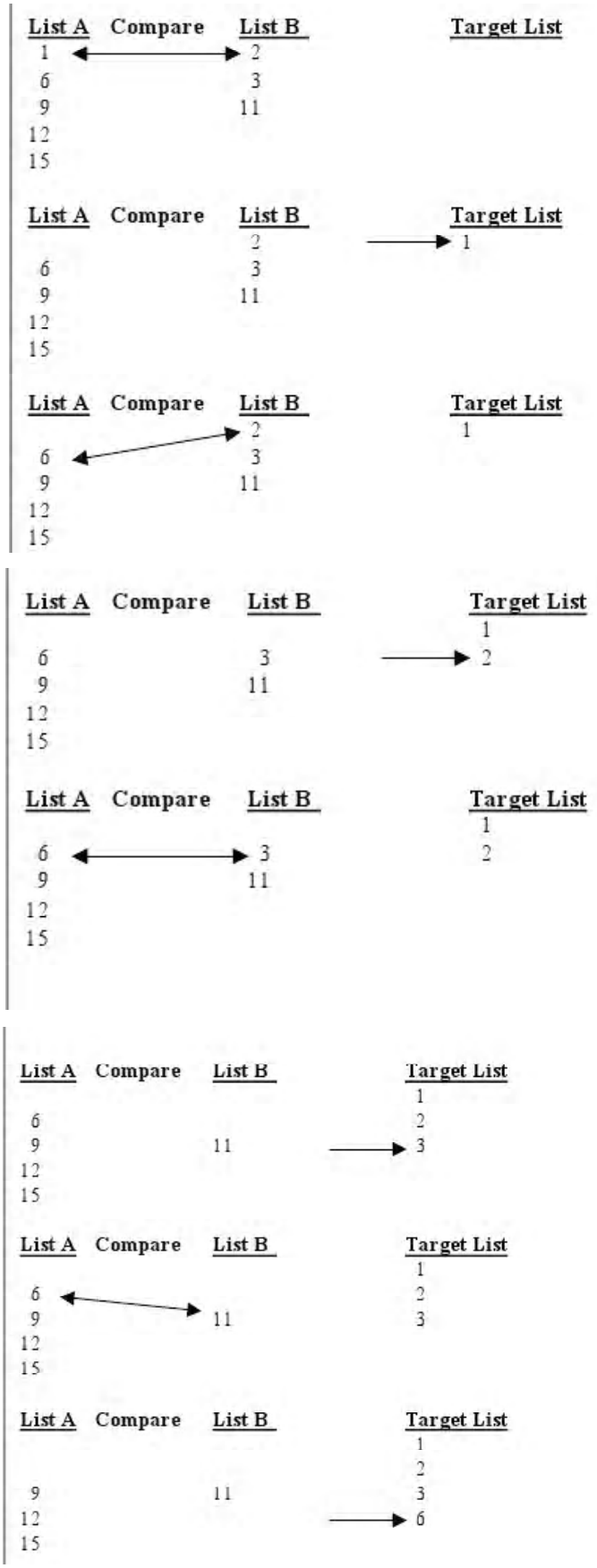
Figure 2.6 *Output of Final List*

In Figure 2.5 `listObject1` contains `{0, 2, 4, 6, 8}` five elements and `listObject2` contains `{1, 3, 5, 7, 9}` elements. Both are merged to produce a final list containing the elements as `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.`

Example 2.18:

For better understanding, an example is taken of two specified lists. `List_one` contains four elements as `List_one ={3,5,7,9}` and `List_two` contains three elements as `List_two= {4,6,7}.` It is assumed that the two elements of given lists are ordered. Therefore, merge list of two ordered list L contains total seven elements that is the sum of four elements from `List_one` and three elements from `List_two.` One element `7` is found common in both lists so it would be dropped out from the final list `L`.

NOTES


List A Compare List B Target List
1 2
6 3
9 11
12
15
List A Compare List B Target List
2 1
6 3
9 11
12
15
List A Compare List B Target List
2 1
6 3
9 11
12
15
List A Compare List B Target List
1
6 3 2
9 11
12
15
List A Compare List B Target List
1
6 3 2
9 11
12
15
List A Compare List B Target List
1
6 2
9 11 3
12
15
List A Compare List B Target List
1
6 2
9 11 3
12
15
List A Compare List B Target List
1
2
9 11 3
12 6
15

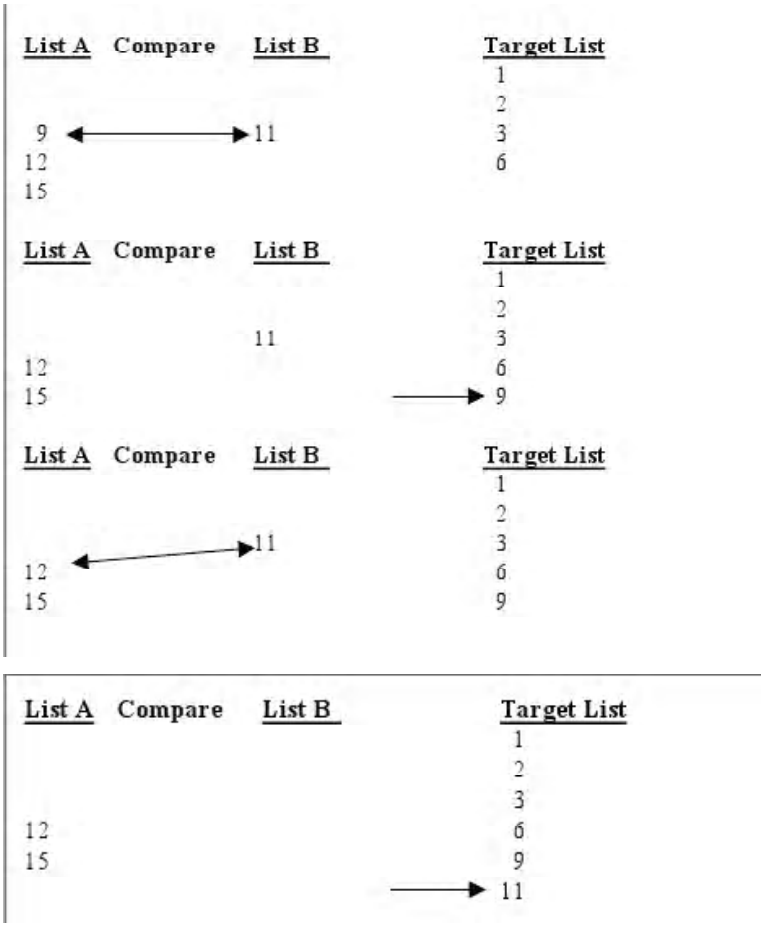


Figure 2.7 *Process Involving in Merging of Ordered List*

In Figure 2.7, the two ordered lists `List A` and `List B` have been taken. The remainder elements of a list can be inserted into the target list that produces the final list that contains values as `1, 2, 3, 6, 9, 11, 12, 15.`

Analysis of time management is done with the process that involves: The total number of elements taken from original list is considered as very large. Each list corresponds with n elements. The two lists are merged therefore it takes second time. A target list is created that consists of all the elements of original lists. The data structure moves with a pace of `2n` that means doubling the time.

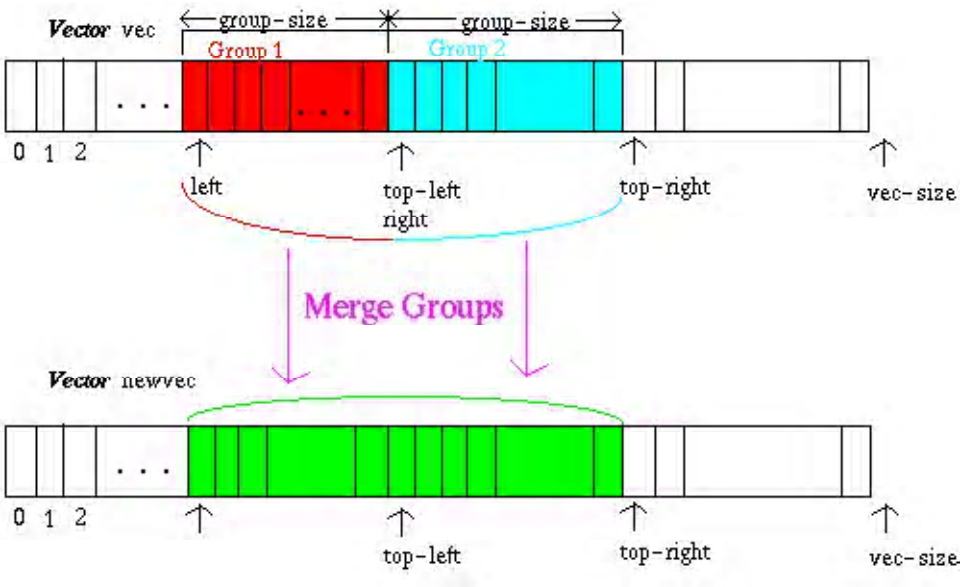


Figure 2.8 *Merging of Ordered List*

In Figure 2.8 `Vector vec` creates two groups `Group1` and `Group 2`. The `Group1` is represented by red colour vector and `Group 2` is represented by blue colour vector. Both ordered lists are merged into green colour vector. In this mechanism, data is moved from one vector to another. Each array index can be compared for both the last group index along with array size. The `Vector newvec` contains the final merged list.

CHECK YOUR PROGRESS

8. Compare the different sorting algorithms with respect to the time and space complexities.
9. Briefly explain the algorithm for selection sort.
10. Write an algorithm to sort the given data in descending order using the bubble sort technique.
11. Formulate a recursive algorithm for binary search with its timing analysis.
12. Write and explain the linear search procedure with a suitable example.
13. Write an algorithm for the matrix addition procedure.
14. In what way is an array different from an ordinary variable?

2.9 SUMMARY

In this unit, you have learned that:

- Algorithms are an important component of the blueprint or plan of a computer program. Knowing when to apply them is crucial to producing software that not only works correctly but also performs efficiently.

- An algorithm is used in computing to efficiently solve a problem with the help of a set order of instructions.
- Flowcharts are used to express algorithms graphically.
- When the program is broken into smaller subproblems, the required algorithm specified for each subproblem is known as stepwise refinement.
- An array is an ordered collection of elements that share the same name.
- Searching refers to an operation of finding the location of an item in a table or a file.

2.10 KEY TERMS

- **Array:** A collection of elements of the same data type.
- **Searching:** Locating a particular element in a data structure.
- **External searching:** When the records are stored in files or on a disk or tape or any secondary storage, then the searching is known as external searching.
- **Internal searching:** When the records to be searched are stored entirely within computer's memory, then it is known as internal searching.
- **Internal sorting**: Sorting of records in a file which is stored in the main memory.
- **External sorting**: Sorting of records in a file which is stored in the secondary memory.
- **Algorithm:** A sequence of computational steps that transform input into output.

2.11 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Purpose, situation, problem, cause, solvable cause, issue and solution.
2. A flowchart refers to a graphical representation of a process which depicts inputs, outputs and units of activity.
3. **Advantages:** (a) clarifies program logic; (b) serves as documentation

   Disadvantages: (a) bulky for the programmer to write; (b) difficult to the understand logic depicted.
4. A flowchart is a graphical representation of the solution to a problem; it is connected with the shape of each box indicating the type of operation being performed. An algorithm is a process for solving a problem and is constructed without any boxes, but a series of steps.
5. Row major ordering and column major ordering

NOTES

6. A prime number is only divisible by 1 or itself.
7. Linear search, binary search and Fibonacci search.
8. Big *O* notation and comparison of sorting algorithms:

Algorithm	Average	Worst Case	Space Usage
Selection sort	$O(n^2/4)$	$O(n^2/4)$	In place
Bubble sort	$O(n^2/4)$	$O(n^2/2)$	In place
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	Extra n entries
Quick sort	$O(n \log_2 n)$	$O(n^2/2)$	Extra $\log_2 n$ entries
Tree / Heap sort	$O(n \log_2 n)$	$O(n \log_2 n)$	In place

9. Beginning with the first element in the array, a search is performed to locate the element which has the smallest key. When this element is found, it is interchanged with the first element in the array. This interchange places the element with the smallest key in the first position of the array. A search for the second element with the smallest key is then carried out by examining the keys of the elements from the second element onwards. The element which was the smallest key is interchanged with the element located in the record position of the array. The process continues until all records have been sorted in the ascending order.
10. Bubble sort technique:

```
/* Algorithm to sort the given data in the descending
order using the bubble sort technique.*/
    BUBBLE_SORT(B,N). Where B is a vector having N elements
    1. [Initialization]
       LAST=N (entire list assumed unsorted at this point)
    2. [Loop on I index]
       Repeat thru step 5 for I = 1 to N – 1 do
    3. [Initialize exchanges counter for this pass]
       EXS = 0
    4. [Compare the unsorted pairs]
       Repeat for J = 1 to LAST – 1 do
          If (B[J] < B[J + 1] ) Then
          B[J] ← → B[J+1]
       EXS = EXS + 1
    5. [Check whether any exchanges occur or?]
       If (EXS == 0) Then
    Return (Sorting finished)
      Else
    LAST = LAST – 1(reduce the size of unsorted list)
    6. [Till maximum number of passes finished]
       Return
```

NOTES

11. Recursive algorithm for binary search:

Function binary search: `R(LOW, HIGH, K, X)`

`K` List of N elements in the ascending order

`X` Element to be found

`LOW` Temporary variable for limit of the search internal

`HIGH` Temporary variable for upper limit of the search internal

`MIDDLE` Temporary variable for middle limit of the search internal

```
LOC   Position Variable
1.[Initiative]
   LOW ← 1
   HIGH← N
2.[Obtain index of midpoint of search]
   MIDDLE [(LOW + HIGH)/2]
3.[Searching]
   if X < K [MIDDLE] then
      LOC ← binary-search-R(LOW, MIDDLE-j, k, l)
else if X > K[MIDDLE] then
      LOC ← binary-search- R(MIDDLE + 1, HIGH, K,X)
else
      LOC ← MIDDLE
      return (loc)
The time complexity for binary search method is
Best Case O(n)
Worst Case O(log2 n)
Average Case O(log2 n)
```

12. **Explanation for linear/sequential search algorithm:**

Linear/sequential search method is the simplest technique for searching, for a given elements, in an unordered list of elements is to scan each entry in the list in a sequential manner until the desired element is found. The algorithm for linear/sequential search method is as follows:

Function linear search `(K, N, X)`

`K` Unordered list with N elements

`X` Element to be found

`I` Index

```
1.[Initialize search]
   I ← 1
2.[Search the list]
   While (K[I] ≠ X & ≤ IN]
      I ← I + 1
   repeat step2
```

```
3.[Successful Search?]
      If J = N + 1 then
   Print ('Element not found in the list')
   return (0)
      else
   Print ('Element found at position')
   return(I)
4. Stop
The time complexities for linear search method are
Worst case n + 1 Comparisons
Average case (n + 1)/2 Comparisons
    Time complexity for linear search method is O[n].
```

13. **Algorithm:**

Step 1: start

Step 2: initialize the variables a, b, c, i, j;

Step 3: enter the first matrix

Step 4: for i = 0, i < 3, j++ is the condition satisfies go to next step else go to

4.1: for j=0,j<3,j++ is the condition satisfies go to the next step

4.2: read a value

Step 5: enter the second matrix

Step 6: for i = 0, i < 3, i++ go to next step

else go to step 7

6.1:for j = 0, j < 3, j++ go to 6.2

6.2: read b

Step 7: print addition of matrix

Step 8: for(i = 0, i < 3, i++) go to 8.1 else go to step 9

8.1: for j = 0, j < 3, j++ go to 8.2

8.2: c[i][j] = a[i][j] + b[i][j]

8.3: print c

Step 9: stop

14. Differences between an array and an ordinary variable:

- An array is a set of variables of the same data type referred to by a common name. A variable is a data name that stores a specific data type value which may alter during the execution of the program.
- Each member in the group is referred to by its position in the group (array) using self-scripts. A variable is referred to by its name.

- Array elements are always stored in contiguous memory locations. Variables can be stored in different memory locations.
- The type of an array and its dimension have to be specified before it can be used so that the complier knows the type and size of the array. Variable type must be declared before using and there is no dimension for a variable.

2.12 QUESTIONS AND EXERCISES

Short-Answer Questions

1. What are the key points to keep in mind while writing a code?
2. List the steps in developing a program.
3. What are the four qualities of a good program?
4. List five important properties of an algorithm.
5. What are the types of algorithms?

Long-Answer Questions

1. Write a detailed note on the steps required to develop a program.
2. What are the characteristics on an algorithm? Explain in detail.
3. Write an explanatory note on debugging.
4. What are the types of algorithms? Explain in detail.
5. Write an explanatory note on Mergesort.

2.13 FURTHER READING

Friedman, Daniel P. *Essentials of Programming Languages*. Boston, MA: MIT Press.

Manber, Udi. *An Introduction to Algorithms: A Creative Approach*. New York: Addison-Wesley.

Farrell, Joyce. *Computer Programming Logic Using Flowcharts*. New York: Boyd & Fraser.

UNIT 3 PROGRAMMING

NOTES

Structure

3.0 INTRODUCTION

Programs are the 'thought processes' of computers. They are written in various languages as a set of instructions to be interpreted and executed by the computer, to enable it to perform the required task. It is also called the software installed in the computer.

NOTES

A program is prepared by first defining the task and then expressing it in an appropriate programming language. Most applications programmers use one of the high-level languages (such as BASIC or C++) or fourth-generation languages that more closely resemble human communication. In this unit, you will study various aspects of programming.

3.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand why C program is the most suitable and commonly used language for computer software
- Distinguish between the different components that go into making a program
- Elaborate on the process of writing a C program
- Analyse the different parts and functions of the program
- Discuss the usage of files for storing data, popularly known as data files
- Explain the concept of the programming paradigm
- Examine the process of debugging and testing of a program to verify and validate it

3.2 INTRODUCTION TO PROGRAMMING

Computer program or software is nothing but a set of instructions to a computer to carry out specified tasks. The instructions are written in a language understandable by the computer. Programs can be developed and executed only in a computer system. Programming environment consists of the following:

- Computer hardware, which has a physical appearance. It consists of central processing unit (CPU), keyboard, main memory, secondary storage devices such as floppy disc drive, hard (Winchester) disc drive, compact disc drive, video monitor, printer, modem, sound card, input/output port, etc.
- Operating system, which is also the software required for the operation of the computer system. It is known as system software. Popular operating systems include Linux, UNIX and Windows family of operating systems.
- Integrated (program) development environment (IDE). Examples include GNU gcc compiler, Borland C++ compiler, Turbo C++ integrated development environment, Microsoft Visual Studio, etc.

At this stage, it is important to understand what a computer programming language is and what the various types of computer languages are.

3.3 INTRODUCTION TO C AS REFERENCE

NOTES

There is wide scope for C language since it is portable and platform independent. Wide acceptance of C language as an efficient language is due to the fact that it is used for implementation of compilers, interpreters and libraries of other programming languages.

In a computer system, it is a program that runs and performs various tasks. A program is written in a language that is understood by the computer. For system programming, 'C' language has been found to be very suitable. Such system programming includes implementation of operating systems and embedded systems. Many devices for industrial or domestic use are now digital machines, be they industrial fans, coolers or home appliances such as washing machines and microwave ovens. These gadgets have microprocessors embedded in them to enable users to program them as per need.

The C language's unique features of code portability as well as efficiency and ability to access specific hardware addresses has made its scope wide for many control devices. C also has a low runtime demand on system resources.

This language serves the purpose of an intermediate in implementing other languages for portability. When C language is used as an intermediate language, there is no need for code generators that are machine specific. Compilers such as BitC, Glasgow and Gambit make use of C in that ways.

C was not developed as a target language for compilers, it was originally developed as a programming language, yet compilers can be written in C. Thus C-based intermediate languages developed. The intermediate language C— is an example.

Many tools have been developed to add more features to C-language and enhance its applicability. Lint is such a tool doing automated source code checking and auditing. Lint is used to detect codes that are questionable on first writing of a program. After using Lint, a C-compiler is used to compile it. There is a set of guidelines in a compiler named MISRA C that is used to avoid questionable codes for embedded systems.

C-based compilers are found along with libraries and mechanisms at operating system level that perform checking of bounds in an array, overflow in buffers, and serialization. It also performs automatic garbage collection. These do not form standard operations in C.

The versatility of C has influenced many languages developed later. These languages are Java, C#, Perl, PHP, JavaScript, LPC, and C shell of UNIX. The syntax of C language has influenced all these languages. The most pervasive influence has been syntactical.

C has been extended to develop languages that are object oriented. Two well known extensions of C are C^{++} and Objective-C. C^{++} and Objective-C were

NOTES

source-to-source compilers. Source codes of these two were first transformed to C before compiling using C compiler.

Bjarne Stroustrup developed C++ by adding object-oriented functionality, but he adopted a C-like syntax. C++ provides support for almost all features of C and hence is considered as superset of C. Objective-C is strict superset of C, permitting object-oriented programming deriving its syntax from C as well as Smalltalk. Syntax for function declarations, a function calls preprocessing, expressions, has been borrowed from C, but OO features were borrowed from Smalltalk.

By retaining few syntax and general style of C, a language known as Limbo was developed. Major innovations such as CSP base concurrency, Garbage collection and other were added to it. Python is a language rooted in C, although its syntax now has little similarity with C. Python is open source. Hence, users of C may do extension of Python using C and can also embed it in C codes. Such a close relationship acted as key factors for the success of Python as a scripting language of general use.

Perl is also a programming language having roots in C, with a similar syntax. This works well with extensions of C programs.

Although the concepts of C are easy to learn, it is dwindling in popularity as many other languages have appeared on the horizon, producing effective software with minimum effort in comparison to C. C is very basic in its essence and still powerful. Nowadays, its scope lies mostly in academic studies, but it is a very flexible language in which many programming decisions are left to programmers.

Features of 'C' Languages

C is portable: C-codes are platform independent. Programs written in C can be run on any machine as it has standard library functions that are not machine specific. It has built-in preprocessor for isolating system-dependent codes.

C is terse: There are some operators in C that can be used to do programming at bit level. Use of increment operator ++ permits writing code similar to those for machine level programming that makes programs runs faster. Use of indirection operator and address arithmetic in combination makes code very concise in C. In other languages, something may be accomplished by writing many statements. Many programmers find this both elegant and efficient.

C is modular: It provides modularity by supporting one style for routine. In external function argurment, call-by-value is used. In C, functions are not nested. A limited form of privacy is provided by using the storage class static within files. These features, along with tools provided by the operating system, provides support for user-defined functions as well as modular programming.

C is efficient on most machines: As certain constructs in the language are explicitly machine-dependent, C finds implementation is natural with respect to machine's architecture. As a machine can do what comes naturally, compiled C

code can be very efficient. Of course, the programmer must be aware of any code that is machine-dependent.

C is appealing: The appeal for C lies in unfettered nature of its operators which are very powerful and provides modularity with concise codes. This also makes it interactive and scope for experimentation.

NOTES

What do I require to write and run a program in 'C'?

The precise steps that have to be followed to create a file containing C code and to compile and execute it depend on three things: **the operating system**, **the text editor** and the **compiler**. We first describe in some detail how it is done in a UNIX environment. Then we discuss how it is done in an MS-DOS environment.

Steps to be followed in writing and running a C program

1. Using a vi editor, create a text file, say *pgm.c*, that contains a C program. The name of the file must end with .c, indicating that the file contains C source code. For example, to use the *vi* editor on a UNIX system, we would give the command

   $ *Vi pgm. c*

   To use an editor, the programmer must know the appropriate commands for inserting and modifying text.

2. Compile the program. This can be done with the command.

   $ *cc pgm. c*

   The cc command invokes in turn the preprocessor, the compiler, and the loader. The preprocessor modifies a copy of the source code according to the preprocessing directives and produces what is called a *translation unit.* The compiler translates the translation unit into object code. If there are errors, then the programmer must start again at step 1 with the editing of the source file. Such errors are known as *syntax error.* If there are no errors, then the loader uses the object code produced by the compiler, along with object code obtained from various libraries provided by the system to create the executable file *a.out.* The program is now ready to be executed.

3. Execute the program. It can be done by using the following command:

   a.out

   Typically, the program will complete execution and a system prompt will reappear on the screen. Any errors that occur during execution are called *run-time errors.* If for some reason the program needs to be changed, the programmer must start again at step 1.

If we compile a different program, then the file *a.out* will be overwritten and its previous contents lost. If the contents of the executable file *a.out* are to be saved, then the file must be moved or renamed. Suppose that we give the following command:

cc sea.c

NOTES

This causes executable code to be written automatically into *a.out.* To save this file, we can give the following command:

mv a.out sea

This causes *a.out* to be moved to *sea*. Now following command can be used to execute the program:

sea

In UNIX, executable file is given the same name dropping the c suffix. This *-o* option may be used for directing the output. For example, command

cc -o sea sea.c

Writes executable output directly into *sea,* and leaves contents of *a.out* intact.

MS-DOS Environment

Here, a different text editor is mostly used. Turbo C uses command line environment as well as an integrated environment. Integrated environment has text editor as well as compiler. In both MS-DOS and UNIX, the command that invokes the C compiler depends on which C compiler is being used. In MS-DOS, executable output of a C compiler bears the same name, but extension .exe is given instead of .c.

Now, start the Turbo C editor and type the program code and press **Alt+F9** to compile the program, **Ctrl+F9** key to execute the program and **Alt+F5** to see the output of the program. When we press **Ctrl+F9** actually at the backend, it invokes the **.EXE** file. That is when we create a source code file having extension **.C**. The '**C**' compiler creates **.obj**, .**BAK** and **.EXE** files. The **.BAK** files contains the same as source code. The **.EXE** is self-executable file. It can be invoked at the DOS prompt. It does not require the presence of any **.C** or **.BAK** or **.obj** files.

Objectives and Application Areas of 'C'

The programming language C was developed with an objective to:

(1) Create a computing environment that is comfortable,

(2) Compile programs using simple compiler,

(3) Provide direct access to memory at low-level,

(4) Generate few machine language instructions for its core language elements,

(5) Run without extensive run-time support,

(6) Encourage machine independent programming.

These features, once incorporated, produce code that can be directly used for systems programming applications and replace assembly language that was adopted traditionally for this purpose. The objective of portability and platform independence was achieved as C was good for both low-level as well as high level, machine independent programming.

NOTES

Thus, a C program can be compiled on different machines with different operating systems little by making change in source code. This language or no run, on different platforms. Its application ranges are wide. At lower end it is embedded microcontrollers and at higher end it is supercomputers.

The maximum application of C was found in UNIX. C was the language for UNIX at the time of its publication. But C can create any type of program, including compilers and various application programs to control different operations in a sophisticated equipment.

C can also be used in fast scientific computations or for creating games. Many business tools for computations, networking and security are made in C. In all, C is an ideal language to master the art of programming.

Thus, practical applications of C are in:

1. System level programming
2. Creation of a new languages
3. Creating games
4. Developing applications such as library systems, transport systems and pay-roll generation systems.
5. Making embedded devices in designing chips and automatic control system in industrial operations.

C programs can be developed using a procedural programming paradigm and also for structured programming in which parameters are passed either by value or by reference using pointers. It also handles heterogeneous aggregate data types as *struct*. This structure with *struct* keyword allows combination of related data elements into one unit for manipulating data.

It has a small set (around 30) of reserved keywords. It has provisions for accessing computer memory at low-level via machine addresses and typed pointers. Pointers are used for array indexing, which is a secondary notion based on pointer arithmetic.

For writing macros, there is standardized preprocessor that is also used for inclusion of source code file and for conditional compilation. C is a simple, small core language, having functions for mathematical calculations and file handling by using library routines.

3.4 REPRESENTATION OF INTEGERS

NOTES

Integer Constants

The following are the types of integers:

```
int
unsigned int
long
unsigned long
```

Variations in Integer Types

Sign bit can also be used for holding value. In such cases, the variable will be called `unsigned int`. The maximum value of an `unsigned int` will be equal to 65535 because you are using the Most Significant Bit (MSB) also for storing the value. The minimum value will obviously be 0.

A long integer is represented as `long int` or simply `long`. The maximum and minimum values of long are as follows:

LONG	MAX	+	2147483647
LONG	MIN	–	2147483647

Unless otherwise specified, integers or long integers will be signed, i.e., the first bit will be reserved for the sign. The `long int` obviously uses 4 bytes or 32 bits.

The magnitudes of `long` can also be doubled by using an unsigned long integer denoted as `unsigned long`.

However, integers are not suitable for very low values and very large values. This can be overcome by floating point or real numbers.

To specify an `unsigned integer` an integer constant is suffexed by letter `u` or `U`. Similarly, if the integer is suffixed with `l` or `L`, it signifies a `long integer`. If you specify `unsigned long integer` you suffix the constant with `ul` or `UL`.

The following are the examples of valid and invalid integers:

Valid integers

```
+345      /* integer */
 345      /* integer */
–345      /* integer */
729u      /* unsigned integer */
729U      /* unsigned integer */
–112345L  /* Long integer */
112345UL  /* Unsigned Long integer */
+1123451  /* Long integer */
```

```
1123451   /* Long integer - if no sign precedes, it is a
positive number */
```

Invalid integers

```
345.0        /* decimal point not allowed */
112, 345L    /* no comma allowed */
112 345UL    /* = blank not allowed */
112890345L   /* exceeds the maximum */
+112 345UL   /* unsigned cannot have + */
(3451        /* ( not allowed */
–345s        /* illegal characters */
```

NOTES

You have so far considered only decimal numbers. C, however, entertains other types of numbers as well. The octal numbers will be preceded by 0 (zero).

The following are the examples of valid and invalid octal numbers:

Valid octal number

```
0346
0547
0120
```

Invalid octal number

```
0394         /* 8 or 9 are not allowed in an octal number
*/
0 x 345      /* prefix has to be zero only */
```

The C language also supports hexadecimal numbers. Here, since the base is 16, we use alphabets also in the numbers as given in Table 3.1.

Table 3.1 *Use of Alphabets for Numbers*

a	or	A	for	10
b	or	B	for	11
c	or	C	for	12
d	or	D	for	13
e	or	E	for	14
f	or	F	for	15

Additionally, hexadecimal numbers will be preceded by `0X` or `ox`, i.e., zero followed by `x`.

The following are the examples of valid and invalid hexadecimal numbers:

Valid hexadecimal numbers

```
0x345
0xA132
0x100
0x20B
```

NOTES

Invalid hexadecimal numbers

```
0x, 123   /* no comma */
0x        /* cannot be empty */
0A00      /* x is missing */
```

Character Set

The C language supports and implements the American Standard Code for Information Interchange (ASCII) for representing characters. The ASCII uses 7 bits for representing each character or digit. The characters are coded from 0000000 (decimal 0) to 1111111 (decimal 127). Therefore, the ASCII consists of a code for 128 characters in all. The ASCII values (decimal equivalent of the 7 bits) of some alphabets and digits are given in Table 3.2.

Table 3.2 *ASCII Values of Selected Alphabets*

ASCII Value	Character or Digit
48	0
49	1
57	9
65	A
66	B
67	C
89	Y
90	Z
97	a
98	b
121	y
122	z

The digits and alphabets are organized sequentially and hence, it is easy to get the ASCII value; for instance, the ASCII value of D is 68, E is 69, 8 is 56, *x* is 120 and so on.

3.5 DATA TYPES

Data is used in a program to get information. In a program used to find out the greater of two numbers, the numbers are data and the output, which says which number is greater, is information. C is a versatile language and handles many different types of data in an elegant manner.

Bit stands for binary digit, i.e., 0 or 1. Each byte contains 8 bits, i.e., 8 consecutive bits of '0' or '1'. Data is handled in a computer generally in terms of bytes and therefore, will be in the form of multiples of 8 bits. Each ASCII character is represented by one byte.

Fundamental Data Types

NOTES

An item that holds data is also called an object. An object has a name or identifier associated with it. Each object can hold a specific type of data. There are five basic data types in C, as follows:

- Character
- Integer
- Real numbers
- Void (comprising an empty set of values)
- Enum **(**which will be introduced later)

You have to understand how a computer works. Assume multiplication of two numbers `a` and `b`. First of all, the two numbers have to be stored in the memory. Then the required calculation has to be performed. The result has also to be stored in the memory. Each number is of a specific data type; for instance, all three of them can be declared to be integers. Each data type occupies a specific size in the memory. What does one mean by size? It is the amount of storage space required; each bit needs one storage space. One byte needs eight storage spaces. If a number is of type integer declared as `int`, it is stored in 2 bytes. The number depending on its type gets stored in different forms. If a number is of `float` type, it takes 4 bytes to store it. All characters can be represented according to the ASCII table and hence, 1 byte, i.e., 8 bits are good enough to store a character, which is represented as `char`.

These sizes may vary from computer to computer. The header files `<limits.h>` and `<float.h>` contain information about the sizes of the data types.

Real numbers can be expressed with single precision or double precision. Double precision means that real numbers can be expressed more precisely. Double precision also means more digits in mantissa. The type `'float'` means single precision and `'double'` means a double precision real number. Table 3.3 indicates the size of various data types.

Table 3.3 *Size of Data Types*

Data Type	Size
`char`	1 byte
`int`	2 bytes
`float`	4 bytes
`double`	8 bytes

Maximum and Minimum Magnitudes

The maximum and minimum values of data types are not limitless. For example, `<limits.h>` specifies the minimum and maximum magnitudes for integers and characters. Since `char` is stored in a byte, it is as same short integer. `char` can

NOTES

be stored as an unsigned character, which means all the 8 bits are used for storage and maximum value in an 8-bit number is 255 when all bits are 1. In case of a `signed char`, first bit is reserved for storing the sign. The sign bit will be positive if first bit is 0 and negative if first bit is 1. Integer values of `signed` and `unsigned chars` are as follows:

CHAR-BIT		8	bits in a `char`
SCHAR	MAX +	127	maximum value of `signed char`
SCHAR	MIN –	127	minimum value of `signed char`
UCHAR	MAX	255	maximum value of `unsigned char`

Now, let us look at the maximum and minimum magnitudes for the integer data type. They are:

INT	MAX +	32767	maximum value of `int`
INT	MIN –	32767	minimum value of `int`

Integer means signed integer. The data type integer occupies 2 bytes or 16 bits. The most significant bit is reserved for sign. It will be '0' when number is positive and '1' when number is negative. Therefore, you can easily calculate how the limits for the integer data types have been arrived at.

The standard has also provided for another type of integer called `short int` with the same maximum and minimum values.

3.6 CONSTANTS/LITERALS

Constants are fixed values which remain unchanged during program execution . These may be numbers or character strings.

e.g. 100, 3.14, 'Hello World', etc.

Constants are data values that cannot be changed during the execution of a program.

A C constant is generally the written version of a number, e.g., 1, 0, 5.73, 12.5e9. Constants can be written in octal or hexadecimal or as long integers.

- Octal Constants in octal are written using a leading zero—016.
- Hexadecimal constants are written with a leading 0x—0x1AE.
- Long constants are written using L in the last —990L.
- Character constants are written as character using single quotes; 'a', 'b', 'c'. There are characters that cannot be represented like that. Hence, you have to use a 2-character sequence.

Types of C Constants

NOTES

- **Integer Constants:** The following are the rules for constructing constants:
  - Must have at least one digit
  - Should not contain a decimal point
  - Could be either negative or positive
  - If no sign precedes, assume it to be positive
  - No commas or blank spaces
- **Real Constants:** Real constants can be expressed in the following two forms:

Fractional form

- Must have at least 1 digit
- Contains a decimal point
- Could be either negative or positive
- If no sign precedes, assume to be positive
- No commas or blank spaces

Exponential form

It is represented in two parts: mantissa and exponent. The following are the rules:

- There must be separation between mantissa part and exponent part by a letter e.
- Mantissa part may contain positive or negative sign.
- Mantissa part is positive by default.
- Exponent should be of at least 1 digit, positive or negative integer.
- Default is positive.

- **Character Constant:** The following are the rules:
  - It is a single alphabet or a single digit or a single special symbol enclosed in a single inverted commas (quotes).
  - Both inverted commas should point to left.

    Example: 'A' valid

    'A' not valid

  Maximum length is one character.

Declaration of a Constant in 'C'

Syntax 1: *const <variable-name> [= <value>];*

Where the **const** modifier permits assignment of an initial value to a variable that program cannot change later.

e.g.: const week = 7; any assignments to **week** will show compiler error.

Syntax 2: *const <type> *<var-name>;*

NOTES

Note: A constant variable may undergo indirect modification by a pointer as follows:

*(int *)& week = 7; When the **const** modifier is used with a pointer in a parameter list of function, it cannot modify variable that the pointer points to.

e.g.: int printf (**const** char *format, ...);

Here the printf function cannot modify the string format.

'C' has the various types of constants, as shown in the Table 3.4:

Table 3.4 *Various Types of Constants*

Constants	Description	Examples
Single character	Character represented in single quotes.	'a'
Multibyte character	More than one character represented in single quotes	'xy'
Wide character	Character should follow with L and usually its size will be 2 bytes. The type of wide character const is wchar_t defined in <stddef.h>	L'X'
Backslash character	All escape sequence characters	'\n'
[signed] int	Numeric values with int range –32768 to 32767	100
[signed] short [int]	Numeric values with in the range –32768 to 32767	–100
[signed] long [int]	Numeric values with in the range -2,147,483,648 to 2,147,483,647 having suffix L	100000L
unsigned [int]	Numeric values with in the range 0 to 65535 having suffix U	60000U
unsigned long [int]	Numeric values with the in the range 0 to 4,294,967,295 having suffix UL or LU	10000UL
Float	Floating point value with in the range 3.4 * (10**–38) to 3.4 * (10**+38) having suffix f with six digit precision.	3.2f, 10.2e–2f
Double	Floating point value with in the range 1.7 * (10**–308) to 1.7 * (10**+308)	12.2, 10.2e–2
Long double	Floating point value with in the range 3.4 * (10**–4932) to 1.1 * (10**+4932) having suffix L	10.2L
Octal	Number system having base 8.uses digits 0–7.0 should be followed while representing octal constants	010
Hexadecimal	Number system having base 16.uses digits 0–9 A-F(or)a-f. 0x should be followed while representing hexadecimal constants	0x12
String	Sequence of characters enclosed in double quotes	"World"

***Note*:** In Table 3.4 L, U, and F are not case sensitive.

3.7 VARIABLES

NOTES

A **variable** refers to a symbol showing a value that is subject to variations. It is opposite to a **constant,** which has a non-varying value. In other words, a constant is fixed as per context of use.

Variables may be non-numeric. For example, the variable *lastname* in a spread sheet, may show last name of a customer. Understanding of constants and variables is fundamental in the field of mathematics, science, engineering and computer programming.

Sometimes, variations in values of variables is literal; the values actually change. For example, in procedural programming using imperative statements, we set the variables and generally modify these using explicit assignment.

Following C program show this:

```
int r = 5;
while (r*r < 100)
{
 r = r + 1;
}
```

Dependence on other variables

More abstract view is taken of variables in science, mathematics, and programming that use non-imperative form. Logical programming or pure functional programming also take the same view of variables. In such a view variation does not mean change; rather it shows dependence on value of other variables in a given expression or dependence of values of expression.

For example, in physics there is an expression $E = mc^2$, E and m are variables whereas c is a constant which is velocity of light in vacuum. The equation describes the interdependence of the values of E and m while c is fixed.

Relationship between pressure (p), volume (V) and temperature (T) is shown by the ideal gas law as $PV = nRT$, where n is a constant. A chemist may manipulate some of these variables to fit others, despite being fully aware of the fact that they cannot defeat the law.

In mathematics, a polynomial of second-order is normally denoted as $ax^2 + bx + c$, where a, b and c are constant terms, whereas x is termed a variable. Study is conducted to find changes in values of function for given values of these constants when value of x changes.

As far as computer programming is concerned, a variable refers to a keyword or phrase (identifier) which has link to a stored value in the memory of the system or an expression to be evaluated. For example, a variable may be named 'total_count' and may store a number.

NOTES

Imperative programming, accesses or changes values as may be needed. But is programming having referential transparancy variables remain bound to expressions. This is found in logical and pure functional programming. Variables in such programming have single value throughout their scope.

Programming language, C++ and few others, strictly adhere to type of data. Each variable has certain type. It stores data conforming to the type. In some languages, variables are treated as data type, most convenient at that time. JavaScript is one such languages. Both approaches have their advantages. The former permits faster programming and execution, whereas latter helps in preventing programming mistakes, leading to saving of time, although it initially consumes more time.

Naming Conventions

Some programming languages use specific characters as prefix or suffix to identifiers for indicating the type of variable. For example:

- In BASIC, the suffix used for dollor sign, $ over a variable name shows string;
- In Perl, characters $, @, %, and &, give indication of variables of type scalar, array, hash, and subroutine respectively.
- In spreadsheets, use of $ is made for variables referring to cells (e.g. B3). These are used for values associated functions, source code or named rangers.

Variables in Source Code

In source code of a computer, a variable is one way you can bound to a memory location. Corresponding value can be stored as a data object in that memory location and the object can be manipulated later using name of the variable.

Variables in Spreadsheets

In a cell of a spreadsheet, there may be a formula that references other cells. A cell reference of this type is also a variable, and value of this cell is the value of the referenced cell.

Scope and Extent

The scope of a variable describes where the variable can be used in the body of a program text. The extent (or lifetime) describes the time, a variable takes a value while executing a program. Scope of a variable shows the property of variable's name but extent shows the property of variable itself.

The scope of a variable name affects its extent.

Scope is lexical feature scope of a variable is the part of a program code having meaning for variable's name and for which the variable is 'visible'. The lifetime of a variable begins with the entry into that scope, while the lifetime of a

variable ends with the exit from that scope. For example, a variable having 'lexical scope' has meaning only inside a fixed block containing statements or subroutine. It may be noted that a 'global variable' is a variable having scope everywhere in that program.

Extent is a run-time aspect associated with a variable. Each binding for a variable to some value may be having its own extent during run-time. The extent of binding points to the part of execution time of the program. During execution time, the variable goes on referring to the same memory location or value. Just as in a closure, a program under execution may make an entry to and exit from a given extent several times.

It is erroneous if attempt is made to use value of a variable at a time when it is not within the extent. If you do this unpredictable results may be there. However, you can assign a new value to such a variable, giving it a new extent. As a contrast, it becomes allowable for binding in a variable to get extended beyond its normal scope, as in Lisp closures and static variables of C. When execution returns into the scope of variable, it may again be used.

To achieve space efficiency, a memory space may be allocated only when variable is to be used and freed if it is no longer required. Note that a variable has only when it is in scope. Wastage of memory space should be avoided. To do this compilers warn programmers if a variable is declared but remain unused.

Scope of variables should be made narrow to the extent that is feasible. It would ensure safeguard against accidental interaction between various parts of program that may modify. To make variables narrow in scope, common techniques used are: Use of different name space for different sections of the program or making individual variables private. To make variables private, scope of variable is made either lexical or dynamic.

There are programming languages employing null or nil as a reserved value for indicating a variable invalid or uninitialized.

Typed and Untyped Variables

In languages that are statically typed, a variable too has a type. Java and NIL are such languages. It means that you can store in it only values a given class. It may be noted here that variables of primitive type carry values of primitive type. Variables of class type carry null references or references to objects whose type belong to that class type or its subclasses. A variable that is of interface type carries null reference or a reference to an object implementing the interface. As far as an array type variable is concerned, it holds a reference to an array or to a null.

In dynamically typed languages, values carry type and do not carry variables. Python is an example of such a language. Both the situations are present simultaneously in common Lisp. A type is declared for a variable, and is presented at the time of compiling the program. In case it is not declared, it is supposed to be

NOTES

T, the universal supertype. There are types of values also. You can check and query these types at run-time. Refer to type system.

An important feature of typing of variables is that it allows the resolution of polymorphisms at compile-time. But it is not the same as polymorphism used for function calls in object-oriented programming, which are known as *virtual functions* in C++. This resolves the call on value type and not the supertypes for the variable.

Variables store simple data. However, there are few programming languages that store values of other data types also. In these languages, functions can also be parametric and polymorphic. Such functions can act like variables for representing data of multiple types. A function named length, for instance, determines length of a list. If a length function like this includes a type variable in its type signature, it becomes parametric polymorphic, as amount of elements is independent of the types of elements in the list.

Parameters

The *formal parameters* of functions are also called variables. For example, we take the following code segment of Python,

```
def addtwo(x):
 return x + 2
addtwo(5) # yields 7
```

here, variable x is *parameter* as it is assigned a value when the function is called. The integer 5 is an *argument*, from which x gets its value. In most of the languages, the scope of function parameters is local. This specific variable named x can only be referred to within the addtwo function (although other functions may also have variables called x).

Memory Allocation

There are wide variations in the specifics of variable allocation and the representation of their values, both in terms of programming languages as well as implementations of a given language. Space for local variables are assigned in implementations of many programming languages. The extent of these variables lasts for a single function call on the *call stack*. The memory of these variables is recovered automatically on return of the function. In *name binding*, name of a variable is bound to the address of few specific block of memory bytes, and that block is manipulated by operations on the variable. In case of variables with large or unknown size values at the time of compilation of the code, referencing is more commonly used. Such variables access location of the value rather than storing the value itself, and allocation of value is done from a pool of memory called *heap*.

A value is an abstraction, an idea. In implementation, some *data object* represents a value. The data object is stored in computer memory at some location. The program, saves memory for every data object and, as memory is finite, ensure

availability of this memory for reuse when object is no longer required represent value of some variables.

Objects distributed from the heap needs recovery, particularly when these objects are no longer required. In a language that does garbage collection, such as C#, Java and Lisp, run-time environment does automatic recovery of objects when they are longer referred to by extent variables. In languages that do not perform garbage collection, such as C, the program should clearly assign memory and subsequently, make it free for recovering memory. If it is not done, it would lead to memory leaks. This exhausts the available memory.

NOTES

If a variable indicates a data structure that is dynamically created, some parts of it can only be accessed indirectly through the variable. In situations like that, garbage collectors should necessarily tackle a condition in which only a part of the memory accessible from the variable be recovered. This also applies to analogous programs that do not have garbage collectors.

Variable Interpolation

Variable interpolation is also known as variable substitution or variable expansion and it refers to a procedure that evaluates an expression or string literal having at least one variable, giving a result by replacing variables corresponding values in memory. It is a functional example of concatenation.

Perl, PHP, Ruby and most UNIX shells, etc. are the examples of the languages which support variable interpolation. Variable interpolation takes place in these languages only when the string literal is double-quoted. If the string literal is single-quoted, there would be no variable interpolation. The variables are recognized since they begin with a sigil (typically '$') in the languages supporting variable interpolation.

For instance, the following Perl code:

```
$name = "Nancy";
print "$name said Hello World to the crowd of people.";
produces the output:
Nancy said Hello World to the crowd of people.
Ruby uses the '#' symbol for interpolation, and lets
you interpolate any expression rather than just variables.
```

Rules for Constructing a Variable Name or an Identifier

- First character of an identifier must be an alphabet or an underscore and remaining characters may be a combination of letters or digits or underscores.
- Keywords must not be used as identifiers (variable names).
- Special characters are not allowed except underscore.
- Recommended maximum length of an identifier is eight characters (some compilers allow more than eight characters length)

- 'C' is a case sensitive language. So, upper- and lower-case letters are significant in identifiers.

Declaration of a Variable

NOTES

Any variable should be declared in a program before using it in the program. A variable declaration will be associated with a data type shown as follows:

Syntax:

```
<data type> var1, var2, …varN;
```

where var1, var2,…varN are the names of the variables of the specified data type.

e.g.: int k; (data type is **int** and variable name is 'k')

float price; (data type is **float** and variable name is 'price')

char name[20]; (data type is **char** and variable name is 'name')

Initialization of a variable

When a value is assigned to a variable while making declaration and is termed as initialization of the variable.

Syntax:

```
<data type> var = value;
```

where 'var' is the name of the variable, '=' is the assignment operator and 'value' may be constant or expression.

e.g.: *int k = 10;(variable 'k' is initialized with value 10)*

Assigning values to variables

Assignment operator (=) is used to assign values to variables. This operator can be used in different forms shown as follows:

Syntax:

variable-name = value; (normal assignment)

variable-name = variable-name =…= value; (multiple assignment)

variable-name binary-operator = value; (compound assignment)

Where *variable-name* is the name of the variable, **=** is an assignment operator, *binary operator* may be an arithmetic, relational, logical operator and *value* may be a constant or an expression.

e.g.:

```
1. int k;
   k = 10;   /* 10 will be assigned to the variable
'k' */
2. int a,b,c;
   a = 10;
  c = b = a; /* first a will be assigned to b and then
assigned to c and values
```

```
of a, b and c are 10, 10 and 10, respectively */
   3. int i;
      i = 10;
      i += 10;  /* i = 20 (i = i + 10) */
      i *= 10;  /* i = 100 (i = i*10) */
```

NOTES

Demo program to show typical characteristics of an assignment operator

```
/*———START OF PROGRAM———*/
#include<stdio.h>
void main()
{
    int a,b,c;
    a = 10; /*normal assignment */
    b = c = a; /* Multiple assignments b=(c = a) */
    a += b + c; /* Compound assignment a = a + b + c
*/
    printf("a =% d, b =% d, c =% d", a, b, c);
}
/*———END OF THE PROGRAM———*/
```

Output: a = 30, b = 10, c = 10

Explanation: In the above assignments, 10 is assigned to **a** (normal assignment), **a** will be assigned to **c** and then assigned to **b** (multiple assignments) and **a + b + c** is assigned to **a** (compound assignment).

Escape sequences in C language

The backslash character '\' is called escape character and is used to escape the usual meaning of the character that follows it.

Escape sequence	Meaning
\a	alert (bell)
\b	backspace (moves the cursor one space to the left)
\f	form feed (advances the printer paper one page)
\n	new line (start at new line)
\0DDD	octal value
\xHHH	hexadecimal value
\r	carriage return
\t	horizontal tab (moves cursor 8 spaces to the right)
\v	vertical tab
\\	backslash
\'	single quote
\"	double quote
\0	null character

3.8 OPERATORS

NOTES

An ***operator*** refers to a symbol which indicates an operation to be executed. Operators are used to manipulate data in a program. The data items that operators act upon are called ***operands***.

e.g.: Sum = A + B

Where 'A' and 'B' are operands and '+' is an operator.

Classification of Operators

- **Arithmetic** [+, –, *, /, %(modulo division)]
- **Assignment** [=, +=, "=, *=, /=, %=]
- **Relational** [>, <, <=, >=, ==, !=]
- **Logical** [&&, ||, !]
- **Conditional** [<expression 1> ? <expression 2> : <expression 3>]
- **Increment and Decrement** [++, —]
- **Bitwise** [~, >>, <<, &, |, ^]
- **Comma Operator** [,]
- **Address operator** [&]
- **Indirection operator** [*]
- **sizeof operator** [sizeof( )]

Note: ~ : is complement

```
>>: is right shift
<<: is left shift
&: bitwise AND
 |: bitwise OR
^: bitwise Exclusive OR (XOR)
```

Arithmetic operators: These operators are used for arithmetic calculations. The C has the following arithmetic operators:

Operator	Name	Purpose
+	plus	addition
–	minus	subtraction
*	asterisk	multiplication
/	slash	division (returns quotient)
%	modulus	division (returns remainder)

Note: Precedence of the arithmetic operators while manipulating arithmetic expressions

First high preference is for the expression within the brackets ( ). Second high precedence is for asterisk (*), slash (/) and modulus (%) and lower precedence is for plus (+) and minus (–) are equal.

NOTES

Demo program based on arithmetic operators

```
/*———START OF PROGRAM———*/
#include<stdio.h>
void main()
{ printf("%d ",10+2);
 printf("%d ",10-2);
 printf("%d ",10*2);
 printf("%d ",10/2);
 printf("%d",10%2);
}
/*———END OF THE PROGRAM———*/
```

Output: 12 8 20 5 0

Explanation: In the above program all basic arithmetic operations have been used on integer values.

Demo program based on the precedence of arithmetic operators

```
/*———START OF PROGRAM———*/
#include<stdio.h>
void main()
{ printf("%d ", 2+3*2);
 printf("%d ", 6/6*3);
 printf("%d ", 3+3/6);
 printf("%d", (3+3)/6);
}
/*———END OF THE PROGRAM———*/
```

Output: 8 3 3 1

Explanation: In the above **printf("%d ", 2+3*2);** statement, asterisk has got more precedence than plus; so the expression value is 8 rather than 10. In the **printf("%d ", 6/6*3);** statement, asterisk and slash have got same precedence and the manipulation will take place from left to right; so value of the expression is 3. In the **printf("%d ", 3+3/6);** statement, slash has got more precedence than plus; so the expression value is 3. In the **printf("%d", (3+3)/6);** statement, expression within the brackets has got more precedence than slash; so the expression value is 1.

NOTES

Relational Operators

These operators are used to compare arithmetic, logical and character expressions.

Operator	Activity
==	equal
!=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

Note:

- All the above operators compare their left-hand side with their right-hand side.
- All binary operators and expression returns Boolean value (0 or 1). So the returning type is integer. If the expression is TRUE, it returns 1; otherwise, 0.

Demo program to show typical characteristics of relational operators

```
#include<stdio.h>
void main()
{
 printf("%d ",2<3); /* true returns 1 */
 printf("%d ",2>3); /* false returns 0 */
 printf("%d ",2==3); /* false returns 0 */
 printf("%d ",2!=3); /* true returns 1 */
 printf("%d ",2<=3); /* true returns 1 */
 printf("%d ",2>=3); /* false returns 0 */
 printf("%d ",2<5<3); /* 2<5 = true(=1) */
        /* 1<3 = true(=1) */
        /* so, 2<5<3 is true */
 }
/*————————END OF THE PROGRAM————————*/
```

Output: 1 0 0 0 1 1 0

Logical Operators

These operators are used to compare or evaluate logical and relational expressions. The result of the logical expression will be either *true* (1) or *false* (0).

Operator	Activity
&&	AND
||	OR
!	NOT

&& (AND) operator: If the expressions each side of the **&&** are true the result of the **&&** is true. If one of them is false the result is false.

Tips to remember the functioning of && and || operators:

General representation	Representation in C
'false' **AND** <anything> becomes false.	(0 && true/false) == 0
'true' **OR** <anything> becomes true.	(1 || true/false) == 1

|| (OR) operator: If either of the expressions each side of the || are true the result of the whole expression is true. The expression is only false if both expressions are false.

! (NOT) operator: for **!** if operand is false (zero value) then result will be true vice versa.

Demo program to display truth table of && (AND)

```
/*———————START OF PROGRAM———————*/
#include<stdio.h>
void main()
{
 printf("0 && 0 = %d \n",0 && 0);
 printf("0 && 1 = %d \n",0 && 1);
 printf("1 && 0 = %d \n",1 && 0);
 printf("1 && 1 = %d \n",1 && 1);
}
/*———————END OF THE PROGRAM———————*/
```

Output:
```
0 && 0 = 0
0 && 1 = 0
1 && 0 = 0
1 && 1 = 1
```

Note: For && if either operands are true (any non-zero value), then result will be true.

Demo program to show the typical characteristics of && (AND)

```
/*——————— START OF PROGRAM ———————————*/
#include<stdio.h>
void main()
{
 int i=5,j=5;
 printf("%d ", 0 && (i=10));
 printf("%d ", 1 && (j=10));
 printf("i=%d j=%d",i,j);
}
/*——————— END OF THE PROGRAM ———————————*/
```

NOTES

Output: 0 1 i=5 j=10

Explanation: If operand1 is false, then && operator will not allow to check the second operand and return value of the result will be 0. For example, in the above example, first **printf** statement's first operand is 0; so irrespective of second expression's value, it will return 0.

NOTES

Demo Program to display truth table of || (OR)

```
/*———START OF PROGRAM———*/
#include<stdio.h>
void main()
{  printf("0 || 0 = %d \n",0 || 0);
   printf("0 || 1 = %d \n",0 || 1);
   printf("1 || 0 = %d \n",1 || 0);
   printf("1 || 1 = %d \n",1 || 1);
}
/*———END OF THE PROGRAM———*/
```

Output:
```
0 || 0 = 0
0 || 1 = 1
1 || 0 = 1
1 || 1 = 1
```

Explanation: If operand1 is true, then || operator will not allow checking the second operand and returning value of the result will be 1. For example, in the above example, second *printf* statement's first operand is 1, so irrespective of the second expression, returns 1.

Note: For || if both operands are false (zero), then result will be false.

Demo program to display truth table of ! (NOT) operator

```
/*———Program Beginning———*/
#include<stdio.h>
void main()
{
 printf("!0 = %d ",!0 );
 printf("!1 = %d ",!1);
}
/*———END OF THE PROGRAM———*/
```

Output:
```
!0 = 1
   !1 = 0
```

Explanation: The above out put shows the truth table of ! operator

Note: for ! if operand is false (zero value), then result will be true vice versa.

Increment and Decrement Operators

C has two useful operators: ++ (***increment operator***) and — (***decrement operator***).

NOTES

The ++ operator adds 1 to its operand and — operator subtracts 1 from its operand. These operators may be either pre-incremented/pre-decremented (precedes the operand) or post-incremented/post-decremented (follows the operand).

e.g.

1.
```
int k,i=1;
k=i++;          /*  k=i; i= i+1;  */
printf("k=%d, i=%d.",k,i);
```

Result of the above statements will be **k=1, i=2**

Note: The expression **i++** will assign first and increments value of **k** later.

```
int k,i=1;
 k=++i;   /* i=i+1; k=i; */
 printf("k=%d, i=%d.", k,i);
```

Result of the above statements will be **k=2, i=2**.

2.
```
int k,i=1;
 k=++i;   /* i=i+1; k=i; */
 printf("k=%d, i=%d.", k,i);
```

Result of the above statements will be **k=2, i=2**.

Note: The expression **++i** will increment 1 first and this value is assigned to **k**.

Note: pre- or post-increments difference exists when we are assigning values; otherwise, there will be no difference.

Example: `i++ or ++i`

Bitwise Operators

These operators are used to operate with bits of data for complementing, testing, setting or shifting the bits of data in a byte or word.

Operator	Operator	Purpose
~	complement	for 1's complementation
>>	right shift	to move bits to right
<<	is left shift	to move bits to left
&	bitwise AND	to reset (0)/compare bits
|	bitwise OR	to set (1) /compare bits
^	bitwise Exclusive OR (XOR)	to toggle or clear the bits

NOTES

Note: Bitwise operators work only with **char** and **int** data types with its qualifiers and are not useful for float, double, void or complex data types.

Demo program for ~ (complement operator):

```
/*———START OF PROGRAM———*/
#include <stdio.h>
void main()
{
 char num=10;
 printf("%d",~num);
}
/*———END OF THE PROGRAM———*/
```

Output: –11

Note:

1's complement of a binary number obtained by changing 0s to 1s and vice versa.

2's compliments = 1's compliment + 1

Comma (,) Operator

The comma operator separates the expressions.

Syntax: exp1, exp2, … expn

Where *exp1, exp2, … expn* are expressions.

e.g.: k= (i=5, j=10, i+j);

Here, first the value 5 is assigned to i, then 10 is assigned to j and the result *i+j* is assigned to k.

Generally, the comma operator is used in **for** loops.

e.g.: for(i=0, j=1; i<=10; i++, j++)

Demo program using comma operator

```
#include<stdio.h>
void main()
{
 int a,b,c;
 a=(b=10,c=20,b+c,b-c);
 printf("%d",a);
}
```

Output: –10

Explanation: In the statement **a=(b=10,c=20,b+c,b-c);** first **b-c** will be calculated, i.e. 10 " 20 = "10 and it will be assigned to the variable **a.**

Address operator (&): This operator is useful in finding the address of a variable of any data type.

NOTES

Indirection operator (*): This operator is used to find the value at a particular memory location. This operator is also called as de-referencing operator.

Note: If both address operator (&) and indirection operator (*) are used side by side (&* or *&), they do nothing to the variable. For example, If 'ptr' is a pointer variable, then &*ptr and *&ptr both are equal to ptr.

sizeof() operator: sizeof operator is used to obtain the size of a variable, which occupies in system's memory. The **sizeof( )** return value(in bytes) is the size of a variable or type within the parentheses .

Syntax: *sizeof(object);*

Where object may be any data type, variable or expression.

e.g.: 1. int n1;

```
n1 = sizeof(int);
2. int n2;
n2 = sizeof(n1);
```

Explanation:

In the first example, **sizeof(int)** returns 2, because size of int data type is 2.
In the second example, *n2* returns 2, because size of integer variable also 2.

CHECK YOUR PROGRESS

1. What are the features of 'C' language?
2. What are some practical applications of C?
3. What are the five basic data types in C?
4. What is the difference between scope and extent of a variable?

3.9 PROGRAM STRUCTURES

A 'C' program may contain one or more parts shown here:

```
Part-1:
   [Comment lines (name of the program, author…)]
   [Link section (to link from system library)]
   [Symbolic constant definitions]
   [Global declarations]
Part-2:
   [return type] main([int arg1 [, char** arg2 [, char** arg3]]])
   {
      [declaration section]
      [statements]
```

```
}
```

Part-3:

```
[Subprograms (user-defined functions)]
   Subprogram section
   Function 1
   Function 2
      (user-defined functions)
         .
   Function n
```

NOTES

The documentation section comprises a set of lines with comments providing program's name, author's name and other details programmer has to use at later stage. The link section gives instructions to compiler for linking functions available in system library. The function of the definition section is to define and these symbolic constants.

There are variables that can be used in more than one function and these are termed *global* variables. They are declared in the global declaration section, located outside and above the functions.

Every 'C' program should necessarily have one section containing **main( )** function. This section has two parts: (i) declaration part and (ii) executable part. The function of the declaration part is to declare all the variables to be used in executable part. The executable part must contain at least one statement. Declaration and executable parts must appear a pair of opening and closing braces. The execution of a program starts at opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. Each statement in both the parts ends with a semicolon.

The subprogram section has user-defined functions. The **main** function calls these functions. They are usually put immediately after the **main** function. However, they may appear in any other order.

Sections, other than the **main** function section, can be absent when not needed. This is illustrated as follows:

Simple 'C' program

```
/* Program for ADDITION */
main( )
{
   int number;
   float amount;
   number = 100;
   amount = 30.75 + 75.35;
   printf("%d\n", number);
   printf("%5.2f", amount);
}
```

Executing a 'C' Program

To execute a C program, you need to take the following steps:

- Create the program
- Compile the program
- Link the program to functions needed, by referring to C library
- Execute the program

These steps have been illustrated in Figure 3.1.

Although these steps remain the same irrespective of the *operating system*, system commands for implementing the steps and conventions for naming *files* may differ on different systems.

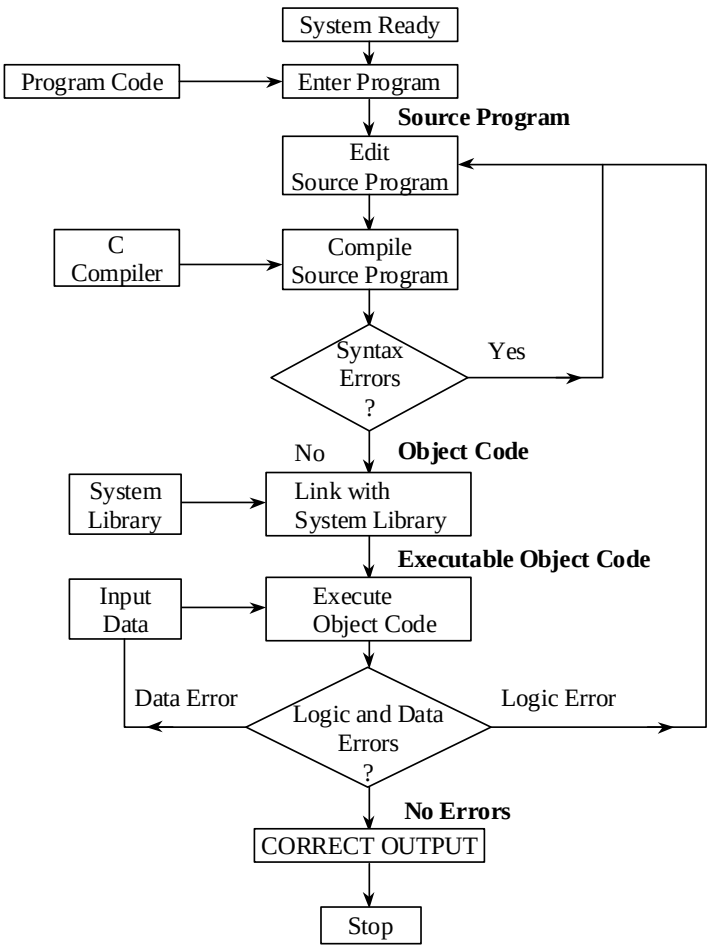


Figure 3.1 *Process of Compiling and Running a C Program*

An operating system is a set of programs controlling all the operations of a computer system. All I/O operations are routed through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

NOTES

NOTES

Compiling and Linking

The program can be created using any word processing software or editor provided by 'C', i.e. Turbo C editor. The file name should end with the characters '.c', like **Sree. c names.c,** etc. Then the command

TCC names.c

Under the MS-DOS operating system would load the program stored in the file **names.c** and generate the **object code.** This code is stored in another file under the name **names.obj.** In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the following command:

LINK names.obj

which generates the **executable code** with the filename **pay.exe.** Now the command

names

would execute the program and give the results.

Instead of performing from command line, we can also use the 'C' editor directly. This enables user's easy way to compile & execute.

Points to Remember

1. Every C program essentially, has a **main( )** function. (Use of more than one **main( )** is illegal.) Program execution begins from main.
2. The execution of a function starts from opening brace of the function and terminates at the corresponding closing brace.
3. C programs should be written in lower-case letters. However, for symbolic names and output strings, upper-case letters should be used.
4. Words in a program line must be separated from each other by a least one space or a tab or a punctuation mark.
5. A semicolon must be used to end every program statement in C.
6. Variables must be declared for their types before use in the program.
7. We must make sure to include *header files* #**include** directive when the program refers to special names and functions that it does not define.
8. Compiler directives such as **define** and **include** are special instructions to the compiler to help it compile a program. They do not end with a semicolon.
9. The sign # of compiler directives must appear in the first column of the line.
10. While using braces for grouping statements, ensure that every opening brace has a corresponding closing brace.
11. C is a free-form language and hence, indentation of various sections should be done to improve readability of the program.

12. Comments can be inserted almost anywhere in a programs. Using of appropriate comments in proper places improves readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and /* appropriately.

NOTES

3.10 ARITHMETIC EXPRESSIONS

The basic arithmetic operators are:

+ addition, e.g., `c = a + b`

– subtraction, e.g., `c = a – b`

* multiplication, e.g., `c = a * b`

/ division, e.g., `c = a/b`

% modulus, e.g., `c = a % b`

When we divide two numbers, we get a remainder and a quotient. To get the quotient we use `c = a/b;`

```
/* c   contains quotient */
```

To get the remainder we use `c = a % b;`

```
/* c   contains the remainder */.
```

% is also popularly called modulus operator. Modulus cannot be used with floating-point numbers.

Therefore, `c = 100/6;` will produce `c = 16`.

`c = 100 % 6`, will produce `c = 4`.

In expressions, the operators can be combined.

For example, `a = 100 + 2/4;`

What is the right answer?

Is it `100 + 0.5 = 100.5`

or `102/4 = 25.5`

To avoid ambiguity, there are defined precedence rules for operators in 'C'. Precedence means evaluation order of operators. However, in an expression there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. In addition, more than one operator may have the same precedence. For example, * and / have the same precedence. To avoid ambiguity in such cases, there is a rule called associativity.

Associativity is either left to right or vice versa. This means that when operators of the same precedence are encountered, the operators having same precedence have to be evaluated from left to right, if the associativity is left to right.

Now refer to the previous example. Since / has precedence over +, the expression will be evaluated as `100 + 0.5 = 100.5.`

NOTES

In the precedence table, operators in the same row have the same precedence. The lower the row, the lower the precedence.

For example, `()`, which represents a function, has a higher precedence than `!`, which is in the lower row. Similarly `*` and `/` have higher precedence over + and −.

Whenever you are in doubt about the outcome of an expression, use of parentheses is a practice to avoid ambiguity.

Consider the following examples:

1) `12 – 3 * 3 = 12 – 9 = 3` and not 27.
2) `24 / 6 * 4 = 4 * 4 = 16` and not 1.
3) `4 + 3 * 2 = 4 + 6 = 10` and not 14.
4)
```
8 + 8 / 2 – 4 * 6 / 2
= 8 + 4 – 4 * 6 / 2
= 8 + 4 – 24 / 2
= 8 + 4 – 12 = 0
```

Watch the steps involved in the last example.

3.11 ASSIGNMENT STATEMENT

Assignment operators are written as follows:

identifier = expression;

For example,

`i = 3`; Note: 3 is an expression

```
const A = 3;
```

'C' allows multiple assignments in the following form:

identifier 1 = identifier 2 = ...= expression.

For example,

```
a = b = z = 25;
```

However, you should know the difference between an assignment operator and an equality operator. In other languages, both are represented by =.
In 'C' the equality operator is expressed as = = and assignment as =.

Shorthand Assignment Operators

You have been looking at simple and easily understandable assignment statements. This can be written in a different manner when the RHS includes LHS; or in other words, when the result of computation is stored in one of the variables in the RHS. The following example will make it clear:

The general form is `exp1 = exp1 + exp2`.

This can be also written as exp1 + = exp2.

Examples:

simple form	special form
`a = a + b;`	`a += b;`
`a = a + 1;`	`a += 1;`
`a= a – b;`	`a – = b;`
`a = a – 2;`	`a – = 2;`
`a = a*b;`	`a*= b;`
`a = a*(b + c);`	`a*= b + c;`
`a = a/b;`	`a / = b;`
`a = a/2;`	`a / = 2;`
`d = d – (a + b);`	`d – = a + b`

The **assignment operators** `=, + =, – =, * =, / =, % =,` have the same precedence or priority; however, they all have a much lower priority or precedence than the arithmetic operators. Therefore, the arithmetic operations will be carried out first before they are used to assign the values.

3.12 LOGICAL EXPRESSION

It would be nice if you could check more than one condition at a time. 'C' provides three logical operators for combining more than one condition. These are as follows:

Logical `and` represented as `&&`

Logical `or` represented as ||

`Negation` or `not` represented as `!` (exclamation).

Some examples on usage of logical operators are given here.

`if x > y` and `if x > z`, then `x` is the greatest.

You will represent the same as,

```
if ((x > y) && (x > y))
printf ("x is the greatest");
```

This program is more elegant.

The syntax for `&&` is,

```
if ((condition1) && (condition2))
    {
    statements-s1
    }
```

Statements–s1 will be executed if and only if both these conditions are true.

The syntax for 'or' is as follows:

NOTES

```
if ((condition 1 ) ||(condition 2))
    {
    statements-s2
    }
```

NOTES

In this case, even if either of these conditions is true, the `statements-s2` will be executed. At least one condition has to be true for the execution of `s2`. However, if both are false, `s2` will not be executed.

The *NOT* operator with symbol **!** can be used along with any other relational or logical operator or as a stand-alone operator. It simply negates the operator following it. The syntax of `'!'` is as follows:

```
if ! (condition) statement s3;
```

`s3` will get executed only when the condition is false.

Consider Algorithm 1 for linear search. Now rewrite Algorithm 1 by using the logical operators. The revised Algorithm 2 is shown later.

ALGORITHM 1

Declaration

```
array  ia  of  size n,  i = 0
item to be searched  = x
found  =  false
do
   if ( ia [i] == x )  then
      found = true
   else
      i  =  i  +  1
while ( i <= n " 1  &&  not found )
```

ALGORITHM 2

Step 1: `If` (`x > y`) and (`x > z`), `x` is the greatest.

Step 2: `Else if` (`x<y`) and (`y>z`), `y` is the greatest.

Step 3: `Else` print z is the greatest.

The complete program is given in Example 3.1.

Example 3.1:

```
This Example demonstrates the use of logical operators*/
#include <stdio.h>
main()
{
    int x,y,z;
    printf ("enter three unequal integers\n");
    scanf("%d%d%d", &x, &y, &z);
    if ((x>y) && (x>z))
```

```
        printf("x is greatest\n");
    else
    {
        if((x<y) && (y>z))
            printf("y is greatest\n");
        else
            printf("z is greatest\n");
    }
}
```

Result of the program

```
enter three unequal integers
12 23 78
z is greatest
```

Now write a program to convert a lower case letter typed into an upper case letter. For this purpose you should refer to the ASCII table in Annexure 1.

If you subtract 32 from ASCII value of a lower case alphabet, you will get the ASCII value of the corresponding upper case letter. Now write an algorithm for the conversion of lower case to an upper case letter. It is given in Algorithm 3.

ALGORITHM 3

Step 1: Send a message for getting a character

Step 2: Get a character

Step 3: Check whether the character typed is >=a and <=z

(This is essential since you can only convert a lower case alphabet into upper case.)

Step 4: If so, subtract 32 from the value of the character; if not, go to step 6

Step 5: Output the character with the revised ASCII value; END

Step 6: Print 'an invalid character' END

The algorithm is implemented in Example 3.2.

Example 3.2:

```
Conversion of lower case letter to upper case*/
#include <stdio.h>
main()
{
    char alpha;
    printf ("enter lower case alphabet\n");
    alpha=getchar();
    if (( alpha >='a')&& (alpha<='z'))
    {
        alpha=  (alpha-32);
```

NOTES

NOTES

```
        putchar (alpha);
    }
    else
        printf("invalid entry; retry");
}
```

Now you can test the program by giving both the valid and invalid inputs; valid inputs are the lower case letters and invalid inputs are all other characters.

Result of the program

The result for the invalid input is as follows:

```
enter lower case alphabet
8
invalid entry; retry
```

The result when tried with a valid input is given below:

```
enter lower case alphabet
n
N
```

The programs should be executed, i.e., tested with both the valid and invalid inputs.

3.13 SEQUENCING

Sequencing in 'C' represents the sequence construct. The term 'sequence construct' corresponds that statements are arranged and executed sequentially that is step-wise. The following flow chart (Figure 3.2) shows the default flowing of statement that maintains sequencing in 'C' language:

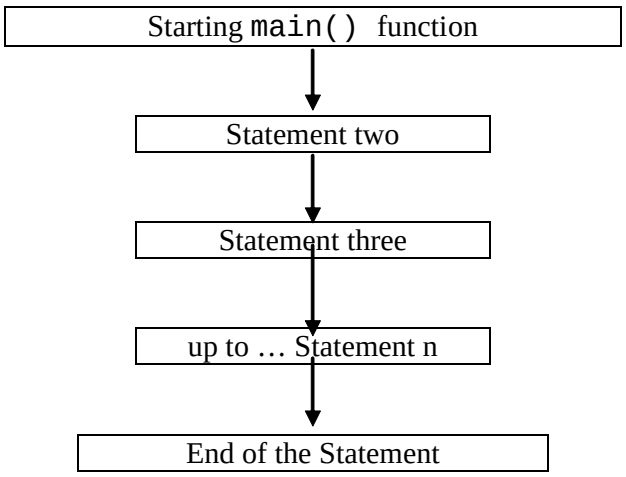


Figure 3.2 *Flowchart—The Sequence Construct in C*

NOTES

For example, the sequencing in C represents that every C program must be defined with header files and start with `main()` function. Body of the main function is turned to execute the sequence. For example, let take an example that deals escape sequences correspond to special character. These special characters are printing characters making the output readable when printing characters are appeared on the screen, file and other output device, for example, printer. The following example shows that how sequencing concept is maintained that makes a new line between two sentences:

```
#include <stdio.h>
#include <conio.h>
void main()
{
printf("\n Information");
printf("\n Technology");
getch();
}
```

The result of the above program is as follows:

```
Information
Technology
```

The above program is a simple program that deals sequencing concept. It first includes two header files named as `#include <stdio.h>` and `#include <conio.h>`. According to above mentioned flowchart, `void main()` function is taken that contains statement as the body of the function. In the above code, alphabet `n ('\n')` makes a line feed that means new line. The function `printf("\n Technology");` prints the word `'Technology'` in the new line that is taken as third statement in the body of the `main()` function. The function `getch();` is used to press any key takes user to the program editing screen. But, the pressed character is not echoed into the screen. The opening and closing curly braces `{}` contain the body of the `main()` function. A sequence concept is analysed that supports properly defined syntax along with standard format as per required output. The system unit executes C statements step-wise-step in which they are defined and written.

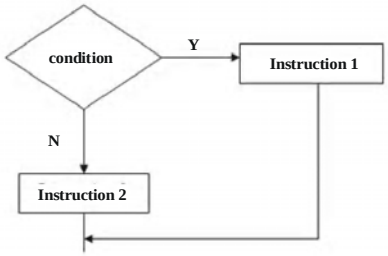
3.14 ALTERATION AND ITERATION

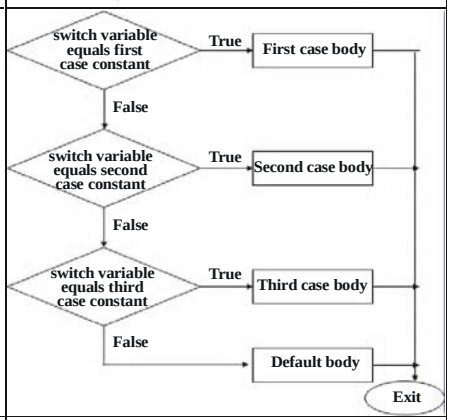
Alteration technique in computer programming includes keywords as **if-else, switch, break, continue** and **comma operation**. The syntax of each of the alterations used in computer programming is as shown in Table 3.5:

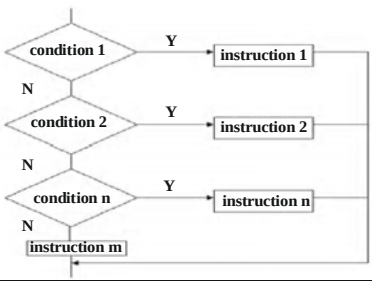
NOTES

Table 3.5 *Alterations Used in Computer Programming*

Alteration Operations	Syntax	Flowchart

if-else	The syntax of if -else is written as follows:
	`if ( condition )`
	`{`
	`expression_set_one;`
	`}`
	`else`
	`{`
	`expression_set_two;`
	`}`
		

switch	The syntax of switch case statement is written as follows:
	`switch ( test ) {`
	`  case 1 :`
	`// Process for test = 1`
	`  ...`
	`  break;`
	`  case 5 :`
	`// Process for test = 5`
	`  ...`
	`  break;`
	`  default :`
	`// Process for all other cases.`
	`  ...`
	`}`
		

break	The syntax of break statement is written as follows:
	`case 10:`
	`// Expression for case`
	`  break;`
		

continue	The continue keyword is declared with jump statement in the following way:
	`jump-statement:`
	`continue;`	

comma operation	The comma operation is used with `'for'` loop. The `'for'` loop allows comma separated initialization. For example,
	`for(i=0,j=0; i<100; i++)`
	`//Comma operation`
	`{`
	`-`
	`-`
	`-Bock of statement`
	`}`	

The following program shows how alteration statements are used in the 'C' language.

```
#include <stdio.h>
#include <conio.h>
#include <process.h> //For exit() function

void main()
{
clrscr();//Clears the screen
void calc_values (int, int, char);
int value_1, value_2;
char ch;
printf("\nEnter two integer values :\n");
scanf("%d%d", &value_1, &value_2);
printf("\n Enter Arithmetic operator[+,-,*,/,%] : \n");
scanf("%c",&ch); //Accepting input value as character
type
calc_values (a, b, ch);
return 0;
}
void calc_values(int x, int y, char c)
{
switch(c)
{
case '+':printf("\n Sum of %d and %d is  = %d", x, y,
x+y);
             break;
case  '-' : printf("\n Product of %d and %d = %d", x, y,
x-y);
            break;
case '*': printf("\n Product of %d and %d = %d", x, y,
x*y);
             break;
case '/': if (x<y)
{
   printf("\n First integer must be greater than second
:");
                exit(0);
 // The function exit(0)causes the program to exit as
successful  termination.
}
```

NOTES

NOTES

```
printf("\n First integer must be greater than second :");
break;
case '%': if(x>y)
{
 printf("\n First integer should be greater than second");
exit(0);
}
printf("\nRemainder : %d", x%y);
break;
default: printf("\n Wrong operands !!!");
}
break;
}
getch();
}
```

The result of preceding program is as follows:

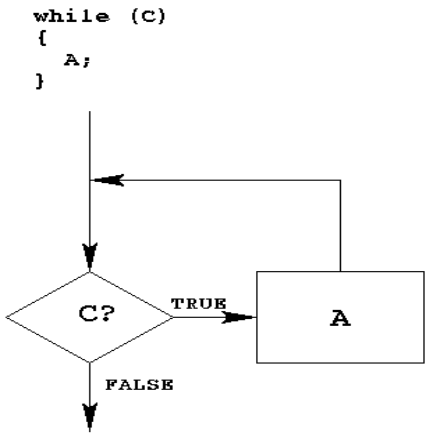
```
Enter two integer values : 3 2
Enter Arithmetic operator[+,-,*,/,%] :+
Sum of 3 and 2 is  = 5
```

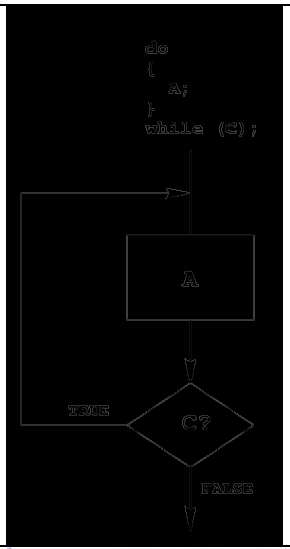
Repeating calculation is known as iteration. Iteration controls while loop, do- while loop, for loop and nested loops. The following Table 3.6 shows the syntax of each of the iterations used in computer programming.

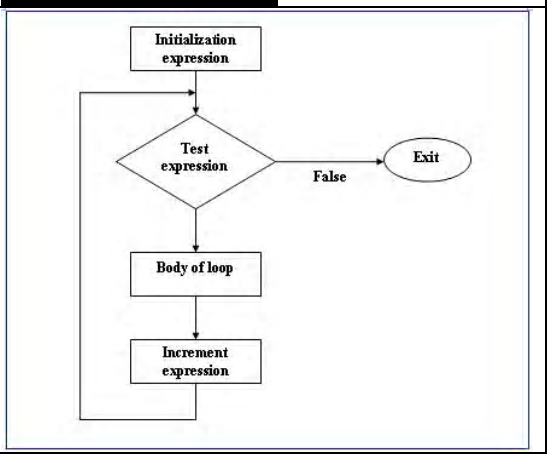
NOTES

Table 3.6 *Iterations Used in Computer Programming*

Iteration Operations	Syntax	Flowchart

while	The syntax of while loop is as follows:
	`while (expression)`
	`{`
	`-`
	`-`
	`- Block of statements is written that is to be executed`
	`}`
		

do … while	The syntax of
	The syntax of while loop is as follows:
	`do{`
	`-`
	`-`
	`- Block of statements is written that is to be executed`
	`} while (expression);`
		

for	The 'for' loop is used to execute the expression's series with multiple number for times.
	`for (expression_one, expression_two; expression_three)`
	`{`
	`-`
	`-`
	`- block of statements to execute`
	`}`
		

3.15 STRING PROCESSING

NOTES

String processing represents the various operations involved in string manipulation and string collection. Handling of character strings is the initialization, processing and manipulation of character or string. The character strings are handled and manipulated in 'C' language for following meaningful operations:

- Reading and displaying the string
- Concatenating the string
- Copying string from one location to other
- Comparing the string whether they match or not
- Extracting the part of string or character

A pseudocode used in string processing is written as follows:

```
int StringSearch(char s1[], char s2[])
//If the first value matches S1 with S2 the value
returns an index of S2 corresponding to this value
with the next one.
//If there is no match, a non-zero (-1) value is returned
{
  int i, j, M = strlen(s2), N = strlen(s1);
  for(i=0, j=0; j< M && i < N; i++, j++)
    if (s1[i] != s2[j])
      {
         i -= j;
         j = -1;
      }
    if (j==M)
   return i-M;
    else return -1;
}
```

In the above algorithm, `s1` and `s2` are taken as two strings that are represented by arrays of characters The function `strlen()` is used as standard library function used in C and C++. This function finds the total length of the specified string no matter space is given. The provided string for `s1` is `'ababcab'` and for `s2` is `'abc'` can be explained the step-wise string processing as follows:

NOTES

```
'ababcab'              i=0,j=0
'abc'
 ^                     matches: increment i and j

'ababcab'              i=1,j=1
'abc'
  ^                    matches: increment i and j

'ababcab'              i=2,j=2
'abc'
   ^                   match fails: i=i-j=0, j=-1,
                       increment i and j

'ababcab'              i=1, j=0
 'abc'                 match fails: i=i-j=1, j=-1
  ^                    increment

'ababcab'              i=2, j=0
  'abc'                matches: increment i and j
   ^

'ababcab'              i=3, j=1
  'abc'                matches: increment i and j
    ^

'ababcab'              i=4, j=2
  'abc'                matches: increment i and j
     ^

                       i=5, j=3, exit loop (j=M),
                       j=M so return i-M = 2
```

Characters or strings are used in memory as ASCII codes. The appended string is used with '\0' that is considered as NULL value in ASCII list. Each character in memory location takes one byte. The rest of successive characters get processed as successive bytes.

Character	**M**	**y**		**a**	**g**	**e**		**i**	**s**
ASCII Value	77	121	32	97	103	101	32	105	115
Character		2		(	t	w	o	)	\0
ASCII Value	32	50	32	40	116	119	111	41	0

Initializing String

In 'C', characters or strings are initialized in one dimensional array as follows:

```
char name_of_month={'J', 'U', 'L', 'Y'};
```

The characters of initialized strings are enclosed with a part of double quotes. The compiler stores the ASCII codes of characters and put into the memory. Let us take an example as:

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

NOTES

```
{
char month[15];
printf ("Enter the name of month:");
gets (month);
printf ("\nName of Month is = %s", month);
}
```

In this example string is stored in the character variable month the string is displayed in the statement. The result of the above program is as follows:

```
Enter the name of month: July
Name of Month is = July
```

It is an array of one dimension. Each character takes one byte. A null character (\0) has ASCII value 0 and it terminates the string. The figure shows the storage of string JULY in the memory recall that \0 specifies a single character whose ASCII value is zero.

J
U
L
Y
\0

String variable is declared as follows where variable `string_name` keeps the size of the specified array:

```
char string_name[size];
```

Example:

```
char month[10];
char address[100];
```

Compiler appends NULL value or `'/0'` at the end of specified string. The `scanf()` function works with `%s` format that is read string text from the system terminal.

String Processing Operations

The various string operations are dealt in 'C' language. For this, various string functions are used. The string functions are used in 'C' as follows:

`strlwr()` function

This function `strlwr()` changes the specified character or string in lowercase.

```
#include <conio.h>
//Header file is included for console input-output
operation
#include <stdio.h>
//Header file is included for standard input-output
```

NOTES

```
operation
#include<string.h>
// Header file is included for string functions
void main()
//Declaring main() function as void
{
      clrscr(); //Clear the screen
     char str_string[]="INFORMATION TECHNOLOGY";
//String is defined
     printf("Original string is=%s", str_string);
//Prints the defined string
      strlwr(str_string);
//Change the string in lowercase
    printf("\nString is in lowercase=%s", str_string);
//Prints the string in lowercase
      getch();
//Does not echo the character when key is pressed from
keyboard
}
```

The result of the program is as follows:

```
Original string is=INFORMATION TECHNOLOGY
String is in lowercase=information technology
```

`strupr()` function

This function `strupr()` changes the specified character or string in lowercase.

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
void main()
{
  clrscr();
  char str_string[]="information technology";
//String is defined
printf("\nOriginal string is=%s", str_string);
//Prints the defined string
strupr(str_string);
//Change the string in uppercase
printf("\nString is in uppercase=%s", str_string);
//Prints the string in uppercase
```

```
getch();
}
```

The result of the program is as follows:

```
Original string is=information technology
String is in uppercase=INFORMATION TECHNOLOGY
```

NOTES

strrev() function

This function strrev() reverses the specified character or string.

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
void main()
{
        clrscr();
        char str_string[]="HELLO!";
        //String is defined
       printf("\nOriginal string is=%s", str_string);
       strrev(str_string);
       printf("\nReversed String is=%s", str_string);
  getch();
}
```

The result of the program is as follows:

```
Original string is=HELLO!
Reversed String is=!OLLEH
```

strlen() function

The function strlen()returns the total length of specified string.

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
void main()
{
   char str_name[20];
   int i;
    clrscr();
    printf("\nEnter a string:");
    scanf("%s",str_name);
    i=strlen(str_name);
      printf("\n Length of string   (%s) is = %d",
str_name,i);
```

```
    getch();
}
```

The result of the program is as follows:

```
Enter a string: Flower
Length of string (Flower) is = 6
```

NOTES

strcat() function

The two strings are concatenated by `strcat()` function. They are combined if the characters of one string are added at the end of other string. This is known as concatenating process. The following syntax is preferred to concatenate the two specified string values:

```
strcat(string_one, string_two);
```

For example,

```
strcpy(string_one,"Red");
strcpy(string_two,"Flower");
printf("%s",strcat(string_one,string_two));
```

The result of this code is `RedFlower`

strcmp() function

In 'C' language, the `strcmp()` function is used to compare the two strings. This function returns a value 0 if two strings match otherwise returns a non-zero value if two given strings do not match exactly. The syntax for `strcmp()` function is written as follows:

```
strcmp(string_one,string_two);
```

The following examples return the value as follows:

`strcmp("Lessen",Lessen");` returns 0 because two strings are equal.

`strcmp("The", "the");` returns a value 9 because the numeric difference comes for ASCII values as 32 between ASCII `'T'` and ASCII `'t'`.

An array of characters known as string is encapsulated with double quotes.

3.16 SUBPROGRAMS

Subprograms refer to functions or subroutines in programming languages. They are basically dependent on the main program and hence known as subprograms. The basic idea behind subprogram is to make a group of collection of statements invoking by the name. However, the semantics idea of calling subprogram is universal, for example, this concept is used in FORTRAN as an odd variation.

NOTES

Some computer languages allow the definition of subprogram inside the subprogram. The computer language Ruby is used to make `def` operator an executable that is defined conditionally to the subprograms. The computer languages Ada, FORTRAN 95 and Python support keywords as parameters in declaring subprograms and support default values. The Ruby language uses an associative array for this mechanism, whereas Python does not use associative array in declaring subprograms. The languages C, Perl, C++ and JavaScript do not accept the same number of arguments as parameters in declaring subprograms. In C language, subprograms are known as functions. If a program contains more number of subprograms, its execution speed is fast and more flexible. The subprogram statement contains the following layout:

```
subprogram_statement
{
-Specification_part   //It is declarative part
-Execution_part      //Sequence of statements is written
-Internal_subprogram_part
-end_subprogram_statement
}
```

The declaration of subprogram is based on programmer and hence optional. In exceptional case, the body of subprogram acts as declaration. However, a corresponding body must be included to make a link. If both declaration part of the body and statement part of the body are given, only the body of subprogram uses rules and regulations of the proper syntax. The following format shows the layout of subprograms defined in high level languages.

```
[Sub programs (User-defined functions)]
          Subprogram section
          Function 1
          Function 2
          (User-defined functions)
          Function n
```

For example, a subprogram is declared in 'C' language as follows:

```
int main()
{
          int i,j;
          char ch;

         -
         -
         -
}
```

NOTES

```
function_one()
{
        -
        -
        -
}
function_two()
{
        -
        -
        -
}
```

A program unit contains an external subprogram. It contains many functions within one main function.

```
main()
{
math_double()

//It takes two double agreements and returns double
value
{
                -
                -
                -
}
total_sum()

//It takes an integer array and returns a long result
{
                -
                -
                -
}
```

For example, in C language, the general syntax of subprogram is written as follows:

```
<Subprogram-Header>
<Body-of-Subprogram>
void swapped_value(int &number_one, int &number_two)
{
            int temp_value;
```

NOTES

```
//Declaring temp_value
            temp_value=number_one;
//Assigning temp_value is equal to number_one
            number_one=number_two;
//Assigning number_one is equal to number_two
            number_two= temp_value;
//Assigning number_two is equal to temp_value
}

int compute_Avg_value(void)
//Declare an integer function as compute_Avg_value
containing void parameter
{
      int val, total_val=0;
for(int i=0;i<10;i++)
//Declaring for loop starting from 0 to 10
{
    scanf("%d", &val); //Accepting input value as per
for loop
    total_val+=val;
//Calculates total value increased by val value
}
return(total_val/10);
}
```

Basically, the two types of subprograms in C language are known as functions and procedures. In the above coding, the subprogram can have formal parameters. If a subprogram does not contain global variables it exchanges the values with the calling program. Sometimes, one-dimensional array is used as parameters in subprogram declaration. In C, C++, FORTRAN, PHP, Ada the formal parameters or arguments are used if no parameter is passed for the formal parameters. For example,

```
Function Header: float compute_way(float income, float
tax_rate, int exemptions =1)
Function Calls:     pay = compute_pay (10000.0, 0.12);
Pay= compute_pay (12000.0, 0.12, 3);
```

***Note*:** In C, parameters along with default arguments must be declared at last in the argument list.

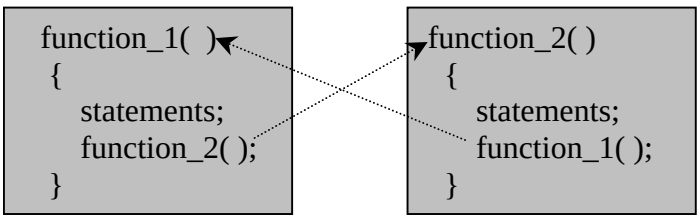
The body of subprogram can be elaborated. The execution of subprogram is invoked by a subprogram call. The association is established between formal and actual parameters in which declarative part of body is elaborated followed by the sequence of the statements of the body. After completing the body of

subprograms (functions), the value is returned to the return statement which is declared at the end of the body of the function.

3.17 RECURSION

NOTES

- *Recursion* is a powerful technique, which is used to call a function itself.
- A function, which invokes itself repeatedly until some condition is satisfied, is called a *recursive function.*
- The number of recursive calls is limited to the size of the stack.
- Two or more functions, which invoke (call) each other, are called *mutually recursive functions.*



Here the first function **function1()** calls the second function; **function2()**, which in turn calls the first function **function1()** again. These two functions are, therefore, called **mutually recursive functions**.

- For certain problems, a recursive solution is straight forward and single as in case of factorial of a number; tower of Hanoi problem, finding GCD, finding Fibonacci series, etc.
- A recursive function is often less efficient compared to an iterative function, but it is more elegant.
- An iterative function is preferred when its recursive equivalent is complex.
- Recursion is not always the best approach. In some cases, using bottom-up (stack implementation) is a good approach to remove recursion from an algorithm. For example, to generate Fibonacci series through bottom-up approach is more efficient than recursive method.

Example programs

1. Write a program to display 1 to 10 numbers using recursion.

Recursive function having without arguments and return value

```
/*————————START OF PROGRAM———— */
#include<stdio.h>
void main()
{
```

NOTES

```
 static int i = 1;
 printf("%d ", i);
 i++;
 if(i <= 10)
 main();
}
/*————————END OF PROGRAM————————————*/
```

Output:

1 2 3 4 5 6 7 8 9 10

Explanation: This program prints 1–10 numbers by calling **main()** function recursively without having arguments.

2. Write a program to display 1 to n numbers using recursion.

Demo program for recursive functions having fixed number of arguments

```
/*————————START OF PROGRAM————————————*/
#include<stdio.h>
void display(int n)
{
 if(n == 0)
 return;
 else
 display(n – 1);
 printf("%d, n);
}
void main()
{display(10);
}
/*————————END OF PROGRAM————————————*/
```

Output:

1 2 3 4 5 6 7 8 9 10

Explanation: This program prints 1 – n numbers by calling **display()** function recursively having argument as **n.**

3. Write a program for recursive functions having fixed number of arguments having return value.

Finding factorial of a given number

```
/*————————START OF PROGRAM————————————*/
#include<stdio.h>
int fact(int n)
```

```
{
 if(n)
 return n*fact(n – 1); /* calling fact function recursively
*/
 else
 return 1; /* control will return to the calling function
*/
 }
void main()
{
 int num;
 printf("Enter a number:");
 scanf("%d", &num);
 printf("Factorial of %d is %d", num, fact(num));
}/*————————END OF THE PROGRAM—————————*/
```

Output:

Factorial for 4 is 24.

Explanation: This program accepts a number and passes the number through the function **fact(num)**. The function **int fact (int n)** performs factorial operation **n*fact(n – 1)** till **n** becomes zero. If n value is zero, then it returns 1 to the calling function.

4. Write a program to add sum of digits of a given number using the recursion method.

Adding digits of a given number

```
/*—————————START OF PROGRAM———————————*/
 #include<stdio.h>

 int sumd(int num)
 {
if(num == 0)
    return 0;
   else
return sumd(num/10) + num%10; /* calling sumd function
*/
                     / *recursively */
 }
void main()
 {
 int num;
 printf("Enter a number:");
```

NOTES

```
 scanf("%d", &num);
 printf("Sum of digits: %d", sumd(num));
 }
/*————————END OF THE PROGRAM———————*/
```

Output:

Enter a number: 123

Sum of digits: 6

Explanation: This program accepts a number and prints the value of sum of digits of it by calling function **sumd(num).** The function **int sumd(int num)** performs recursion operation until num value becomes zero.

5. Write a program for tower of Hanoi using the recursion method.

Tower of Hanoi using recursion

```
/*———————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
void hanoi(int n, int startn, int midn, int destn)
{if(n!= 0)
 {hanoi(n – 1,startn, destn, midn);
 printf("\nmove disk %d from %c to %c", n, startn, destn);
 hanoi(n-1, midn, startn, destn);
 }
}
void main()
{
 char sn = 'a', mn = 'b', en = 'c';
 int n;
 clrscr();
 printf("Enter a value for n:");
 scanf("%d", &n);
 printf("Tower of Hanoi problem with %d disks", n);
 hanoi(n, sn, mn, en);
}
/*—————————END OF THE PROGRAM—————————*/
```

Output:

```
Enter a value for n: 3
Tower of Hanoi problem with 3 disks
move disk 1 from a to c
```

```
move disk 2 from a to b
move disk 1 from c to b
move disk 3 from a to c
move disk 1 from b to a
move disk 2 from b to c
move disk 1 from a to c
```

Explanation: The above program performs Tower of Hanoi problem for the given number of disks

6. Write a program, which simulates dir/s command using the recursion method.

Simulation of DIR/S command using the recursion technique

```
/*————————Name of the file: disp_all.c————————*/
/*———————————STAR OF PROGRAM———————————*/
#include<stdio.h>
#include<dos.h>
#include<dir.h>
void search(char *path)
{
 int i;
 static int tab = 0;
 struct ffblk ff; /*ffblk is a DOS file control block
structure */
 chdir(path); /* chdir() is a built-in function, which
*/
 /* changes current directory */
 printf("\n");
 for(i = 0; i < tab – 1; ++i)
 printf(" ");
 printf("[%s]",path);
 i = findfirst("*.*",&ff,FA_DIREC|FA_ARCH|FA_RDONLY);
 /* findfirst() searches disk directory */
 while(i! = – 1)
  {if((ff.ff_attrib & FA_DIREC) == FA_DIREC &&
ff.ff_name[0]!='.')
    {tab++;
 search(ff.ff_name);
 }
 else
 {printf("\n");
 For (i = 0; i < tab; ++i)
```

NOTES

NOTES

```
 printf(" ");
 printf("%s",ff.ff_name);
 }
 i = findnext(&ff); /* findnext()continues search for
the next File
 (or)Directory */
 }
 chdir("..");
 tab–;
}
void main(int argc,char *argv[]) /*command line arguments
are used */
{clrscr();
 search(argv[1]);
}
/*————————END OF THE PROGRAM————————*/
```

Usage of the above program: At the command prompt type **disp_all <path of a directory>** where this program has been stored.

```
e.g.: C: \> disp_all c:\
```

The above program displays files in the specified directory as well as all subdirectories.

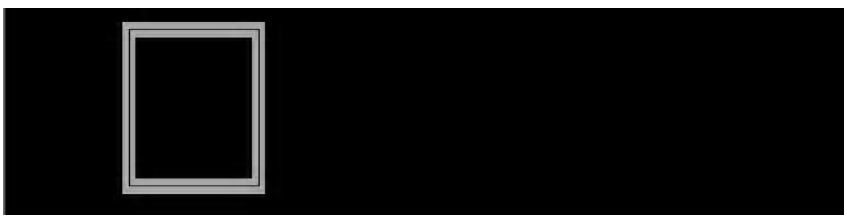
7. Write a program to print a box in a text mode for given coordinates with specified colour (Standard VGA colours only: BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, LIGHTGRAY, DARKGRAY, LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, and WHITE).

Printing a box in the text mode

```
/*————————STAR OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
void box(int x1, int y1, int x2, int y2, int bg, int
fg)
{
 int i;
 textcolor(fg); /* selects new character color in text
mode */
 textbackground(bg); /* select new text background color
*/
 gotoxy(x1, y1); /* positions cursor in text window */
 cprintf("%c", 218); /* writes formatted output to the
text*/
```

```
 /* window on the screen */
gotoxy(x1, y2);
cprintf("%c", 192);
gotoxy(x2, y1);
cprintf("%c", 191);
gotoxy(x2, y2);
cprintf("%c", 217);
for(i = x1 + 1; i < x2; ++i)
{
 gotoxy(i, y1);
 cprintf("%c", 196);
 gotoxy(i, y2);
 cprintf("%c", 196);
}
for(i = y1 + 1; i < y2; ++i)
{
 gotoxy(x1, i);
 cprintf("%c", 179);
 gotoxy(x2, i);
 cprintf("%c", 179);
}
}
void main()
{
 box(10, 10, 20, 20, WHITE, BLACK);
}
/*————————END OF PROGRAM————————*/
```

Output:



Write a program to display a popup menu.

Displaying a popup menu

```
/*————————START OF PROGRAM————————*/
#include<stdarg.h>
```

NOTES

NOTES

```
#include<conio.h>
/*include the box function of the previous program */
int menu(int x, int y, int bg, int fg, int n,...);
void main()
{
 printf("%d", menu(10, 10, WHITE, BLACK, 3, "File", "Edit",
"View"));
}
int menu(int x, int y, int bg, int fg, int n,...)
{
 char ch,str[20][20];
 int len, i, index = 0;
 va_list va;
 va_start(va, n);
 for(i = 0; i < n; ++i)
 strcpy(str[i], va_arg(va, char *));
 va_end(va);
 len = strlen(str[0]);
 box(x - 1, y - 1, x + len, y + n, bg, fg); /* the function
box defined in
the previous program */
 while(1)
 {
 for(i = 0; i < n; ++i)
 {if(index == i)
 {
    textcolor(bg);
    textbackground(fg);
 }
 else
 {textcolor(fg);
 textbackground(bg);
 }
 gotoxy(x, y + i);
 cprintf("%s", str[i]);
 }
 Ch = getch();
 if(ch == 0)
 ch = getch();
 switch(ch)
 {
```

```
case 72: index--;
    if(index < 0)
    index = n - 1;
    break;
case 80: index++;
    if(index >= n)
    index = 0;
    break;
case '\r': return(index);
 }
}
}
/*————————END OF THE PROGRAM————————*/
```

Output:

Explanation: This program creates a popup menu with specified number of options using the function **int menu(int x, int y, int bg, int fg, int n,...)** where **x** indicates x-axis, **y** indicates y-axis, **bg** indicates background, **fg** indicates foreground, **n** indicates number of options for the popup menu and **…**(**ellipse** symbol) indicates passing the variable number of arguments, which are having equal length of strings.

In the above output, the popup menu consists of three options, namely, **File, Edit** and **View**. If you select **File,** it returns zero; if you select **Edit,** it returns one and if you select **View,** it returns two.

3.18 ARRAYS

An array refers to a group of similar elements that share a common name and are differentiated from one another by their position within the array.

Why do we need an array?

Ordinary variables are capable of holding a single value. However, there are situations where we want to store more than one value in a single variable.

For example, if you want to present the percentage of marks obtained by 100 students in the ascending order, you have the following two options for storing these marks in the memory:

NOTES

(*i*) Constructing 100 variables for storing the percentage of marks obtained by 100 students, i.e., each variable indicating one student's marks.

(*ii*) Constructing one variable, called an array, which can store memory locations. These similar elements may be all int's or all char's or all float's.

Like any other variable, an array should be declared so as to make the compiler aware of the type and size of an array.

Syntax:

Data type array_name [dimension]

```
int x[5];
```

Few important points about an array

The name of an array cannot be same as that of any variable declared within the program.

The size of the array is specified using subscript notation.

The subscript used to declare an array is called dimension.

The dimension used to declare an array must always be a positive integer constant.

Subscripts are non-negative integers, which are used to access the array element contents. The dimensionality of an array is determined by the number of subscripts. For example, ***age[i]*** is an element in the one-dimension; correspondingly, ages[i][j] is an element in the two-dimensional array ages.

They are written when we declare an array. The following example shows the subscript operator.

```
int marks [10];
```

Here the open and close square brackets are called subscript operator. Inside this subscript operator, we mention the size of array.

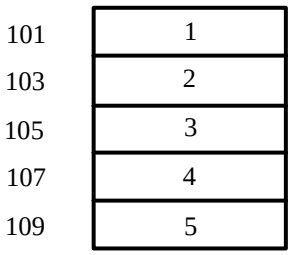
Restrictions: The subscript value must always be the integer constant while defining an array. It should not be a negative integer. We can use the constants that are defined using the #define preprocessor directive statement.

Assigning Constant Value

A global variable is one that is declared outside any function (usually before main). A special feature of the global arrays is that they can be initialized when they are declared. This is done by following an array name with dimension and equal sign, followed by a pair of braces. These braces contain a series of constant values separated by commas.

```
Ex: int name[5] = {1, 2, 3, 4, 5};
```

The memory map will look like:



It is not always necessary to specify dimension of the array if it is declared as follows:

```
int name[ ] = {1, 2, 3, 4, 5};
```

Since the square brackets following the array name are empty, the compiler determines how many elements to allocate for array by counting number of values within curly braces.

This approach can avoid errors. If dimension is specified explicitly, values then are needed; a syntax error is flagged by the compiler.

Defining the size of each array as a symbolic constant makes programs more scalable. The 'C' language treats character strings simply as an array of characters.

```
Ex: char name[10]
```

Name character array, which holds maximum of 10 characters.

When the compiler sees a constant string, it terminates it with an additional null character called NULL character.

Static: We have seen how global arrays can be initialized. The global variables are classified in C as static variables, which mean they come into existence when the program is executed and continue to exist until the entire program terminates.

```
Ex: static int num;
```

This is to declare a local variable num as static.

In array: static int num[10];

Initialization and storage classes: We have a convenient way to initialize the array at the beginning of a program. However, only static and external array can be initialized, whereas automatic and register array cannot be initialized

Before trying to initialize an array, let us see what is there in it if we don't put anything there.

```
main()
{
   int fun[2];          /*automatic array */
   static int win[2];         /*static array */
```

```
    printf("%d %d\n", fun[1], win[1]);
}
```

The output of this program is

```
< garbage value > 0
```

NOTES

This reflects the following rules. If you do noting, external and static arrays are initialized to zero. Automatic and register arrays get whatever garbage happens to be left over in that part of memory.

When the array elements are initialized by at least one initializer, then the remaining elements are initialized to zero.

During compile time, a static array gets initialized and at run-time, an automatic array gets initialized.

Example 3.3:

```
main()
{
int days[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31};
int i;
for(i = 0; i < 12; i++)
printf("days[%d] = %d", i, days[i] );
}
```

The output of the following program is:

```
days[0] = 31
days[1] = 28
days[2] = 31
days[3] = 30
days[4] = 31
days[5] = 30
days[6] = 31
days[7] = 31
days[8] = 30
days[9] = 31
days[10] = 0
days[11] = 0
```

Array initialization: An array can be initialized while declaring or we can store values in it during program execution. When the array is initialized where it has been declared, you may or may not mention its dimensions.

Example 3.4:

```
int num [6] = {1, 2, 3, 4, 5, 6};
int p[] = {2, 4, 6, 8};
```

NOTES

If the square brackets following the array name are empty, the compiler determines the number of elements to be allowed by counting the number of values within the curly braces. If the entered elements in the array are less than the size of the array variable, then the remaining elements are left unused with some garbage values stored into them.

Example 3.5:

```
/* To illustrate array initialization and printing values
*/
    #include<stdio.h>
    main ( )
    {
       int a [5] = {1, 2, 3, 4, 5}, i;
              // Values can be initialized to the array
while declaring using {}
       float b[] = {1, 2, 5.12, 3.45};
             /* It is not necessary to give length while
initializing */
    char name [10] = "SRINU"; /* String assigning to
array */
       printf("\n The elements of array a are \n\n");
       for(i = 0; i < 5; i++)    /*  printing  the
elements through loop */
       {
          printf("\n%d element of array a = %d", a[i]);
       }
       printf("\n\n The elements of array b are \n\n");
       for(i = 0; i < 3; i++)
       {
          printf("\n%d element of array b = %f", i,
i[b]);
       }
       printf("\n\n The string is = %s\n\n", name);
    }
```

Output:

The elements of array a are:

0 element of array a = 1

1 element of array a = 2

2 elements of array a = 3

3 elements of array a = 4

4 elements of array a = 5

NOTES

The elements of array b are:

0 element of array a = 1

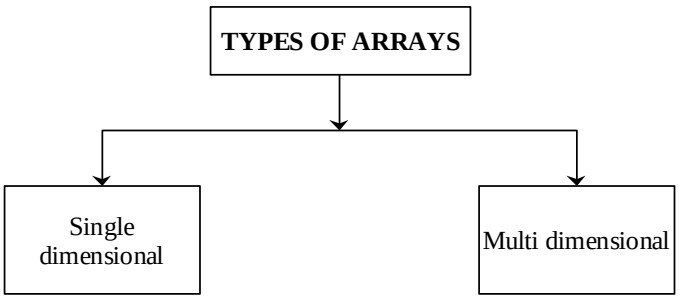
1 element of array a = 2

2 elements of array a = 3

The string is = SRINU

Types of Arrays

Arrays are classified into two types, which are as follows:



Single-dimensional array

It is an array consisting of a single row or a single column.

Single-dimensional array (vector) represents the number of consecutive memory locations in which data are stored and each element can be accessed through an index.

Declaration of single-dimensional arrays

Syntax:

```
<type> <variable_name>[<index>];
```

Example 3.6:

```
int num[10]; /* an array of 10 integers */
char vowels[5]; /* an array of 5 characters*/
...
```

Note: In C, array index starts with 0

Example 3.7:

```
int x[3];
Represented as [1 2 3]
```

Example 3.8:

```
/* Program to declare an array, accept number to array
through keyboard and print them */
    #include<stdio.h>
    main ( )
```

```
{
   int arr [10], i; /* Array arr declaration */

   for (i = 0; i < 10; i++)
   {
      printf("Enter % d value:", i + 1);
      scanf(%d", & arr[i]);   /* Accepting values */
   }
   for(i = 0; i < 10; i++)
      printf("arr[%d] = % d\n", i , i[arr]);/* i[arr]
same as an[i] */
   }
```

Output:

```
Enter 1 value: 1
Enter 2 value: 2
Enter 3 value: 3
Enter 4 value : 4
Enter 5 value: 5
Enter 6 value: 6
Enter 7 value: 7
Enter 8 value: 8
Enter 9 value: 9
Enter 10 value: 10
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
arr[6] = 7
arr[7] = 8
arr[8] = 9
arr[9] = 10
```

Example 3.9:

```
/* Program to sort in the ascending order */
#include<stdio.h>
        main( )
        {
```

NOTES

```
                int sort [10], i, j, temp;
                printf("Enter numbers for array sort
\n\n");
                for (i = 0; i < 1 –; i++)
                {
                      printf("sort [%d] =", i);
                      scanf("%d", & sort[i]);
             }
             /* loop for sorting the numbers */
             for (i = 0; i < 10; i++)
          {
             for (i = 0, i < 10; j++)
          {
                      if(sort[i] > sort[j])
                      {
                                  temp = sort[i];
                                  sort[i] = sort[j];
                                  sort[j] = temp;
                      }
             }
          }
                printf("After sorting the values \n");
                for(i = 0; i < 10; j++)
                      printf("sort[%d]  =  %d\n",  i,
sort[i]);
             }
```

Output:

Enter numbers for array sort

```
Sort[0] = 455
Sort[1] = 344
Sort[2] = 5
Sort[3] = 34
Sort[4] = 56
Sort[5] = 345
Sort[6] = 324
Sort[7] = 56
Sort[8] = 45
Sort[9] = 56
```

After sorting the values

```
Sort [0] = 5
Sort[1] = 34
Sort[2] = 45
Sort[3] = 56
Sort[4] = 56
Sort[5] = 56
Sort[6] = 324
Sort[7] = 344
Sort[8] = 345
Sort[9] = 455
```

NOTES

Multi-Dimensional Arrays

Definition

An array having more than one dimension is a multi-dimensional array.

In C, you can declare an array of arrays called multi-dimensional as follows:

Declaration of multi-dimension arrays

Syntax:

```
<type> <variable_name>[index1][index2]...;
```

Example 3.10:

```
int a[3][3];
char a[10][3][4];
.....
```

An array consists of a variable name with a list of bracketed constant expressions. The function of an index in brackets is to define the number of elements in a given dimension. There are two bracketed expressions in the two-dimensional arrays, three bracketed expressions in the three-dimensional arrays, and so on.

A **double-dimensional/two-dimensional** array can be contemplated as an array of the single-dimensional arrays.

Declaration of Double-dimension arrays:

Syntax:

```
<type> <variable_name>[index1][index2];
```

Example 3.11:

```
int a[3][3];
char a[10][3];
.....
```

Two-dimensional array

Definition

A two-dimensional array is a grid containing rows and columns in which the element is uniquely specified by a row and a column.

NOTES

Syntax:

Datatype array_name[row_size] [column_size];

```
int b[2][2];
```

- This example 3 represents a two-dimensional array with two rows and two columns. The number 2 stored in the element of array, is designated as row 1, column 2.
- When dealing with single-dimensional arrays, we specify a single subscript to locate a specific element.
- Elements of double-dimensional arrays are located by means of row and column dimensions. Two subscripts are required.

Example 3.12:

```
/* Program to illustrate double dimensional array */
#include<stdio.h>
main()
{
   int dob[2][2]; //  Double-dimensional  array
declaring
   /* First 2 indicate number of rows, second 2
indicate number                          o      f
columns */
   dob[0][0] = 6;      /* Assigning array elements
*/
   dob[0][1] = 7;
   dob[1][0] = 8;
   dob[1][1] = 10;
   printf("\n\n The elements of the double-
dimensional array \n\n"):
   printf("dob [0][0] = %d\n", dob[0][0]);
   printf("dob [0][1] = %d\n", dob[0][1]);
   printf("dob [1][0] = %d\n", dob[0][1]);
   printf("dob [1][1] = %d\n", dob [1][1];
}
```

Output:

The elements of the double-dimensional array

```
dob [0] [0] = 6
```

```
dob [0] [1] = 7
dob [1] [0] = 8
dob [1] [1] = 10
```

NOTES

Two-dimensional array initialization: A double-dimensional array is initialized in the same way as a single-dimensional array.

Example 3.13:

```
a[2][2] =   {
            {23, 34},
            {45, 55}
      };
or
a[2][2] = {23, 34, 45, 55};
```

It is important to remember that while initializing a double-dimensional array, it is necessary to mention the column dimension, whereas row dimension is optional.

Comparison of single-dimensional and multi-dimensional arrays

- A single-dimensional array has only one subscript used to refer to the elements in that array, whereas in multi-dimensional arrays, more than one subscript is used to identify the elements in an array.
- A single-dimensional array is a collection of similar type data elements, whereas a multi-dimensional array may be contemplated as an array of single-dimensional arrays.
- In single-dimensional array, elements are individual data items.
- In multi-dimensional array, elements are itself arrays.

Memory: Arrays with more than a single dimension are considered as multi-dimensional arrays. In the memory, multi-dimensional arrays are handled as follows:

A two-dimensional character array with dimensions 2, 2 (2 rows and 2 columns) requires 2*2, 4 bytes. If the array held 2-byte integers, 8 bytes would be required. If the array held doubles (assuming 8 bytes per character), 32 bytes would be required.

A three-dimensional character array with dimensions 2, 2, 2 requires 2*2*2, 8 bytes. If the array held 2-byte integers, 16 bytes would be required. If the array held doubles (assuming 8 bytes per character), 64 bytes would be required.

A four-dimensional character array with dimensions 2, 2, 2, 2 requires 2*2*2*2, 16 bytes. If the array held 2-byte integers, 32 bytes would be required. If the array held doubles (assuming 8 bytes per character), 128 bytes would be required.

In multi-dimensional arrays, it takes the computer time to compute each index. This means that accessing an element in a multi-dimensional array can be slower than accessing an element in a single-dimensional array.

NOTES

Characters Arrays (strings)

Arrays are not confined to numeric elements; however, they may also consist of character or strings.

In 'C', a string is internally stored as an array of characters. As a string may be of any length, the end of the string is indicated by the single character '\0', i.e. the null character with ASCII value 0.

As in the Example 3.13, each box represents a memory location containing one character. Each character of a string is assigned to each element of array.

Example 3.14:

```
#include <stdio.h>
main()
{
   char name [20];
   printf{"Enter Your Name");
   scanf("%s", name);
   printf("hello%s", name);
}
```

Output:

Enter your name = Parthiv

Hello Parthiv

Example 3.15:

```
/* Program to illustrate strings in arrays */
#include <stdio.h>
main()
{
   char name [30];
   int i = 0;
   gets(name);          /* Accepting name */
   while(name[i]! = '\0')
   /* \0' Null character, every string will be
terminated with '0'*/
   {
      putchar(name[i]);
      i ++;
```

```
    }
    printf("\n The length of the string = %d", i);
}
```

Passing arrays to functions

NOTES

Array elements can be passed to a function by calling the function by value or by reference. In the call by value, we pass the values of array elements to the functions, whereas in the call by reference, we pass the addresses of array elements to the functions.

Example 3.16:

```
    /* Program to explain how to pass an array to function
*/
    #include<stdio.h>
    void accept(int []);
    void display(int []);
    main ()
    {
       int arr[6];
       accept (arr);   /* Passing array arr to function accept
*/
       display (arr); /* Passing array arr to function
display */
    }
    void accept (int a[]) /* will holds array arr */
    {
       int i ;
       printf ("\n\n Enter values for matrix arr\n\n");
       for (i = 0 ; i < 6; i ++)
       {
          printf("arr[%d] =", i);
          scanf("%d", & a[i]);
       }}
    void display (int a []) /* a [] will holds array arr
*/
    {
       int i;
       printf("\n\n Values of matrix arr \n\n");
       for (i = 0; i < 6; i ++)
          printf("arr[%d] = % d\n", i, a[i]);
       }
```

NOTES

Output:

Enter values for matrix

```
arr [0] = 5
arr [1] = 8
arr [2] = 5
arr [3] = 7
arr [4] = 65
arr [5] = 54
```

values of matrix arr

```
arr [0] = 5
arr [1] = 8
arr [2] = 5
arr [3] = 7
arr [4] = 65
arr [5] = 54
/* Additional programs on Arrays */
```

Addition and Subtraction

Consider two matrices A and B.

Then their addition produces resultant matrix C as

The following algorithm describes the matrix addition procedure:

Algorithm

Step 1: start

Step 2: initialize the variables a, b, c, i, j;

Step 3: enter the first matrix

Step 4: for i = 0, i < 3, j++ is the condition satisfies go to the next step, else go to step 5

4.1: for j = 0, j < 3, j++ is the condition satisfies go to the next step

4.2: read a value

Step 5: enter the second matrix

Step 6: for i = 0, i < 3, i++ go to the next step, else go to step7

6.1: for j = 0, j < 3, j++ go to 6.2

6.2: read b

Step 7: print addition of matrix

Step 8: for(i = 0, i < 3, i++) go to 8.1, else go to step 9

8.1: for j = 0, j < 3, j++ go to 8.2

8.2: c[i][j] = a[i][j] + b[i][j]

8.3: print c

Step 9: stop

NOTES

Example 3.17:

Write a 'C' program to perform addition of two matrices of the order of M×N

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
    int a[3][3], b[3][3], c[3][3], i, j;
    clrscr();
    printf("enter the elements of first matrix");
    for (i = 0; i < 3; i++)
    {
       for(j = 0; j < 3; j++)
          {
             scanf("%d",&a[I][j]);
          }
    }
    printf("enter the elements of second matrix");
    for(I = 0; I < 3; I++)
    {
       for(j = 0; j < 3; j++)
    {
    c[I][j] = a[I][j] + b[I][j];
    printf("%3d\t", c[I][j]);
    }
    printf("\n");
    }
    getch();
    }
```

Input:

Enter the elements of the first matrix

1 2 3

4 5 6

7 8 9

NOTES

Enter the second matrix

1 2 3

4 5 6

7 8 9

Output:

2 4 6

8 10 12

4 16 18

Subtraction

Consider two matrices A and B.

Then their subtraction produces resultant matrix C as

The following algorithm describes the matrix subtraction procedure:

Algorithm:

Step 1: start

Step 2: initialize the variables a, b, c, i, j;

Step 3: enter the first matrix

Step 4: for i = 0, i < 3, j++ is the condition satisfies go to the next step, else go to step 5

4.1: for j = 0, j < 3, j++ is the condition satisfies go to the next step

4.2: read a value

Step 5: enter the second matrix

Step 6: for i = 0, i < 3, i++ go to the next step, else go to step 7

6.1: for j = 0, j < 3, j++ go to 6.2

6.2: read b

Step 7: print subtraction of matrix

Step 8: for(i = 0, i < 3, i++) go to 8.1 else go to step 9

8.1: for j = 0, j < 3, j++ go to 8.2

8.2: c[i][j] = a[i][j] " b[i][j]

8.3: print c

Step 9: stop

Example 3.18:

Write a 'C' program to perform subtraction of two matrices of the order of M×N

```
/*----------STARTING THE PROGRAM----------*/
    #include<stdio.h>
    #include<stdlib.h>
    void main()
    {
     int a[10][10], b[10][10], c[10][10];
     int m, n, k, i, j, l;
     do
     {
        printf("Enter value for m <= 10:");
        scanf("%d",&m);
     }while(m > 10);
     do
       {
        printf("Enter value for n <= 10:");
        scanf("%d",&n);
        }while(n > 10);
     for(i = 0; i < m; ++i)
     for(j = 0; j < n; ++j)
     {
        printf("Enter a value for a[%d][%d]:",i, j);
        scanf("%d",&a[i][j]);
       }
     for(i = 0; i < m; ++i)
     for(j = 0; j < n; ++j)
     {
        printf("Enter a value for b[%d][%d]:", i, j);
        scanf("%d", &b[i][j]);
     }
     for(i = 0; i < m; ++i)
     for(j = 0; j < n; ++j)
           c[i][j] = a[i][j] - b[i][j];
        for(i = 0; i < m; ++i, printf("\n"))
        for(j = 0; j < n; ++j)
           printf("%d ", c[i][j]);
    }
```

Output:

```
Enter value for m <= 10:2
Enter value for n <= 10:2
Enter a value for a[0][0]:1
```

NOTES

NOTES

```
Enter a value for a[0][1]:2
Enter a value for a[1][0]:3
Enter a value for a[1][1]:4
Enter a value for b[0][0]:0
Enter a value for b[0][1]:1
Enter a value for b[1][0]:2
Enter a value for b[1][1]:3
1 1
1 1
```

Explanation:

The subtraction of two matrices of the same order is again a matrix of the same type obtained by the subtraction of the corresponding elements of two matrices.

Multiplication of Arrays

Suppose

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{bmatrix}$$

The product matrix *AB* is a 2 × 3 matrix. The elements in the first row of *AB* are obtained, respectively, by multiplying the first row of *A* by each of the columns of *B:*

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{bmatrix} = \begin{bmatrix} 1.2+3.3 & 1.0+3.2 & 1.(-4)+3.6 \\ & & \end{bmatrix} = \begin{bmatrix} 11 & 6 & 14 \\ & & \end{bmatrix}$$

Similarly, the elements in the second row of *AB* are obtained, respectively, by multiplying the second row of *A* by each of the columns of *B:*

$$= \begin{bmatrix} 11 & 6 & 14 \\ 2.2+4.3 & 2.0+4.2 & 2.(4)+4.6 \end{bmatrix} = \begin{bmatrix} 11 & 6 & 14 \\ 16 & 8 & 16 \end{bmatrix}$$

That is, $$AB = \begin{bmatrix} 11 & 6 & 14 \\ 16 & 8 & 16 \end{bmatrix}$$

The following algorithm finds the product *AB* of matrices *A* and *B*, which are stored as two-dimensional arrays:

Algorithm:

(Matrix Multiplication) MATMUL(A, B, C, M, P, N)

Let A be an M × P matrix array, and let B be a P × N matrix array. This algorithm stores the product of A and B in an M × N matrix array C.

1. Repeat Steps 2 to 4 for I = 1 to M:
2. Repeat Steps 3 and 4 for J = 1 to N:

3. Set C[I, J] : = 0. [Initializes C[I, J].]
4. Repeat for K = 1 to P:
   C[I, J]:=C[I, J] + A[I, K]*B[K, J]
   [End of inner loop.]
   [End of Step 2 middle loop.]
   [End of Step 1 outer loop.]
5. Exit.

NOTES

The complexity of a matrix multiplication algorithm is measured by counting the number C of multiplications. The reason that additions are not counted in such algorithms is that computer multiplication takes much more time than computer addition. The complexity of the above algorithm is equal to

$$C = m \cdot n \cdot p$$

This comes from the fact that Step 4, which contains the only multiplication is executed $m \cdot n \cdot p$ times. Extensive research has been done on finding algorithms for matrix multiplication, which minimize the number of multiplications. The next example gives an important and surprising result in this area.

Example 3.19:

Write a program for multiplication of matrices of form m × n, n × k resulting m × k matrix

```
#include<stdio.h>
#include<stdlib.h>
void main()
{  int a[10][10], b[10][10], c[10][10];
    int m, n, k, i, j, l;
    do
    {
       printf("Enter value for m <= 10:");
       scanf("%d", &m);
    }while(m > 10);
    do
    {
       printf("Enter value for n <= 10:");
       scanf("%d", &n);
    }while(n > 10);
 do
    {
       printf("Enter value for k <= 10:");
       scanf("%d", &k);
    }while(n > 10);
```

NOTES

```
        for(i = 0; i < m; ++i)
        for(j = 0; j < n; ++j)
        {
           printf("Enter a value for a[%d][%d]:", i, j);
           scanf("%d", &a[i][j]);
        }
        for(i = 0; i < n; ++i)
        for(j = 0; j < k; ++j)
        {
           printf("Enter a value for b[%d][%d]:", i, j);
           scanf("%d", &b[i][j]);
        }
        for(i = 0; i < m; ++i)
        {
          for(j = 0; j < k; ++j)
          {
          c[i][j] = 0;
           for(l = 0; l < n; ++l)
              c[i][j] += a[i][l] * b[l][j];
           }
        }
       for(i = 0; i < m; ++i, printf("\n"))
        for(j = 0; j < k; ++j)
          printf("%d ", c[i][j]);
    }
```

Output:

```
Enter value for m <= 10:2
Enter value for n <= 10:2
Enter value for k <= 10:3
Enter a value for a[0][0]:1
Enter a value for a[0][1]:2
Enter a value for a[1][0]:3
Enter a value for a[1][1]:4
Enter a value for b[0][0]:5
Enter a value for b[0][1]:6
Enter a value for b[0][2]:7
Enter a value for b[1][0]:8
Enter a value for b[1][1]:9
Enter a value for b[1][2]:2
   21 24 11
   47 54 29
```

Explanation:

The product of two matrices is defined only when the number of columns of first is equal to the number of rows of second. If **A** and **B** are two matrices of order $m \times n$ and $n \times p$, respectively, then the product **AB** will be of order $m \times p$.

NOTES

Thus if $A=[a_{ij}]_{mxn}$ and $B=[b_{ij}]_{nxp}$

then $AB=[c_{ij}]_{mxp}$ where $c_{ij}=\sum_{r=1}^{n} a_{ir} b_{rj}, 1 \le i \le m$ and $1 \le j \le p$

Sorting and Printing Techniques

Let A be a list of *n* numbers. *Sorting* A refers to the operation of rearranging the elements of A so they are in the increasing order, i.e. so that

```
A[1] < A[2] < A[3] < … < A[N]
```

For example, suppose A originally is the list

```
8, 4, 19, 2, 7, 13, 5, 16
```

After sorting, A is the list

```
2, 4, 5, 7, 8, 13, 16, 19
```

Sorting may seem to be a trivial task. Actually, sorting efficiently may be quite complicated. In fact, there are many, many different sorting algorithms. Here we present and discuss a very simple sorting algorithm known as the *quick sort*.

At each step, a particular element is placed in its final position within the list, i.e. at each step an element is placed in its correct position in an array with the following constraints:

- All the elements, which precede this element, have smaller keys/values
- All the elements that follow it have larger keys.

This technique partitions the list into two sub-lists. The same process can in turn be applied to each of these sub-lists and repeat until all elements are placed in their final position. It performs well on larger list of elements.

Procedure Quick Sort (K, LB, UB) or portion exchange sort:

K	Array of n elements
LB	Lower bound of the current sub-list
UB	Upper bound of the correct sub-list
I	Index variable
J	Index variable
KEY	Key value
FLAG	Logical Variable

```
1. [Initialize]
   FLAG ← True
```

NOTES

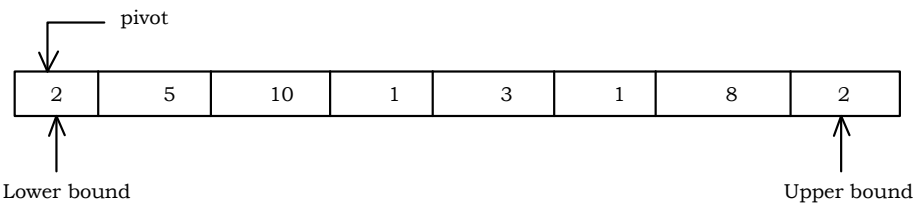
```
2. [Perform Sort]
   If LB < UB then
      I ← LB
      J ← UB+1
      KEY ← K[LB]
      Repeat while [FLAG]
      I ← I+1
      Repeat while [I] < KEY
         I ← I+1
         J ← J+1
      Repeat while K[J] > KEY
         J ← J - 1
      if I < J then
         K[I] ↔ K[J]
      (interchange elements)
      else FLAG ← False
         K [LB] ↔ K
      Call Quick_Sort (K,LB, J - 1)
      Call Quick_Sort (K, J+1, OB)
3. [Finished]
   Return
Example
```

Unsorted Array	:	42	23	36	42	65	58	94	36	99	87
Passers											
1.	:	{11	23	36}	42	{65	58	94	74	99	87}
2.	:	11	{23	36}	42	{65	58	94	74	99	87}
3.	:	11	23	{36}	42	{65	58	94	74	99	87}
4.	:	11	23	36	42	{58}	65	{94	74	99	87}
5.	:	11	23	36	42	58	65	{94	74	99	87
6.	:	11	23	36	42	58	65	{87	74}	94	{99}
7.	:	11	23	36	42	58	65	{74}	87	94	{99}
8.	:	11	23	36	42	58	65	74	87	99	{99}
9.	:	11	23	36	42	58	65	74	87	94	99

Sorted

Describe Quick sort algorithm for the input: 20, 5, 100, 15, 30, 10, 80, 25

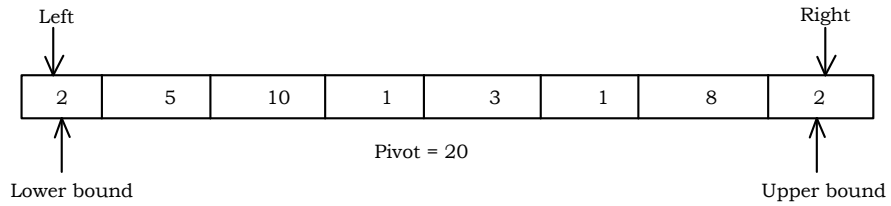
1. **Choosing a pivot:** You can select any item among the given list of elements as a pivot, choose the leftmost one to avoid confusion.



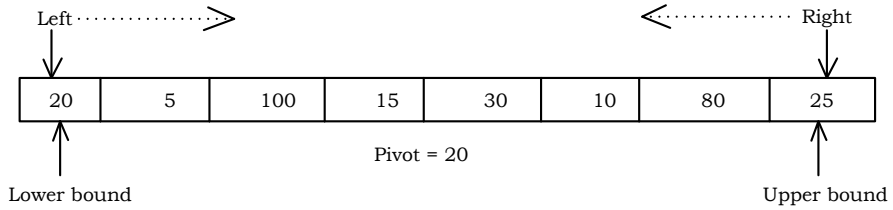
2. Arrange all elements to the left that are less to the pivot (=20) and arrange all elements to the right that are greater to it.

NOTES

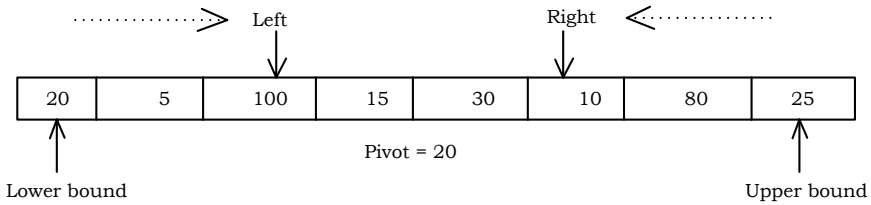
Set the left and right markers as shown here:



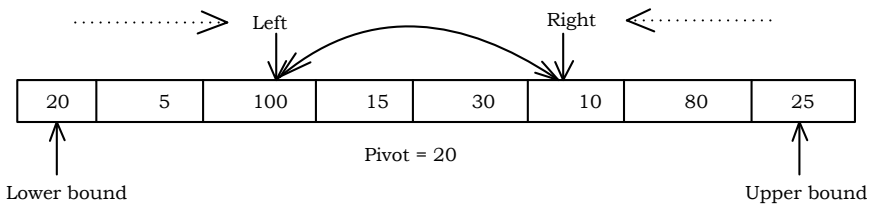
- Move the markers until they cross over



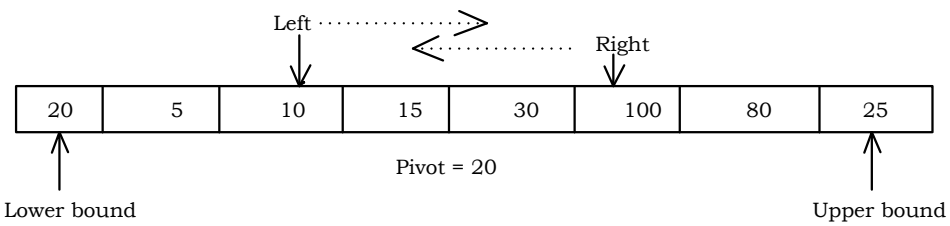
- Move the left pointer while it points to elements, which are less than or equal to the pivot and move right, which are greater than or equal to the pivot



- Swap the two elements (left and right) on the wrong side of the pivot

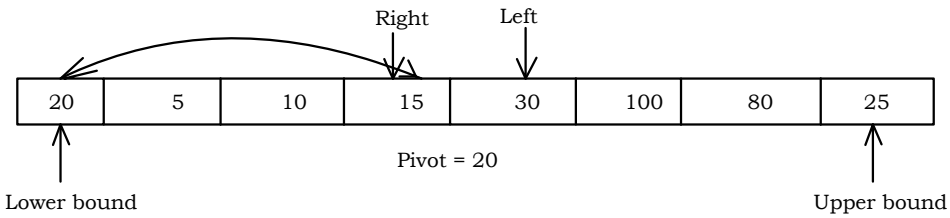


- Again continue the same process until Right > Left

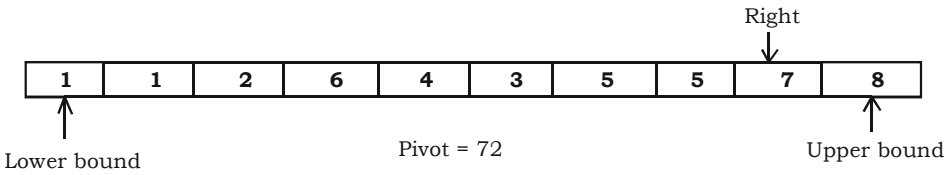


NOTES

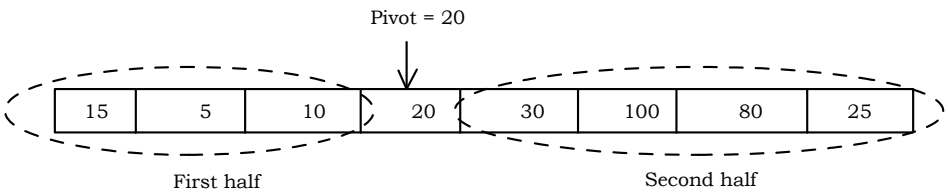
- Finally, swap the pivot and right



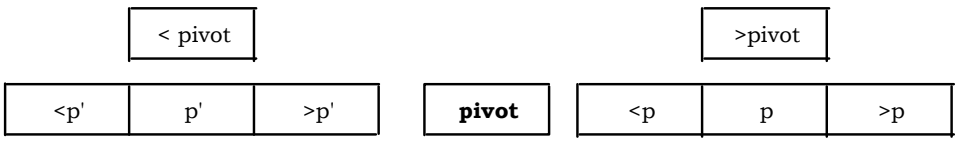
- Now return the position of the pivot



- Now the two halves are divided by the pivot. Apply the same procedure to each half.



- Continue this process for the Left half portion and the Right half portion



'C' program that uses a function to sort an array of integers

```
# include <stdio.h>
# include <Conio.h>
main ( )
{  int a[10], n ;
   printf ("Enter size of array:");
   scanf("%d", & n);
   printf ("Enter elements for array");
   for (i = 0; i < n; i++)
   scanf ("%d", & a[i]);
   Sort (a, h);
   getch ( );
}
   Sort (a, n)
   int a[ ], n;
   {  int i, j, temp;
```

```
        for (i = 0; k < n – 1; i++)
           If (a[j – 1] > a[j])
           {  temp = a [j – 1]
              a[j – 1] = a[j];
              a [j] = temp;
           }
        for (i = o; i < n; i++)
        printf ("%d", a[i]);
     }
```

Time complexity of quick sort:

The analysis of the Quick Sort is given by $T_{QS}(N) = P(N) + T_{QS}(J - L) + T_{QS}(U - J)$

Where P(N), $T_{QS}(J - L)$, and $T_{QS}(U - J)$ denote the times to partition the given table, sort the left portion, and sort the right portion of the array, respectively. the time to partition a array is O(N).

The Worst case time analysis, assuming J = L, then becomes

$$T_{QS}^{W}(N) = P(N) + T_{QS}^{W}(0) + T_{QS}^{W}(N - 1) = C * N + T_{QS}^{W}(N - 1)$$

=

$$C * N + C * (N - 1) + T_{QS}^{W}(N - 2) = C * N + C * (N - 1) + C * (N - 2) + T_{QS}^{W}(N - 3)$$

$$\sum_{k=1}^{N} c * k + T_{QS}^{W}(0) = C * \frac{(N + 1)(N)}{2} = O(N^2)$$

Sometimes, we can avoid the worst case by choosing more carefully the element for the final placement at each stage. Instead of always choosing lower bound as PIVOT, we could choose an arbitrary value in the interval [L, U] or we can select the middle element as PIVOT, i.e. [(L + U)/2].

The Best case analysis occurs when the array is always partitioned in half.

$$T_{QS}^{B}(N) = P(N) + 2T_{QS}^{B}(N/2) = c * N + 2T_{QS}^{B}(N/2) = c * N + 2c(N/2) + 4T_{QS}^{B}(N/4)$$

$$= c * N + 2c(N/2) + 4c(N/4) + 8\,T_{QS}^{B}(N/8)\; T_{QS}^{B} = 3 * C * N + 8T_{QS}^{B}(N/8)$$

$$= (\log_2 N) * c * N + 2^{\log_2 N} * T_{QS}^{B}(1) = O(N \log_2 N)$$

$\therefore$ The average and best cases for quick sort is O(nlogn)

Write an algorithm or a C program for quick sort

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
#define MAXCOLS 10
typedef int VECTOR[MAXCOLS];
```

NOTES

```
void quick_sort(VECTOR a, int lb, int ub)
{ int temp, key, i, j, flag = 1;
 if(lb < ub)
 { i = lb;
 J = ub + 1;
 key = a[lb];
 while(flag)
 {i = i + 1;
 while(a[i] < key)
 i = i + 1;
 j = j – 1;
 while(a[j] > key)
 j = j – 1;
 if(i < j)
    {temp = a[i];
 a[i] = a[j];
 a[j] = temp;
 }
 else
 flag = 0;
 }
 Temp = a[lb];
 a[lb] = a[j];
 a[j] = temp;
 quick_sort(a, lb, j – 1);
 quick_sort(a, j + 1, ub);
 }
}
void main()
{
 VECTOR A = {5, 4, 3, 2, 1};
 int i;
 quick_sort(a, 0, 4);
 for(i = 0; i < 5; ++i)
 printf("%d", a[i]);
}
```

Single operations that involve all arrays are not allowed in 'C'. if 'a' and 'b' are similar arrays (same data type, same size). Assignment operations and comparison operations should necessarily be executed on the element-by-element basis. This is completed within the loop that is used to process one array element. The number

of passes through the loop will be equal to the number of array elements to be processed.

Example 3.20:

NOTES

Accepting 'n' elements for one-dimensional array

```
main()
{
      int a[10], i=0, n;
      printf("enter size of array");
      scanf  ("%d", &n);
      for   (i=0; i< n; i++)
      scanf("%d", &a[i]);
}
```

Printing the array elements on the screen

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

Accepting elements and displaying the elements for two-dimensional array:

```
scanf("%d%d", &m, &n);
for(i = 0; i < n; i+ +)
   for(j = 0; j < n; j++)
      scanf("%d", &a[i[j]);
```

3.19 FUNCTIONS

Modular Programming Overview

You have been writing small programs in 'C' to illustrate the fundamental concepts of data structures and algorithms. In practice, you have to write programs to solve real-life situations taking into account all the features of the language. In fact, 'C' is used for developing operating systems and other system software. Such programs should be free from logical errors and program or code errors. In the olden days, people wrote larger programs sequentially using all the constructs of a language. This led to difficulties in debugging and even understanding. As no technique for

NOTES

good programming was available, an effective working program was suitable enough. However, this led to a software crisis in the sixties. It was then that experts felt methodologies should be evolved to improve the quality of programs. One of the recommendations was modular programming. A module can be a single function or a group of related functions carrying out a specific task. Since a programmer will not be able to comprehend the meaning of a program of more than 100 lines, one of the features of modular programming is to divide the program into smaller modules or functions, which will execute a particular task. A program consists of a number of modules properly linked. This enables the programmer to analyse each such small block and then draw the bigger picture containing all the blocks to know exactly what the program is doing. If a program is developed in this modular way, it will become easy to divide and conquer the problem. This is one of the features of structured programming.

The main objective of structured programming is to plan and develop a program as a collection of modules. 'C' language was developed for achieving this objective easily. Therefore, structured programming also means that the execution of statements progresses linearly. There is no back and forth traversal while executing the program. The program uses a single entry–single exit pattern. The `label` and `goto` statements affect linear programming and hence, are discouraged. The `while, for, do-while, switch`, etc. statements enable structured programming. These loop statements do a particular predefined task and hence, can be independently checked for their correctness. Structured programming results in understandable programs.

'C' contains a number of library functions. You have already used many of them. They are as follows:

```
printf( )
scanf( )
getchar( )
putchar( )
gets( )
puts( )
strlen( )
```

In order to use the library functions, you must include the header files supplied with the system such as `stdio.h, string.h`, etc. In effect, the library functions are hiding the programs or in other words, they are encapsulated. You only pass arguments to use them. The arguments are, for example, what you put within brackets following `printf(), scanf(), gets(), puts()`, etc. The availability of such functions is a feature of modular programming. The availability of these functions has reduced considerably the need to code the print or other statements and check them in every program.

NOTES

In the top-down programming method, the main function calls other specialized functions for carrying out a specialized function such as `strlen( )`. However, for the availability of this function, you would have to write a routine whenever you want to find the length of a string as part of the main function. This would increase the length of the code, affect readability and increase complexity. Therefore, it would be better to follow the modular programming concept. This is illustrated in Figure 3.3:

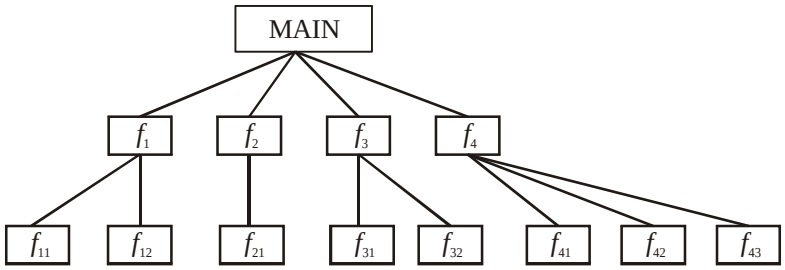


Figure 3.3 *Modular Programming Concept*

Develop the program in such a manner that the main program calls a number of functions for carrying out specific tasks. Each function may, in turn, call other specialized functions wherever required. Such a program will be easy to understand, debug and maintain.

In order to improve the quality of programming, you should, therefore, call functions wherever required. The functions supplied with the system are the library functions. User-developed functions are called user-defined functions.

You now know that each function performs a specified task. Arguments or data are passed to the function and the function performs some specified actions. Before you go into more details, you must understand the usage of functions. Assuming that you want to reverse a number in a program, you split the program into two parts. The `main()` function gets the number to be reversed. It passes the number to a function called `reverse()` whose specialized and only job is to get the number, reverse it and send back the reversed number to the function, which calls it. The diagrammatic representation is as follows:

```
main

reverse
```

The `main()` function passes on a number to reverse whenever it wants to reverse it. The `reverse()` function reverses the number and sends it back to the `main()` function. Thus, the task is perfectly partitioned with perfect understanding and protocol. Later on, if some other function wants to reverse a number, it can bank upon the capability of already tested `reverse()` function and use it.

NOTES

General Form of Function

A function consists of three parts:

- Function prototype
- Function definition
- Function call

Function Prototypes

A function prototype is also called function declaration. A function may be declared at the beginning of the main function. The function declaration is of the following type:

```
return data type  function name (formal argument 1, argument 2, ..... );
```

A function, after execution, may return a value to the function, which called it. It may not return a value at all but may perform some operations instead. It may return an integer, a character or a float. If it returns a float you may declare the function as,

```
float f1(float arg 1, int arg 2);
```

If it does not return any value, you may write the above as,

```
void fun2(float arg1, int arg2);
/*void means nothing*/.
```

If it returns a character, you may write,

```
char fun3(float arg1, int arg2);
```

If no arguments are passed into a function, an empty pair of parentheses must follow the function name. For example,

```
char fun4( );
```

The arguments declared as part of the prototype are also known as formal parameters. The formal arguments indicate the type of data to be transferred from the calling function. This is about the function declaration.

Function Call—Passing Arguments to a Function

You may call a function either directly or indirectly. When you call the function, you pass actual arguments or values. Calling a function is also known as function reference.

There must be a one-to-one correspondence between the formal arguments declared and the actual arguments sent. They should be of the same data type and in the same order.

```
sum = f1 (20.5, 10); fun4( );
```

NOTES

Function Arguments

You know now that an argument is the parameter or value. It could be of any of the valid types such as all forms of integers or a float or char. You come across two types of arguments when you deal with functions:

- Formal arguments
- Actual arguments

The formal arguments are defined in the function declaration in the calling function. What is an actual argument? The data, which is passed from the calling function to the called function, is called actual argument. The actual arguments are passed to the called function through a function call.

Each actual argument supplied by the calling function should correspond to the formal arguments in the same order. The new ANSI standard permits the declaration of the data types within function declaration to be followed by the argument name. You have used only this type of declaration as it will help the students to follow C++ program easily. This helps in understanding one-to-one correspondence between the actual arguments supplied and those received in the function, and facilitates the compiler to verify that one-to-one correspondence exists and that the right number of parameters have been passed. It may be noted that the formal arguments cannot be used for any other purposes. It only gives a prototype for the function. Thus, the names of the formal arguments are dummy and will not be recognized elsewhere, even in the function in which it is defined.

Although the types of variables in the function declaration, also known as prototype, and function call are same, the names need not be the same. You have already used this concept in Example 3.23 after defining `float a` and `float b` in the functions prototype; you first called `add (a, b)`, `add (c, d)` and then `add (sum1, sum2)`. Thus the formal arguments defined in the prototype and the actual arguments were not the same in two of the above cases.

When the actual arguments are passed to a function, the function notes the order in which they are received and appropriately stores them in different locations. You must note that even if you use `a` and `b` in the `add` function, they will be stored in different locations. They will have no relationship with `a` and `b` of the main function. Therefore, even if `a` and `b` are assigned different values in the called function, the corresponding values in the calling function would not have changed.

Function Definition

The function definition can be written anywhere in the file with a proper declarator followed by the declaration of local variables and statements. The function definition consists of two parts, namely function declarator or heading and function functions. The function heading is similar to function declaration, but will *not* terminate with a semicolon.

The use of functions will be demonstrated with simple programs in this unit.

Assume that you wish to get two integers. Pass them to a function `add`.

NOTES

Add them in the add function. Return the value to the main function and print it. The algorithm for solving the problem is as follows:

Main Function

```
Step 1:   define function add
Step 2:   get 2 integers
Step 3:   call add and pass the 2 values
Step 4:   get the sum
Step 5:   print the value
          function add
Step 1:   get the value
Step 2:   add them
Step 3:   return the value to main
```

Thus you have divided the problem. The program is given below:

Example 3.21:

```
/* use of function*/
#include <stdio.h>
int main()
{
    int a=0, b=0, sum=0;
    int add(int a, int b); /*function declaration*/
    printf("enter 2 integers\n");
    scanf("%d%d", &a, &b);
    sum =add(a, b);   /*function call*/
    printf("sum of %d and %d =%d", a, b, sum);
}
/*function definition*/
int add (int c, int d)   /*function declarator*/
{
    int e;
    e= c+d;
    return e;
}
```

Result of the program

```
enter 2 integers
6667  4445
sum of 6667 and 4445 =11112
```

The explanation as to how the program works is given below:

On the fifth statement (seventh line), the declaration of the function add is given. Note that the function will return an integer. Hence, the return type is defined as int. The formal arguments are defined as int a and int b. The function

NOTES

name is `add`. You cannot use a variable without declaring it as also a function without telling the compiler about it. Note also that function declaration ends with a semicolon, similar to the declaration of any other variable. Function declaration should appear at the beginning of the calling function. It hints to the compiler that the function is going to call the function `add` later in the program. If the calling function is not going to pass any arguments, then empty parentheses are to be written after the function name. The parentheses must be present in the function declaration. This happens when the function is called to perform an operation without passing arguments. In this case, if `a` and `b` are part of the *called* function `add` itself, then we need not pass any parameters. In such a case, the function declaration will be as follows assuming that the called function returns an integer:

```
int add  ( ) ;
```

In Example 3.21, you get the values of `a` and `b`. After that you call the function `add` and assign the value returned by the function to an already defined `int` variable `sum` as given below:

```
sum  =  add ( a , b );
```

Note that `add(a, b)` is the function call or function reference. Here, the return type is not to be given. The type of the arguments are also not to be given. It is a simple statement without all the elements of the function declaration. However, the function name and the names of the arguments passed, if any, should be present in the function call. When the program sees a function reference or function call, it looks for and calls the function and transfers the arguments.

The function definition consists of two parts, i.e., the function declarator and function body.

The function declarator is a replica of the function declaration. The only difference is that the declaration in the calling function will end with a semicolon and the declarator in the called function will not end with a semicolon. As in `main()`, the entire functions body will be enclosed within braces. The whole function can be assumed to be one program statement. This means, that all the statements within the body will be executed one after another before the program execution returns to the place in the main function from where it was called.

The important points to be noted are:

- The declarator must agree totally with the declaration in the *called* function, i.e., the return data type, the function name, the argument type should all appear in the same order. The declarator will not end with a semicolon.
- You can also give the same name as in the calling function—in the declaration statement or the function call—or different names to the arguments in the function declarator. Here, you have given the names `c` and `d`. What is important, however, is that the type of arguments should appear as it is in the declaration in the calling program. They must also appear in the same order.

- At the time of execution, when the function encounters the closing brace }, it returns control to the calling program and returns to the same place at which the function was called.

NOTES

In this program, you have a specific statement `return (e)` before the closing brace. Therefore, the program will go back to the main function with value of `e`. This value will be substituted as,

```
sum = (returned value)
```

Therefore, `sum` gets the value, which is printed in the next statement. This is how the function works.

Assume now that the program gets `a` and `b` values, gets their `sum1`, gets `c` and `d` and gets their `sum2` and then both the sums are passed to the function to get their total. The program for doing this is as follows:

Example 3.22:

```
/* A function called many times */
#include <stdio.h>
int main()
{
    float a, b, c, d, sum1, sum2, sum3;
    float add(float a, float b); /*function declaration*/
    printf("enter 2 float numbers\n");
    scanf("%f%f", &a, &b);
    sum1 =add(a, b);   /*function call*/
    printf("enter 2 more float numbers\n");
    scanf("%f%f", &c, &d);
    sum2 =add(c, d);   /*function call*/
    sum3 =add(sum1, sum2);   /*function call*/
    printf("sum of %f and %f =%f\n", a, b, sum1);
    printf("sum of %f and %f =%f\n", c, d, sum2);
    printf("sum of %f and %f =%f\n", sum1,sum2, sum3);
}
/*function definition*/
float add (float c, float d)   /*function declarator*/
{
    float e;
    e = c+d;
    return e;
}
```

Result of the program

```
enter 2 float numbers
1.5  3.7
```

```
enter 2 more float numbers
5.6    8.9
sum of 1.500000 and 3.700000 =5.200000
sum of 5.600000 and 8.900000 =14.500000
sum of 5.200000 and 14.500000 =19.70000
```

NOTES

You have defined sum1, sum2 and sum3 as float variables.

You are calling function add three times with the following assignment statements:

```
sum1 = add( a, b );
sum2 = add( c, d);
sum3 = add( sum1 , sum2 );
```

Thus the program goes back and forth between main and add as follows:

```
main()
add (a, b)
main()
add (c, d)
main()
add (sum 1, sum 2)
main()
```

Had you not used the function add, you would have to write statements pertaining to add 3 times in the main program. Such a program would be large and difficult to read. In this method you have to code for add only once and hence, the program size is small. This is one of the reasons for the usage of functions.

In Example 3.22, We could add another function call by add (10.005, 3.1125); This statement will also work perfectly. After the function is executed, the sum will be returned to the main function. Therefore, both variables and constants can be passed to a function by making use of the same function declaration.

You have seen a program, which calls a function thrice. Now a problem, which calls three functions will be discussed.

Problem

The user gives a four-digit number. If the number is odd, then the number has to be reversed. If it is even, then the number is to be doubled. If it is evenly divisible by three, then the digits are to be added. Now write the algorithm for solving the problem.

Step 1: Get the number.

Step 2: If number is odd, call the reverse function.

Step 3: Else multiply the number by 2 and hence, call multiply.

Step 4: If number is evenly divisible by 3, call add-digits.

These are the steps. The first one, namely writing a function to multiply by 2 is simple. You will look at the other two steps now.

Reverse

NOTES

You have already seen reversing the number in a program. You will use the same steps.

Step 1: `reverse = 0`

Step 2:
```
while ( number > 0 )
      reverse = reverse * 10 +  (number % 10 )
      number = number/10
```

Step 3: `return (reverse)`

ADD digits function

Step 1: `sum = 0`

Step 2:
```
while number > 0
      sum = sum + (number % 10)
      number = number / 10
```

Step 3: `return (sum)`

See how the above algorithm adds digits and works.

Give 4321 as the number.

Step 1: `sum = 0`

Step 2:
```
Iteration 1
sum = 0 +  modulus of  (4321/10)
    = 0 + 1 = 1
number = 4321/10 = 432
Iteration 2
sum = 1 + modulus of (432/10)
     =  1 + 2
```

After 4 iterations,

```
sum =1 + 2 + 3 + 4
```

Step 3: Sum is returned.

The program is given below:

Example 3.23:

```
/*program to demonstrate calling
multiple functions*/
#include<stdio.h>
int main()
{
    long nummul=0;
    long num=0, rev=0, add_digit=0;
       /*good practice to initialize all variables*/
```

NOTES

```
    long reverse(long num);
    long mult(long num);
    int sum_digit(long num);
    printf("enter unsigned number\n");
    scanf("%lu", &num);
    if (num%2)  /*remainder 1*/
    {
       rev = reverse(num);
       printf("number is odd\n");
       printf("number  entered=%lu\n  number
reversed=%lu\n", num, rev);
    }
    else
    {
       nummul=mult(num);
       printf("number is even\n");
       printf("number=%lu\n its multiple=%lu\n", num,
nummul);
    }
    if (num%3 ==0)
    {
       add_digit= sum_digit(num);
       printf("number evenly divisible by 3\n");
       printf("sum of digits =%lu", add_digit);
    }
}
long reverse(long n)
{
    long r=0;
    while (n>0)
    {
       r=r*10+(n%10);
       n=n/10;
    }
    return r;
}
long mult(long p)
{
    long sq;
    sq=2*p;
    return sq;
```

NOTES

```
}
int sum_digit(long num)
{
    long sum=0;
    while (num >0)
    {
       sum=sum+(num%10);
       num=num/10;
    }
    return sum;
}
```

Result of the program

```
enter unsigned number
4321
number is odd
number entered=4321
number reversed=1234
```

Look at the program. After getting the number from the user, it evaluates if remainder of `(num/2)` = true; i.e., if the remainder is `1`, then it is true. If `remainder = 1`, then the number is odd and hence, the reverse function is called. The returned value is assigned to `rev` and printed.

If the number is even, the number is doubled. Since the doubled value may exceed the maximum of the unsigned integer, we have declared it as a long integer.

Next we check if the number is evenly divisible by 3.

If it is so then we add the digits. Thus, the main function of Example 3.23 calls three functions for carrying out specific tasks. All the three functions are supplied with the same arguments, but return different values.

Scope Rules for Function

The scope of the variable is local to the function unless it is a global variable. For example,

```
int     function1(int I )
{ int j=100;
   double  function2 (int j) ;
   function2 (j) ;
}
double function2 (int p)
{   double m;
   return m;
}
```

NOTES

The variable j in function1 is not known to function2. You pass it to function2 through the argument j. This will be assigned as equal to int p. Similarly, m in function2 is not known to function1. It can be made known to function1 through the return statement. This makes the scope rules of variables in function quite clear. The scope of variables is local to the function where defined. However, global variables are accessible by all functions in the program if they are defined above all functions.

Example 3.24:

```
/* to demonstrate that the scope of a
variable is local to the function*/
#include <stdio.h>
int main()
{
    float a=100.250, b=200.50;
    void change (float a, float b);
    change(a, b);
    printf("a= %f b= %f\n ", a, b);
    printf("these are the original values");
}
/*function definition*/
void change (float a, float b)  /*function declarator*/
{
    a +=1000;
    b-=200.5;
}
```

Result of the program

```
a= 100.250000 b= 200.500000
these are the original values
```

You passed a = 100.25 and b = 200.5 to the function. In the function, you modified a as 1100.25 and b as zero. However, when you print a and b in the main function, you get the same old values. This confirms that variables are local to the function unless otherwise specified.

Notice that in the calling function the type declaration of formal parameters is symbolic and used only to indicate the format. You will notice, for instance in Example 3.21, that the int a has been declared and assigned a value of 0. This has no relationship with int a in the function declaration. You could even omit the variable name and declare as int *add*(int, int). It will still work. Here a and b have been given for better readability.

This is the reverse in the case of a called function. In the same program, int c and int d are explicitly defined in function add in the declarator. The variables are used further in the function add. This is not the case with the variables

NOTES

in the declaration statement or prototype of the calling function, which will never be used further.

This method of invoking a function is called call by value, i.e., you call the functions with values as arguments.

Library Functions

You have used the `scanf()` function, which is called a library function. We have only invoked the function or called the function, but where is function declaration and function definition? We have no need to write them. That is the advantage of library functions and the 'C' language. The declaration of `scanf()` and `printf()` are in `stdio.h`. Thus all declarations required for the library functions are in the header files. The function definitions are in separate functions with `.lib` extensions. However, where are the formal arguments? The formal arguments pertaining to the declaration will be in the header files. The arguments pertaining to the declarators will be in the respective library functions. The actual arguments are enclosed within the function reference or function call; for example,

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

Whatever is specified within the parentheses are the actual arguments. We specify the actual arguments to match the function prototype specified in the header files and get the job done very easily using the library functions.

Return

Return Values

The return data type is declared in the function declaration in the main function or the calling function, and the declarator is indicated in the first line of the function definition. If no value is to be returned, the return data type `void` is specified. `Void` simply mean NULL or nothing. Therefore, it does not fall in any other data types such as `int` or `float` or `char`.

The return value as we have seen is the result of computation in the called function. You return a value, which is stored in a data type in the called function. The return value means that the value thus stored in the called function is assigned or copied to a variable in the main or calling function. Therefore, to receive the result, a data type should have been declared and preferably initialized in the calling function.

The return statement can be any of the following types:

```
return (sum) ;
return V1;
return  "true" ;
return 'Z' ;
return 0;
return 4.0 + 3.0; etc.
```

In some examples, you have returned variables whose values are known when they are returned and in other examples you return constants. You can even return expressions. If the return statement is not present, then it means the return data type is `void`.

NOTES

You can also have multiple return statements in a function. However, for every call, only one of the return statements will be active and only one value will be returned.

Recursion

The previous section dealt with the concept of a function calling another function as well as multiple functions being called by a number of functions. A function calling itself is called recursion and the function may call itself either directly or indirectly. This concept is difficult to understand unless explained through examples. Every program can be written without using recursion, but the reverse is not true. Some problems, however, are suitable for recursion. For example, the factorial problem, can be solved using recursion as shown in Example 3.25 below:

Example 3.25:

```
to find the factorial of a given number*/
#include <stdio.h>
int main()
{
    int n;
    long int result;
    long int fact(int n);
    printf("Enter the number whose ");
    printf("factorial is to be found\n");
    scanf("%d", &n);
    result=fact(n);
    printf("result=%ld", result);
}
long int fact(int n)
{
    if (n<1) return 0;
    else
       if (n==1) return 1;
       else
          return (n*fact(n-1));
}
```

Result of the program

```
Enter the number whose factorial is to be found
10
result=3628800
```

NOTES

Now analyse how the program proceeds. You get an integer n from the keyboard. In order to find factorial n, you call fact(n), where fact is the function for finding the factorial of the number n. The recursion takes place in function fact. Assume that n = 1. The main function calls fact(1), which will be assigned to result in the main function after return from the function. In the function since n is equal to 1, 1 is returned and printed in main().

Next, assume you want to find out the factorial of, say 2, and fact(2) is called. In the function fact, since n is not equal to 1, n * fact(n – 1) is returned, i.e., 2 * fact 1 is returned to result, Result = 2 * fact(1). This intermediate result is stored somewhere and can be called stack. Stack is an array, which stores values and gives the last element first. The writing into stack is popularly called push and getting information from stack is called pop. You have not defined any stack and therefore, you can assume that the system does this for you. After pushing the intermediate result into stack, the program calls fact(1), which returns 1. Now the intermediate result is popped and the value of fact 1 is substituted to get the factorial of 2 as 2.

Let us now call factorial 5. We call fact and get back,

result = 5 * *fact*(4) [1]

Now fact(4) is called to get 4 * fact(3). Substituting this in equation [1] we get,

result = 5 * 4 * fact(3)

fact(3) again is called to get 3 * fact(2) and so on till we get fact(1), which will be returned as 1. Therefore, we get factorial 5 as 5 × 4 × 3 × 2 × 1. Such repetitive calling of the same function is called recursion. The calls are recursive calls. The result and function fact has been declared to be of type long so as to take care of large numbers. If the fact function were not called repeatedly, the program size would have become very large. Thus recursion keeps the program size small, but understanding recursion is not easy. If the program can be visualized as recursive, it will result in a compact code. Recursive functions can easily become infinite loops. Therefore, the functions should have a statement with if so that the program will definitely terminate. In the factorial program, assume for a moment that the first statement in fact function is absent. Then we have to end the program only when n becomes 1. What will happen if by chance n is entered as a negative number? The program will get into an endless loop. Therefore, to avoid such eventualities, you can either have a statement as follows:

```
If (n < 1)        exit()
```

Or, you could do this by the statement if (n < 1), return 0 as has been done here.

This will ensure that if either 0 or a negative number is entered, we get the factorial as 0 and the program will terminate gracefully.

NOTES

You should also note that recursive programs simulate the use of stack. We write to the stack and retrieve information from the stack. Writing to stack is called push and retrieving or reading or getting value from the stack is known as pop. The feature of stack is last-in-first-out (LIFO) and therefore, we get the value of the data entered last first as illustrated in the example below.

You push or pop only through the top of the stack. Assume that you want to push `a, b` and `c` one at a time. You push `'a'`. It will be on the top of the stack. When you push `b, a` will be pushed to the next lower location and `b` will occupy the top of the stack. Next when you push `'c', 'c'` will occupy the top of the stack, `'b'` one location before and `'a'` one location before `'b'`. Thus you can push data till the stack becomes full. If you pop the stack now, it will eject `'c'`, then `'b'` and so on. Thus you pop on a last-in-first-out basis.

Reconsider the factorial program in which we were storing intermediate results in a stack like manner and popping LIFO. Take the example of finding the factorial of `4`. On the first call, we get `4 * fact(3)`. You push this into the stack and in the next call you get `3 * fact(2)`. Again you push the intermediate result to stack and evaluate `fact(2)` to get `2 * fact(1)`. This is also put to stack. Now you pass `fact(1)` and get `fact(1)` as `1` after which we get the value of `fact(2)` by popping `fact(2)` as `2 * fact(1)`. The popping continues till the result is obtained. Such a conceptualization will help you to understand recursion easily.

We had earlier discussed a problem to reverse the characters in a string. The same program can be executed recursively. As you know, `\0` is appended at the end of the string. This property can also be used to solve the problem.

The program for reversing a word is given in Example 3.26 below:

Example 3.26:

```
to reverse a string using recursion*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
    char *wp =”abcdefgh”;
    void rev(char *w);
    rev(wp);
}
void rev(char *w1)
{
    char w=*w1;
    if (w!=’\0') rev(++w1);
       putch(w);
}
```

Result of the program

```
hgfedcba
```

How does the program work?

NOTES

The string `wp` is passed to the function `rev`. The first character is assigned to `char w` in the function. The leading character a from the `string wp` will be the first character of `'w'`. Since `w` is not NULL (`'\0'`), the function `rev` is called after incrementing the pointer. The pointer points to the next character in the string, which is `b`. Now `a` is pushed to the stack. Now the string `*w1` starts from the character `b` because of the increment. The function `rev` is called with `w1`, which begins with `b` now. It is copied to `w` and again the if statement is entered. Again, since `b` is not NULL, function `rev` is called, and on pushing `b` to stack. You get NULL in the next increment after pushing `h`, i.e., (`++w1`). Therefore, the next statement `putch()` is executed, which pops a character at a time in the reverse order of pushing. Thus, the word has been reversed. Note that `w1` is the address, which points to the first character. It is obvious recursion, which is neither easy to explain nor easily understood as the problem below will confirm.

Implementation of Euclid's GCD Algorithm

The Euclid's `gcd` algorithm is quite suitable for recursion. The modified algorithm, which uses recursion, is given below:

Algorithm using recursion

```
Main Function
Step 1 Read two integers m and n
Step 2 Call function gcd (m, n)
Step 3 Print gcd
          Function gcd (m, n)
Step 1 if (n==0) return m;
          else
return (gcd(n, m%n));
Step 3 End
```

When two integers are received, the main function calls `gcd` function. In the `gcd` function, it is checked whether `n` equals to zero. If so, `m` is the `gcd`. If not, the function calls `gcd` again by changing the arguments as follows:

```
M = n
N = m % n
```

See the clarity in the above function.

We can easily observe that by using recursion, even the number of steps in the program has gone down. It requires a little skill to convert the program into a recursive function. The program implementing the preceding algorithm is given as follows:

Example 3.27:

```
Euclid's GCD*/
#include<stdio.h>
int main()
{
int m, n;
int gcd(int m, int n);
printf("Enter 2 integers\n");
scanf("%d %d", &m,&n);
printf("GCD of %d and %d=%d", m, n, gcd(m,n));
}
int gcd(int m, int n)
{
if (n==0) return m;
else
return (gcd(n, m%n));
}
```

NOTES

The program was executed twice and the result of the program is given hereafter.

Result of the program

```
First Time
Enter 2 integers
12 256
GCD of 12 and 256 = 4
Second Time
Enter 2 integers
1225 625
GCD of 1225 and 625 = 25
```

We can even enter the first number to be lower than the second number as executed during the first time. The program works all right because in one iteration, the numbers get reversed, namely the first number is larger than the second number after iteration. Thus, recursion is quite suitable for solving this problem.

CHECK YOUR PROGRESS

5. What are the steps to be taken to execute a program written in C?
6. What is string processing?
7. What is a recursive function?
8. List some library functions of C.
9. What are the three parts of a function?

3.20 POINTERS

NOTES

Definition

A pointer is a variable that points to another variable, i.e. it holds the memory address of another variable.

Pointer Declaration

As in case of all other variables, pointer variables, should also be necessarily declared before you use them in 'C' program. When a pointer variable is declared, an asterisk (*) must precede the variable name. This identifies the fact that the variable is a pointer.

Syntax:

data_type *ptvar;

Where ptvar refers to the name of the pointer variable.

Example 3.28:

```
/* Program to explain pointer variables, how to store
address, and print */
#include <stdio.h>
main ()
{
   int a;

   int *p;
   /* *p = Pointer variable declaration to store the
            address of an integer variable, * called
as Indirection      operator (or) value at address
operator */
   a = 10;
   p = &a;
   /* Assigning the address of integer variable a
      to pointer variable p. & is the address of
operator */
   printf("\n The value of a = %d\n", a); // Printing
value of a
   printf("\n The address of a = %u\n", &a); // & a
means address of a
   printf("\n The address of a = %u\n", p); // printing
the p value
   printf("\n The value of a = %d/n",*p);
            /* *p means value at p contained address
*/
```

```
      printf("\n The address of p = %u\n, &p); // & p
means address of p
      printf("\n The value of p = %u\n:, p); /* Printing
p value, i.e. address of a */
    printf("\n The value of a = %d\n", *(&a)); // same as
a
    }
```

Output:

```
The value of a = 10
The address of a = 65524
The address of a = 65524
The value of a = 10
The address of p = 65522
The value of p = 65524
The value of a = 10
```

Address and Indirection Operators

Address Operator: It is an operator which assesses the address of its operand. It is also called substitution variable.

Example 3.29:

```
#include <stdio.h>
   main( )
   {
   int i = 3;
   int * p;
   p = & i;
printf("%d %d", i, p);
   }
```

In the statement p = &i, the address operator assign the address of it to the pointer 'p'.

Indirection Operator (*): In Example 3.29, the value of i can be accessed by *p, where * refers to the unary operator which is known as **indirection operator**.

The indirection operator operates only on a pointer variable.

Example 3.30:

```
#include <stdio.h>
main ( )
{
int i = 3;
int *p, j;
p = &i
```

NOTES

```
j = *p;
printf("%d %d %d", i, p, j);
}
```

NOTES

Output:

3,4082

i.e. the value of i is indirectly assigned to j.

Pointer to Pointer

As we know a pointer is a variable which holds the address of another variable. Since a pointer itself is a numeric variable, it is also accumulated in computer's memory at a specific address. A pointer to pointer refers to a variable that controls the address of a pointer variable. To declaring a pointer to pointer, a double indirection operator should be used.

Syntax:

```
data_type *ptvar;
int **j
```

Most common use involves array of pointers.

Example 3.31:

```
/* Program to illustrate pointer to pointer */
   #include <stdio.h>
   main ()
   {
   int i, *p, **p1, ***p2, /*** indicates pointer of
pointer to pointer,
   *** indicates pointer of pointer to pointer */
i = 10;
p = &i;   /* Assigning address of a to pointer p */
p1 = &p;  /* Assigning address of pointer p to
            pointer p1 */
p2 = &p1;   /* Assigning address of pointer of
pointer  to pointer to pointer */
printf("\n The addresses of i = %u", &i);
printf("\n The address of i = %u", p);
printf("\n The address of p = %u", &p);
printf("\n The address of p = %u", p1);
printf("\n The address of p1 = %u", &p1);
printf("\n The address of p1 = %u", p2);
printf("\n The address of p1 = %u", &p2);
printf("\n\n The value of i = % d", i);
```

```
    printf("\n The value of i = %d", *p);
    printf("\n The value of i = d", **p1); /* Value at
address of *p1
                (value at off of p) */
    printf("\n The value of i = %d", ***p2); /* Value
at address of **p2
                value at add of *pw value
                                        at add of p2 */
    printf("\n\n Teh value of p = %u:, p);
    printf("\n The value of p1 = %u", p1);
    printf("\n The value of p2 = %u", p2);
    }
```

NOTES

Output:

The address of i = 65524

The address of i = 65524

The address of p = 65522

The address of p = 65522

The address of p1 = 65520

The address of p1 = 65520

The address of p2 = 65518

The value of i = 10

The value of i = 10

The value of i = 10

The value of i = 10

The value of p = 65524

The value of p1 = 65522

The value of p2 = 65520

Function with Pointers (Call by Reference)

Pointers are often passed as arguments to a function. When a pointer is passed to a function, the address of a data item is passed to function. When the address of a data item is passed as an argument to a function, it is referred to as *call by reference*.

The contents of that address can be accessed freely, either within the function or with the calling portion. Moreover, any change that is made to the data item will be recognized in both the function and the calling routine.

Example 3.32:

```
/* Program to illustrate passing the pointer to function
(call by reference) */
```

NOTES

```
#include <stdio.h>
void swap (int *, int *);
main ( )
{
   int a, b ;
   printf("Enter two values \n");
   scanf("%d %d", &a, &b);
   printf("\n Before swapping the values of a and
b\n");
   printf(" %d\t %d\n\n", a,b);
   swap (&a, &b); /* Passing address of a and address
of b to swap */
   printf("After swapping the values of a and b\n");
   printf("%d\t%d", a, b);
}
void swap (int *p, int *q)
/* p hold address of a and q holds address of b */
{
int temp;
   temp = *p;     // swapping a and b values through
                  pointer
   *p = *q;
   *q = temp;
   return;
}
```

Output:

Enter two values

12

45

Before swapping the values of a and b

12 45

After swapping the values of a and b

45 12

Pointer versus srrays

Array elements are invariably accumulated in adjoining memory locations. It would be easier to access the elements using a subscript if there is fixed logic involved in accessing the elements.

Array elements should be accessed using pointers, if the elements are to be accessed in a fixed order, say from beginning to end or from end to beginning or

every alternate element. Accessing the elements by pointers would work faster than subscripts.

Example 3.33:

```
/* Program to illustrate arrays with pointers */
       #include <stdio.h>
       main ()
    {
       int arr[10]. *p, i;
       p = &arr [0];        /* Passing the base address
of an array to pointer p. */
          printf("Enter values for array arr\n\n");
          for (i = 0; i < 10; i++)
          {
             printf("arr[%d] =", i);
             scanf("%d", p + i); /* P initially holds
base address of arr array here p + i means incrementing
pointer p */
    }
    printf("\n\n The array elements are \n\n");
    for (i = 0; i < 10; i++)
    {printf("arr [%d] = %d\t%d\t%d\t%d\n", arr[i], i[arr],
*(i + p), *(p + i)); }
```

Pointer and Multi-Dimensional Arrays

Similar to one-dimensional array, a multi-dimensional array can also be represented with an equivalent pointer notation. A two-dimensional array, for example, is actually a collection of one-dimensional array. Therefore, we can define a two-dimensional array as a pointer to a group of contiguous one-dimensional arrays.

Syntax:

```
data_type (*ptvar) [expression 2]
int (*i) [20];
```

This can be generalized to higher dimensional arrays.

Syntax:

Data_type (*ptvar) [Expression 2] [Expression 3] ..... [Expression n];

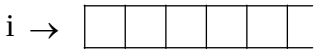
Example 3.34:

```
int (*i) [20] ;
```

In the first declaration, i is defined to be a pointer to a group of contiguous, 20-element integer arrays. Thus, i points to first 20-element array, which is actually first row—row 0 of the original two-dimensional array. Similarly (i + 1) points to the second 20-element array, which is the second row and so on.

NOTES

NOTES



1st one-dimensional array

2nd one-dimensional array

.....

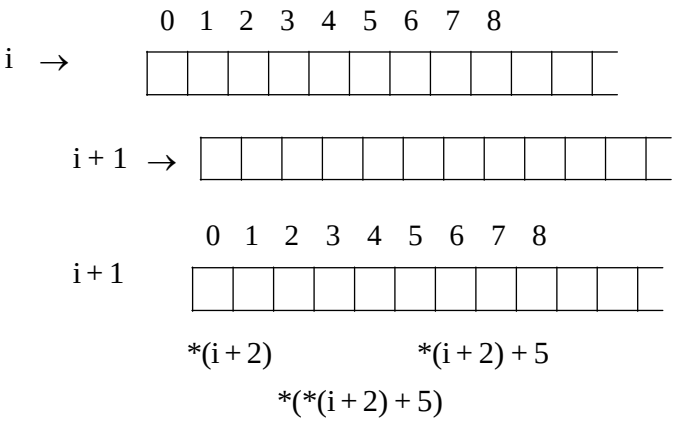
10th one-dimensional array

Accessing elements of two-dimensional array

Suppose you want access element of row 2, column 5.

One way is i [2] [5]

Another way is *(*(i + 2) + 5)



First note that (i + 2) is a pointer to row 2. Therefore, the object of this pointer *(i + 2) refers to the entire row. Since row 2 is a one-dimensional array, *(i + 2) is actually a pointer to the first element in row 2. Hence, (*(i + 2)) is a pointer to element 5 in row 2. The object of this *(*(i + 2) + 5), therefore, refers to item in column 5 of row 2.

i[2] [5], *(i [2] + 1), *(*(i + 2) + 1)

All above expressions refers to same elements.

Example 3.35:

```
/* To explain pointers with double dimensions */
#include <stdio.h>
main ( )
{
int dob [3] [3], *p, i, j; for (i = 0; i < 3; i++)
   {
```

```
        printf("\n\n Enter % d row values \n\n", i +
1);
           for (j = 0; j < 3; j++)
           {
              printf(dob[%d] [%d] =", i, j);
              scanf("%f", (*(dob + i) + j));
           }
        }
        printf("\n\n Array values are \n\n");
        for (i = 0; i < 3; i++)
        {
           for (j = 0; j < 3; j++)
   printf("%d\t", *(*(dob + i) + j));
           printf("\n"); } }
```

Array of Pointers

An array of pointers refers to a collection of addresses. The addresses present in the array of pointers may be addresses of ordinary variables or addresses of array variables or any other addresses. All those rules which are applicable to ordinary arrays are also applicable to array of pointers.

Example 3.36:

```
   /* Program to illustrate the array of pointers */
   #include <stdio.h>
   main ( )
   {
      int arr[3], *point[3], i, j; /* point[3] is to
store the address */
      for (i = 0; i < 3; i++)
         point[i] = arr + i; /* Assigning address of
arr array to point array */
      printf("\n\n Enter values \n\n");
      for(i = 0; i < 3; i++)
         scanf(%d", arr + i);
   printf("\n The addresses of each array elements are
\n");
      for (i = 0; i < 3; i++)
         printf("arr [%d] = %u\n", point[i]);
      printf("\n The addresses of each array elements
are\n");
      for (i = 0; i < 3; i++)
         printf("arr[%d] = %u\n", i, &arr[i]);
```

NOTES

```
        printf("\n The values in dob array are \n");
        for ( i = 0; i < 3; i++)
           printf("arr [%d] = %d\n", i, *point [i]);
           printf("\n The values in dob array are \n);
        for (i = 0; i < 3; i++)
           printf("arr [%d] = %d\n", i, arr[i]);
    }
```

Dynamic Memory Allocation

The 'C' library has functions to allocate memory storage space at run-time. This process is known as the dynamic memory allocation. The function of the dynamic memory allocation is to allow the program to react, while it is carrying out the demands of memory, such as use input. Each function for running dynamic memory allocation needs the header file alloc.h.

The various dynamic memory functions are as follows:

* malloc ( )
* calloc ( )
* realloc ( )
* free ( )

The malloc ( ) function

The malloc ( ) function is one of the C's memory allocation function. When we call a function, we just pass it the number of bytes of memory needed. malloc( ) finds and reserves a block of memory of the required size and returns the address of the first byte in the block and the return type is a pointer of type void.

Syntax:

Void *malloc(size_t num);

- malloc( ) allocates a block of memory, i.e. the number of bytes stored in size_t.
- The arguments size_t is defined in stdlib.h as unsigned.
- The malloc( ) function allocates number of bytes of storage space and returns a pointer (address) to the first byte.

The calloc( ) function

This function deals with memory and it is normally used for requesting the memory space at run-time for storing derived data types such as arrays and structures. As malloc ( ) allocates a single block of storage space, calloc( ) allocates multiple blocks of storage. If the allocation is successful, the function returns a pointer (address) to the first byte. If allocation is not successful, the function returns null.

Syntax:

Void *calloc(size_t num, size_t size);

- Here size_t unsized.
- Num is the number of objects to allocate.
- Size is the size in bytes of each object.

NOTES

Example 3.37:

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
main ( )
{
char *str;
str = (char*) calloc (10, 1);
/* allocate memory for string */
strcpy (str, "Srinivas"); /* copy Srinivas into str
*/
printf("String is %s\n", str); /* display string */
free (str);                    /* free memory */
}
```

Output:

String is Srinivas

The realloc( ) function

The realloc( ) function changes the size of a block of memory that was already allocated by the functions malloc( ) or calloc ( ). This process is termed as reallocation of memory.

Syntax:

Void *realloc(void *ptr, size_t size);

- Here ptr argument is a pointer to the original block of memory; size specifies the new size.

Example 3.38:

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
main ( )
{
char *name ;
name = (char*) malloc (12);
                          // Allocating memory
```

NOTES

```
strcpy (name, "Parthiv Sai");
printf("Name is %s\n", name);
name=(char*) realloc (name, 30);
                        // Reallocating memory to 20)
strcpy (name, "B.S. Software Solutions");
printf("String is %s\n", name);
free (name);            /* Releasing memory */
}
```

Output:

Name is Parthiv Sai

String is B.S. Software Solutions

The free( ) function

The free( ) function is used to de-allocate or free the memory, which is allocated by the malloc( ), calloc( ) or realloc( ) functions so that the memory is available for future use.

Syntax:

void free(void *ptr);

- The free( ) function frees the memory pointed to by ptr.
- If ptr is null, free( ) function does nothing.

Built-in String Functions

A string is a collection of characters and terminated with NULL characters and it is denoted by **\0.** The four string handling functions are as follows:

- strlen() – returns the length of the given string as argument.
- strcpy() – to copy the string from source to destination variable.
- strcmp() – to compare the two given string.
- strcat() – to concatenate (combine or join) the both given strings into one.

*For all the above functions, we require the header file string.h.

strlen(): The function strlen() takes string as argument and returns the length of the integers as shown below in the code chunk.

```
Syntax: int strlen(char*);
int len;
char *str ="Sreenivasa Rao";
len = strlen(str);
printf("%d", len);
```

Here the ***len*** variable contains the 14 value since the number of characters is fourteen including the space character.

strcpy(): The function strcpy() copies the content of source string to destination string, stopping after the terminating NULL character. The syntax is as follows:

NOTES

```
char * strcpy(char *dest, char *source);
```

This function returns the pointer to the character.

```
char *s=" Parthiv Sai";
char *s1;
strcpy(s1, s);
printf("the copied string is %s", s1);
```

strcmp(): The function strcmp() compares the two given strings character by character, and returns a value 0 if both the strings are equal, and value < 0 if the string one is less than string two and > 1 if string one is greater than string two. The syntax is as follows:

```
int strcmp(char *, char*);
int val;
char *str1= "Apple";
char *str2 = "apple";
val = strcmp("Apple", "apple");
```

Here, the first string is less than the second string; hence, the value of val is less than zero (0), since the ASCII value of character 'A' differs from character 'a'.

strcat(): This function concatenates two strings resulting in a single string. It takes two arguments, namely, the pointers to the two strings. The resultant string is stored in the first string specified in the argument list.

```
char destination[25];
char *blank = " ", *c = "Bandaru", *t = "Parthiv";
strcpy(destination, t);
strcat(destination, blank);
strcat(destination, c);
printf("%s\n", destination);
```

3.21 INPUT AND OUTPUT

Input and output operations are very much concerned with interacting with programmers and so their form is highly dependent on the specific programming language used, and occasionally, even on the computer system itself.

We define two new types of statement to handle input and output. The read statement allows us to read given values into indicated variables and the write

NOTES

statement allows us to display results. Input may come from a terminal or from some other form of input device. Output may appear on a display screen or printed on paper.

The form of the read statement is as follows:

```
Read(input list)
```

The *input list* gives the names of the variables to which values are to be given as they are encountered in the input stream.

For example:

```
Read(A, B, C)
```

The next three values encountered would be given to the variables A, B, and C. The first to A, the second to B, and the third to C.

Suppose that the actual values entered were 16, 3, 7, 21, 16, 0, 4, 8, 1. These could be on one input line, or spread over several. Consider the following three read statements:

```
Read(A, B, C)
Read(D, E, F, G)
Read(X, Y)
```

The values are given to the variables A, B, C, D, E, F, G, X, Y one by one in the order written. The process of giving these values to the indicated variables is like the assignment operation in two aspects. First, it is a destructive operation in the same sense that the assignment operation was a destructive operation. Whatever values, if any, that the variables had previously possessed are overwritten. Data of any type can be input; if the value in the input stream is of a different type from that of the input list, a conversion is required.

The output statement is like the input statement in format. It is possible to display the contents of any variable, the result of any expression or the value of any constant. The general form is as follows:

```
Write(output list)
```

The following sequence of statements illustrates the action of a write statement. Assume that all variables are real.

```
MARK1 73
MARK2 65
MARK3 94
MARK4 87
AVERAGE (MARK1 + MARK2 + MARK3 + MARK4) / 4
Write(AVERAGE)
```

The first four statements assign to the indicated variables on four tests. The fifth statement computes the average grade. The write statement then displays this calculated result:

79.75

Let us observe the case when we replace the above write statement with

```
Write(MARK1, MARK2, MARK3, MARK4, AVERAGE)
```

NOTES

The resulting output would be as follows:

73 65 94 87 79.75

Note that all numbers are written as real numbers. This is because the variables themselves are all real variables. Now, however, we have simply five numbers displayed. Suppose, instead, that we gave the following output statements:

```
Write('Individual grades are', MARK1, MARK2, MARK3,
MARK4)
Write('Final average is', AVERAGE)
```

In this case, the output would be displayed as follows:

```
Individual grades are 73 65 94 87
Final average is 79.75
```

In terms of information content, the last form is definitely superior. This last example shows the use of two different kinds of items in an output list. We have used variables such as MARK1 and AVERAGE. We have also used constants, in this case string constants, such as 'Final average is'.

There can be many special features associated with input and output. Here a very general approach is given.

Finally, in designing the input portion of an algorithm, some consideration should be given to robustness of an algorithm. A robust algorithm is one, which produces reasonable results no matter what input is supplied. The requirements of robustness are: (i) the program should check the validity of its input and (ii) the program should take some appropriate action for unacceptable input. In a batch environment, these actions may involve including proper messages in the output stream and ending execution. For example, in an interactive environment, a more proper action may include output of a message followed by a prompt for input of a corrected value. Checking for some types of invalid data is relatively easy. Here in the example, negative grades should be rejected. Most compilers automatically terminate execution with a standard message for some types of errors. These often include illegal characters when expecting numeric input and many more.

Input–Output Operations

In 'C', input and output operations can be classified as follows:

- Console Input/Output operations
- Disk Input/Output operations
- Port Input/Output operations

NOTES

Console Input/Output operations

These functions get input from the keyboard and gives output to the VDU. These functions can be further classified into two categories: Formatted Console Input/ Output functions and Unformatted Console Input/Output functions. All these functions are defined in standard input–output library and the header files **stdio.h** and **conio.h** includes all input/output functions.

Formatted Input/Output functions get the input from the keyboard and give the output to the VDU as per the specified requirement. **scanf()** and **printf()** comes under the formatted input/output functions.

The **scanf()** function allows the user to enter numerical values and string values. This function is described in the system file **stdio.h**.

Syntax for scanf( ):

```
int scanf(<format string>, arguments)
Where
<format string> consists of format specifiers
```

arguments represent address of variables and this function returns an integer value that shows the number of successful input readings.

Note: In scanf() function, space(s) should not give after the last format specifier in the format string.

The **printf()** function is the opposite of **scanf()**. It writes data available in the variables to the screen and like **scanf(),** it is also described in the system file called **stdio.h**.

Syntax for printf():

```
int printf(<format string>, arguments)
Where
<format string> may consists of format specifiers
```

arguments represent variables, constants or addresses and this function returns an integer value that shows the number of characters printed out to the screen.

Unformatted Input/Output functions/macros get the input as a single character or multiple characters using keyboard and gives output of a single character or multiple characters to the VDU.

Function Name	Operation
int **getch**(void)	Reads a character form keyboard without echo and without pressing enter key it returns character value.
int **getche**(void)	Reads a character from keyboard with echo and without pressing enter key it returns character value.
int **fgetc**(stdio)	Reads characters from keyboard with echo and with pressing enter key it returns first character value.

NOTES

int **getc**(stdin)	Reads characters from keyboard until enter key is pressed having echo and with pressing key it returns first character value. It is a macro version of **fgetc()**.
int **getchar**(void)	Reads characters from keyboard until enter key is pressed having echo and with pressing enter key it returns first character value. It is the macro form of **getc()**.
int **putch**(int)	Prints a character to the screen.
int **fputc**(int, stdout)	Prints a character to the screen.
int **fputc**(int, stdprn)	Prints a character to the printer.
int **putc**(int, stdout)	Prints a character to the screen. It is the macro version of **fputc(stdout)**.
int **putc(**int, stdprn)	Sends a character to the printer. It is the macro version of **fputc(stdout)**.
int **putchar**(int)	Prints a character to the screen. It is the macro form of putc(stdout).

Note:

- **getch(), getche(), fputc(), putchar()** and **putc()** are defined in <**conio.h**>, whereas

  getchar(), getc(), fgetc() and **putch** are defined in <**stdio.h**>
- **stdout** refers to standard output buffer.

  stdin refers to standard input buffer.

  stdprn refers to standard printer buffer.
- In reading a character, all functions and macros first check whether there are any characters in the console buffer. If any characters are present, it directly reads from the buffer without waiting for input. To clear the input buffer, use **fflush(stdin)**.Where **stdin** points input buffer and **fflush** clears the input buffer.

Unformatted I/O functions/macros for string type:

Function Name	**Operation**
char * **gets**(char *string)	Receives a string until a newline character (\n) is found (until enter key is pressed). It returns a pointer to the argument string.
int **puts**(const char *s)	Sends a string to the screen and appends a newline character. It returns the last character written. Otherwise, a value of EOF is returned.
char ***cgets**(char *str)	Reads a string from console. str[0] must contain the maximum length of the string to be read. On return,

NOTES

	str[1] is set to the number of characters actually read. The string begins at str[2]. The function returns &str[2].
int **cputs**(const char *str)	Writes a string to the text window on the screen. It returns the last character printed.

Note: **getch()** and **puts()** are defined in <**stdio.h**>.

cgets() and **cputs()** are defined in <**conio.h**>.

3.22 FILES

For programming `<stdio.h>` is included in every file. This file is essential for any program for reading from standard input device or writing to the standard output device. File `<stdio.h>` has declarations to the pointers to three files, namely `stdin`, `stdout` and `stderr`. It means that the contents of these files are added to the program, when the program executes. Each of the files performs an essential task as follows:

(a) `stdin` facilitates usage of the standard input device for program execution and normally points to the keyboard, which is the default input device.

(b) `stdout` facilitates the usage of a standard output device where program output is displayed and points to the video monitor.

(c) `stderr` facilitates sending error messages to the standard device that is again the monitor.

`stdin`, `stdout` and `stderr` are pointers or file pointers and are declared in `<stdio.h>`. So far you have been using `stdin` and `stdout` for input and output. In this unit, we will learn to use disk drives either hard disk or floppy disk as the medium for input/output. In day-to-day usage of large applications, the standard input/output is neither convenient nor adequate to handle large volumes of data and hence, the disk drives only serve as Input/Output (I/O) devices. You will learn about the usage of files for storing data, popularly known as data files. Data files stored in the secondary or auxiliary storage devices, such as hard disk drives or floppy disks, are permanent unless deleted. In contrast, what is written to the monitor is only for the immediate use. The data stored in disk drives can be accessed later and modified, if necessary.

In C, we come across two types of files:

a. Stream-oriented

b. System-oriented

System-oriented files or low-level files are more closely related to the operating system and hence, require more complex programming skills to use them. They may be found to be more efficient than the former in some cases, but we will not discuss them further because of their complexity. Instead, we will discuss stream-oriented files only in this unit.

NOTES

Stream-oriented files are also called standard files. Data can be stored in the standard files in two ways as given below:

- Storing characters or numerals consecutively. Each character is interpreted as an individual data item.
- The data items are arranged in blocks in an unformatted manner. Each block may be an array or a structure.

Let us see how disk I/O is organized. If the file is stored in a floppy or hard disk drive, the following actions are involved in reading from the file:

- Finding out where the data is.
- Positioning the head over the correct location on the disk.
- Reading the content.
- Transmitting to the main memory.

Similar activities are involved in writing to a disk as well. If the computer, or more specifically the operating system, which handles files in a computer, reads or writes one character at a time comprising the four steps listed above, then it will be uninteresting and the response will be too slow. It may cause wear out of the storage system quickly. Therefore, it would be better to receive large volumes of data or characters to a buffer in the computer system and then perform whatever actions are dictated by the program. Similarly, all characters to be written can be collected in a buffer and written on to the disk, either after the buffer is full or after the operation is completed. This will minimize the overheads required for the read or write operations. The buffer is also, the memory, which is used to store data temporarily without the knowledge of the user. In fact, you created a buffer and stored values into it before printing them using the sprintf() function. The concept is similar here also. This is a good practice. Therefore, the characters are read or written through a buffer assigned by the system. The operations are essentially performed as depicted pictorially as follows:

File Pointer

What is a file pointer? It is a pointer to a file, just like other pointers to arrays, structures, etc. It points to a structure that contains information about the file. The information connected with a file is as follows:

- Location of the buffer
- The position in the file of the character currently being pointed to
- Whether the file is being read or written
- Whether an error has occurred or the end of the file has been reached

You do not need to know the details of these because `stdio.h` handles it elegantly. There is a structure with `typedef FILE` in `stdio.h`, which handles all file-

NOTES

related tasks as above, whether it is in the floppy or the hard disk drive. Therefore, in order to use a file without difficulty, you have to include `stdio.h` and declare a file pointer, which points to `FILE` as shown:

```
FILE * fp;
```

Therefore, the file pointer points to a structure, which contains information about the file management functions. When you open a file, and when the opening of the file is successful, the file pointer will point to the first character in the file. In other words, the file gets opened and loaded to the buffer. NULL is a macro defined in `<stdio.h>`, which indicates that file open has failed. Therefore, when file open is successful, the file pointer will point to the address of the buffer, which will be a non-zero integer value. If not, the file pointer will get a value of NULL, which is 0.

The file pointer will point to the next character after the first one is fetched or copied on to the system. The structure `FILE` keeps track of where the pointer remains at any point in time after opening the file. It keeps track of which files are being used. It also knows whether the file is empty, the end of the file has been reached or an error has occurred. You do not have to worry about the file management tasks once a file pointer has been declared in our program to point to `FILE`. Since `FILE` is known to `<stdio.h>`, you do not have to bother about it. This declaration of structure `FILE` has relieved the programmer from most of the mundane jobs.

Opening and Closing a Data File

Any file has to be opened for any further processing, such as reading, writing or appending, i.e., writing at the end of the file. The characters will be written or read one after another from the beginning to the end, unless otherwise specified. You have to open the file to assign file pointer for taking care of further operations. Hence, you can declare,

```
FILE * fp;
fp = fopen ("filename", "r");
```

The filename is the name of the file, which you want to open. You must give the path name correctly so that the file can be opened. `'r'` indicates that the file has to be opened for reading purposes.

`fp = fopen ("Ex1.C", "r");` will enable opening file `Ex1.C`.

Therefore, the arguments to `fopen()` are the name of the file and the mode character. Obviously `w` is for write, `a` for append, i.e., adding at the end of the file. If these characters are specified, the operations as indicated can be performed after opening the file. It is, therefore, essential to indicate the operations to be performed before opening the file. When the file is opened in the `'w'` mode, the data will be written to the file from the beginning. This means that if the named file is already in existence, the previous contents will be lost due to overwriting. If the file does not exist, then a file with the assigned name will be opened. When the append mode is specified, the writing will start after the last entry, or in other words, previous contents of the file will be preserved.

NOTES

FILE provides the link between the operating system and the program currently being executed. FILE is a structure containing information about the corresponding files, including information such as:

- Location of file
- Location of buffer
- Size of file

After executing the command in read mode, the file will be loaded into the buffer if it is present. If the file is absent, or the file specification is not correct, then the file will not be opened. If the opening of the file is successful, the pointer will point to the first character in the file, and if not, NULL is returned, meaning that the access is not successful. The fopen() function returns a pointer to the starting address of the buffer area associated with the file and assigns it to the file pointer, fp in this case.

After the operations are completed, the file has to be closed. The syntax for closing file is given below:

```
fclose(filepointer);
```

fclose() also flushes or empties the buffer. The function fputc() performs putting one character into a file. If for every fputc(), the computer prints a character to a file, then it will get tired. Therefore, it collects all the characters to be written onto a file in the buffer. When the buffer is full or when fclose() is executed, the buffer is emptied by writing to the hard disc drive in the assigned file name.

Concept of Binary Files

We can open files in the text mode or the binary mode. In the binary mode, data stored in binary format and storage space will be equal to the number of bytes required for storage of various data types. In the text mode, they will be stored as alphanumeric characters. If you require to use the file in the binary mode, you must use 'rb' for reading, 'wb' for writing, and 'ab' for appending. If you want to store data in the text mode, you have to append *t* to the mode character as 'rt', 'wt', 'at', etc. Since the default is in the text mode, t will be assumed if nothing is specified after the mode character. Therefore, mode 'w' means opening a text file for writing.

The difference between opening files in the binary mode and the text mode are given below in Table 3.7:

NOTES

Table 3.7 *Difference between Binary Mode and Text Mode Operations*

Text Mode	Binary Mode
New line character (\n) is converted to CR|LF combination while writing to file.	No such conversion.
While reading, CR|LF is converted back to \n.	Does not arise.
A special character is inserted at the end of the file. While reading the file, EOF is detected.	There is no such arrangement.
Text mode needs more than the 2 bytes for storing an integer, since it treats each digit as a character. e.g., 30,000 needs 5 bytes.	In binary mode the numbers will be stored in the specified width. 30000 needs 2 bytes only.

Therefore, binary files and text files are to be processed taking into account their properties as above, although the file could be opened in any mode. The file I/O functions, such as `fgetc`, `fputc`, `fscanf`, `fprintf`, `fgets`, `fputs`, are applicable to the operations in any of the modes.

The files can be used to store employee records using structures in a payroll program. Book records can be stored in a file in a library database. Inventories can be stored in a file. However, storing all these in the text mode will consume more space on the file. Hence, the binary mode can be used to create the files. Some files cannot be stored in the text mode at all, such as executable files.

Formatted I/O Operations with Files

We are familiar with reading and writing. So far we were reading from and writing to standard input/output. Therefore, we used functions for the formatted I/O with `stdio` such as `scanf()` and `printf()`. We also used unformatted I/O such as `getch()`, `putch()` and other statements. When dealing with files, there are similar functions for I/O. The functions `getc()`, `fgetc()`, `fputc()` and `putc()` are unformatted file I/O functions similar to `getch()` and `putch()`. We will consider the formatted file operations in this section. When it pertains to standard input or output, we use `scanf()` and `printf()`. To handle formatted I/O with files, we have to use `fscanf()` and `fprintf()`.

We can write numbers, characters, etc. to the file using `fprintf()`. This helps in writing to the file neatly with a proper format. In fact, any output can be directed to a file instead of the monitor. However, we have to indicate which file we are writing to by giving the file pointer. The following syntax has to be followed for `fprintf()`:

```
fprintf(filepointer, "format specifier", variable names);
```

We are only adding the file pointer as one of the parameters before the format specifier. This is similar to `sprintf()`, which helps in writing to a buffer.

In the case of `sprintf()`, buffer was a pointer to a string variable. Here, instead of a pointer to a string variable, a pointer to a file is given in the `fprintf()` statement. Like the string pointer in `sprintf()`, the file pointer should have been declared in the function and should be pointing to the file.

NOTES

Before writing to a file, the file must be opened in the write mode. You can declare the following:

```
FILE * fp ;
fp = fopen ("filename", "wb");
```

You have to write `wb` within double quotes for opening a file for writing in the binary mode. Therefore, `fopen()` searches the named file. If the file is found, it starts writing. Obviously the previous contents will be lost. If a file is not found, a new file will be created. If unable to open a file for writing, NULL will be returned.

We can also append data to the file after the existing contents. In this manner, we will be able to preserve the contents of a file. However, when you open the file in the append mode, and the file is not present, a new file will be opened. If a file is present, then writing is carried out from the current end of the file. After writing is completed either in the write mode or the append mode, a special character will be automatically included at the end of the text in case of text files. In case of binary files, no special character will be appended. This can be read back as EOF. Usually it is – 1, but it is implementation-dependent. Hence, it is safer to use EOF to check the end of the text files.

Writing and Reading a Data File

Let us look at a program to write numbers to a binary file using `fprintf()` and then read from the file using `fscanf()`. It is given in Example 3.39.

Example 3.39: `writing`

```
digits to a binary file and then reading*/
#include <stdio.h>
int main()
{
    int alpha,i;
    FILE *fp;
    fp=fopen("ss.doc", "wb");
    if(fp==NULL)
       printf("could not open file\n");
    else
    {
       for (i=0; i<=99; i++)
          fprintf(fp," %d", i);
          fclose(fp);
          /*now read the contents*/
          fp=fopen("ss.doc", "rb");
       for (i=0; i<100; i++)
```

NOTES

```
            {
               fscanf(fp,"%d", &alpha);
               printf(" %d", alpha);
            }
            fclose (fp);
        }
    }
```

Result of the program

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99

What does the program do?

(a) The file `ss.doc` is opened in the binary mode for writing. If the opening of the file was not successful, the message will be displayed and program execution will stop. If successful, the program will enter the `else` block. Numbers 0 to 99 are generated one after another and written then and there to the file using the `fprintf()` function. There should be space before `%d` as shown in `fprintf()`, otherwise the program may not work.

(b) The file is closed using `fclose()`.

(c) Now the same file is opened for reading in the binary mode.

(d) Next the text is scanned using `fscanf()`, one at a time, and written on the monitor using simple `printf()`. The difference between `scanf()` and `fscanf()` is the specification of the file pointer before the format specifier.

(e) After reading, the file is closed.

The result of the program is read from the file `ss.doc` and printed on the monitor.

In all the programs involving files, a similar check to see that file opening was successful should be made. For the sake of improved readability, this statement has been skipped in the rest of the programs.

Let us look at one more example of writing, appending and then reading one integer at a time with the help of the `for` loop. Look at Example 3.40 below:

Example 3.40: `writing, then appending digits to a file and then reading*/`

```
#include <stdio.h>
int main()
{
    int alpha,i;
    FILE *fp;
    fp=fopen("ss.doc", "wb");
```

```
    for (i=0; i<20; i++)
       fprintf(fp," %d", i);
       fclose(fp);
       fp=fopen("ss.doc", "ab");
    for (i=20; i<100; i++)
       fprintf(fp," %d", i);
       fclose(fp);
    /*now read the contents*/
    fp=fopen("ss.doc", "rb");
    for (i=0; i<100; i++)
    {
       fscanf(fp,"%d", &alpha);
       printf(" %d", alpha);
    }
    fclose (fp);
}
```

NOTES

Result of the program

The result will be same as Example 3.39

A binary file is opened in the write mode, and digits from 0 to 19 are written on to the file. The file is then closed using `fclose()`. The same file is opened in the append mode again, and numbers from 20 to 99 are appended to the file. After the file is closed, the file is opened in the read mode. The contents of the file are read using `fscanf()` and written to the monitor. Remember to leave a space before `%d` in `fprintf()` as otherwise you may have a problem. The file is closed again. We have used the same file pointer, since at any time only one file is in use. If more than one file is to be kept open simultaneously, it may call for multiple pointers.

Unformatted Data Files

After having worked with the formatted I/O, let us now look at the unformatted I/O. If you want to read a character from the file, you can use the `getc()` or `fgetc()` functions. If `alpha` is the name of the character variable, you can write,

```
alpha = fgetc (fp);
```

This means the character pointed to by `fp` is read and assigned to `alpha`. A summary of header files and functions are given in Annexure 3. You can also go to the help screen of the `'C'` language system to get more details as well as search for help on any of the library functions. The help screen gives the syntax of the functions and also provides examples in which the function or command is used. Even after reading this book or any other book on 'C', you will not be able to use all the functions. Hence, the best way is to take the help from the help screen whenever other functions are to be used.

NOTES

`fgetc()` reads the character pointed to by `fp`. It then increments `fp` so that `fp` points to the next character. We can keep on incrementing `fp` till the end of file, i.e., end of data is reached. When a file is created in the text mode, the system inserts a special character at the end of the text. Therefore, while reading a file, when the last character has been read and the end of the file is reached, EOF is returned by the file pointer. The following program reads one character at a time till EOF is reached from an already created text file, `ss.doc`. The program is implemented using the `do...while` statement.

Example 3.41: `reading characters from a file */`

```
#include <stdio.h>
int main()
{
    int alpha;
    FILE *fp;
    fp=fopen("ss.doc", "r");
    do
    {
       alpha=fgetc(fp);
       putchar(alpha);
    } while(alpha!=EOF);
    fclose (fp);
}
```

Result of the program

Since file `ss.doc` is read, the output will be same as Example 3.39, if no change has been made in the file. If we were to read from a binary file, EOF may not be recognized. Therefore, a counter can be set up to read a predefined number of characters as given in the previous examples.

Processing a Data File

File Copy

File copy can be achieved by reading one character at a time and writing to another file either in the write mode or the append mode. Here it is proposed to read from a file and write to two different files, one in the write mode and another in the append mode. This means we have to open three files in the following manner:

```
FILE * fr, *fw; *fa;
```

You can assign three file pointers as given above. Three files are then opened. You can use any name for the file pointers and there may be as many file pointers as the number of files to be used. The program is given below:

Example 3.42: reading from file ss, writing to file ws and appending to file as, all at a time*/

```
#include <stdio.h>
int main()
{
    int alpha;
    FILE *fr,*fw, *fa;
    fr=fopen("ss.doc", "r");
    fw=fopen("ws.doc", "w");
    fa=fopen("as.doc", "a");
    do
    {
       alpha=fgetc(fr);
       fputc(alpha, fw);
       fputc(alpha, fa);
       putchar(alpha);
    }while(alpha!=EOF);
    fclose (fr);
    fclose (fw);
    fclose (fa);
}
```

NOTES

After opening the three files, `alpha()` gets the character, which is written to both the files using `fputc()`, and the character is also displayed on the screen. This is continued till EOF is received in `alpha` from `ss.doc`, the source file.

Finally, the files are closed. Verify that our program has worked alright.

Since, we are also writing to the monitor, in addition to writing and appending to files, the program output appears as follows.

Result of the program

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78
79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97
98 99
```

There are some more mode specifiers with `fopen` like `r+`, `w+` and `a+`, which are given in the table below:

Mode Specifier	Purpose
`r+`	Opens an already existing file for reading and writing.
`w+`	Opens a new file for writing as well as reading.
`a+`	Opens an already existing file for appending and reading.

NOTES

Line Input/Output

We have discussed writing to and reading from a file, one character at a time, using both the unformatted and formatted I/O for the purpose. We can also read one line at a time. This is enabled by the `fgets()` function. This is a standard library function with the following syntax:

```
Char * fgets (char * buf, int max line, FILE * fp);
```

`fgets()` reads the next line from the file pointed to by `fp` into the character array `buf`. The line means characters up to `maxline` –1, i.e., if `maxline` is 80, each execution of the function will permit reading of up to 79 characters in the next line. Here 79 is the maximum, but you can even read 10 characters at a time, if it is specified.

```
fgets(alpha, 10, fr);
```

Here `alpha` is the name of buffer from where 10 characters are to be read at a time. The file pointer `fr` points to the file from which the line is read, and the line read is terminated with NULL. Therefore, it returns a line if available and NULL if the file is empty or an error occurs in opening the file or reading the file.

The complementary function to `fgets()` is `fputs()`. Obviously `fputs()` will write a line of text into the file. The syntax is as follows:

```
int fputs (char * buf , file * fp );
```

The contents of array `buf` are written onto the file pointed to by `fp`. It returns EOF on error and zero otherwise. Note that the execution of `fgets()` returns a line and `fputs()` returns zero after a normal operation.

The functions `gets()` and `puts()` were used with `stdio`, whereas `fgets()` and `fputs()` operate on files.

We can write a program to transfer two lines of text from the buffer to a file and then read the contents of the file to the monitor. This is shown in Example 3.43.

Example 3.43: `writing and reading lines on files */`

```
#include <stdio.h>
#include<string.h>
int main()
{
    int i;
    char alpha[80];
    FILE *fr,*fw;
    fw=fopen("ws.doc", "wb");
    for(i=0; i<2; i++)
    {
       printf("Enter a line up to 80 characters\n");
```

```
        gets(alpha);
        fputs(alpha, fw);
    }
    fclose(fw);
    fr=fopen("ws.doc", "rb");
    while
    (   fgets(alpha,20, fr)!=NULL)
        puts(alpha);
        fclose (fr);
}
```

Note carefully the `fgets()` statement. Here `alpha` is the buffer with a width of 80 characters. Each line can be up to 80 characters and two lines are entered through `alpha` to `ws.doc`. Later on, 20 characters are read into `alpha` at a time from same file till NULL is returned.

Result of the program

```
Enter a line upto 80 characters
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Enter a line upto 80 characters
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaabbbb
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb
bb
```

More than 20 numbers of `a & b` were written on to the file. However, since we have specified reading 20 characters at a time. The output appears in 6 lines. Had we specified reading more characters at a time, the number of reads would have reduced. You can try this yourself.

Thus you can read and write one line at a time.

Use of the Command Line Argument

This can be used to copy a file to another file. Assume that the first named file is to be copied to the second named file. We may write a program and convert it into an executable file, specifying the argument in the DOS command line.

We may specify as follows at the C> prompt:

```
C > prgname . exe f1.cpp f2.cpp.
```

This means that we want to copy the contents of `f1.cpp` to `f2.cpp`. Here the number of arguments are 3, and therefore `argc` will contain 3.

```
*agrv[0] = prgname.exe
```

```
*agrv[1] = f1.cpp - source to copy from
*agrv[2] = f2.cpp - file where to be copied
```

A character at a time is to be fetched from `f1.cpp` and put into `f2.cpp`.

NOTES

Personal File of an Employee

A menu-based program to create employee records on file and calculate the age of any employee on date is given below:

Example 3.44:

```
Create a Personal File for Employees &
calculate the age of any employee ON DATE*/
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
typedef struct
{
    char name[40];
    char code[5];
    char dob[9];
    char qual[40];
}employee;
    FILE *fp;
    struct date today;
int main()
{
    int create_emp();
    int calc_age();
    int ret,ch,onscrn=1;
    getdate(&today);
    printf("Today's Date Is %d/%d/%d\nIs It O.K :",
       today.da_day,today.da_mon,today.da_year);
    scanf("%c",&ch);
    onscrn=1;
    while(onscrn)
    {
       clrscr();
       printf("1: Create Employee Data File\n");
       printf("2: Calculate Age Of Employee\n");
       printf("3: Exit From Program\nEnter Your Choice
:");
```

NOTES

```
        scanf("%d",&ch);
        switch(ch)
        {
           case 1:
              create_emp();
              break;
           case 2:
              calc_age();
              break;
           case 3:
              onscrn=0;
              break;
        }
    }
    fclose(fp);
}
int create_emp()
{
    employee emp1;
    int i,n;
    fp=fopen("emp.dat","a");
    clrscr();
    printf("How Many Employees :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
       clrscr();
       printf("Employee %d Details :\n",i+1);
       printf("\n\nEmployee Name :");
       scanf("%s",&emp1.name);
       printf("Employee Code :");
       scanf("%s",&emp1.code);
       printf("Date Of Birth :(dd/mm/yy)");
       scanf("%s",&emp1.dob);
       printf("Qualification :");
       scanf("%s",&emp1.qual);
       fprintf(fp,  "%40s%5s%9s%40s\n",  emp1.name,
emp1.code,emp1.dob, emp1.qual);
    }
    fclose(fp);
```

NOTES

```
    return(0);
}
int calc_age()
{
    int ret,nyob,age,llfound=0,onscrn=1;
    employee emp1;
    char nam[40],*sear,*ori;
    char yob[5];
    fp=fopen("emp.dat","r");
    clrscr();
    printf("Employee Name To Search :");
    scanf("%s",nam);
    sear =strlwr(nam);
    while(onscrn)
    {
       ret=fscanf(fp, "%40s%5s%9s%40s\n", emp1.name,
emp1.code, emp1.dob, emp1.qual);
       if(ret==EOF)
       {
          onscrn=0;
          continue;
       }
       ori=strlwr(emp1.name);
       if(strcmp(sear,ori)==0)
       {
          clrscr();
          printf("Employee Name :%s\n",emp1.name);
          printf("Employee Code :%s\n",emp1.code);
          printf("Date of Birth :%s\n",emp1.dob);
          printf("Qualification :%s\n",emp1.qual);
          strcpy(yob,"19");
          strncat(yob,emp1.dob+6,2);
          yob[4]=0;
          nyob=atoi(yob);
          age = today.da_year - nyob;
          printf("Age of Employee :%d\n",age);
          getch();
          onscrn=0;
          llfound=1;
       }
    }
```

```
    fclose(fp);
    if (!llfound)
    {
       printf("%s Not found in emp.dat\n",nam);
       getch();
    }
    return(0);
}
```

Result of the program

```
Employee Name     :   saravanan
Employee Code     :   06
Date of Birth     :   02/06/63
Qualification     :   MBA
Age of Employee   :   36
```

To understand the program, read the following:

The function <dos.h> is included. Look at the online help and see what it does. You will find that it defines various constants and declarations needed for DOS and 8086 specific calls. You can use it to get the system date to calculate the age of the employee. After the system date is confirmed, the menu appears. If you choose 1, it calls `create_emp` and asks for the number of employees. Then it accepts the records of a specified number of employees. The employee record is a structure.

After creating the records, you can opt to calculate the employee's ages by entering 2. This invokes the function `calc_age`. The function asks for the name of employee it must search for. If the name matches, the age will be calculated and displayed. The records are written in the append mode, so you will not lose the records. Note that the structures are written to the file using `fprintf()` and read from the file using `fscanf()`. Note also that `date` is a structure with three members `da_day, da_mon, da_year.`

This example has demonstrated that structures can be written to a file.

3.23 STRUCTURED PROGRAMMING CONCEPTS

A procedural program has a list of instructions telling computer what to do, step-by-step. It may instruct the computer for opening a file, then reading a number, multiplying it to another number and displaying the result. Programming units have main block with other blocks for programs, functions, precedures, subrouting etc. It also gives scope for files and may include modules and/or libraries.

Procedural programming is most convenient when program is small. It is a natural way of instructing a computer to perform a task. Language of computer

NOTES

NOTES

processor, the machine code, is itself procedural. Thus, translating high-level procedural programs into machine readable code. A procedural program can split a lengthy list containing many instructions into shorter ones by using functions.

Procedural programming is also an imperative programming specifying steps to solve a problem. But such programming environment makes use of 'procedure call'. These procedures are in form of functions, methods, subroutines or routines. Steps are enumerated as these contain many computational steps in series. A procedure can be called from any where in the program body during execution of a program.

Concept of modularity as well of scope of program codes form the base of procedure programming. A main procedural program contains modules. These modules are also called package(s) or unit(s).

Structured programming is in fact, a subdiscipline of procedural programming. Small programs are better written in procedure-oriented programming. Bigger and complex programs are better written in structured manner. Structured programming is also known as modular programming. This enforces a logical structure for the program, making it more efficient, easy to understand and easier to modify.

Languages like ALGOL, ADA, PL/1, Pascal, and dBASE have features that enforce a logical structure for the program that is to be written. Structured programming employs top-down design model. In this design model, developers create a structure for the program structure by breaking it into separate sections and subsections. A modular approach is followed. Similar functions are coded as a separate module or submodule. Thus code can be easily loaded and since each module performs one specific task, these are reused in other programs too where similar task is performed.

These modules are tested individually, before integrating with other modules to create the overall program structure. These modules are arranged in a hierarchical manner using looping constructs, available in almost every programming language. These are, 'for', 'repeat' and 'while'.

Procedural programming discourages GOTO statement. But in structured programming 'GOTO' is used.

There are different methodologies or techniques to create structured programs. There are three most common approaches:

1. Structured programming of Edsger Dijkstra: In this methodology a program logic results in a structure made of few similar sub-structures. This makes it easy to understand.
2. Derived view of Dijkstra: This view advocates splitting of whole program into many sub-sections with each having one entry point. But there is difference of opinion and a single exit point is considered better.

3. Data Structured Programming or Jackson Structured Programming: This is based on alignment and synchronization of program structures with data structures. This approach applies fundamental structures of Dijkstra.

NOTES

Low-level Structured Programming

Structured programs, at low level, contain simple and hierarchical program flow structures having sequence, selection, and repetition.

Sequence is an ordered execution of statements. Selection is one out of many statements whose execution depends on the condition and state of the program. Keywords used are, `if..then..else..endif`, `switch`, or `case`. In repetition a statement is executed repeatedly until the program reaches a certain condition or state of operations that are applied to elements of a set. Keywords for this are: `while`, `repeat`, `for` or `do..until`. These are looping statements. It is recommended that every loop has one entry point and few approaches also advocate only one exit point. Few languages enforce this condition.

A language is 'block-structured' if it has a syntax that encloses structures between bracketed keywords. For example, an if-statement bracketed by `if..fi`. This is done in ALGOL 68. In PL/1 , a code section is bracketed by `BEGIN..END`. A language is 'comb-structured' if syntax tells to enclose structures inside an ordered series of keywords. Such a language has multiple structure keywords and each keyword define separate sections inside a block. In ADA, a block is a 4-pronged comb with keywords, `DECLARE`, `BEGIN`, `EXCEPTION`, `END`. The `if`-statement is a 4-pronged comb with keywords `IF`, `THEN`, `ELSE`, `END IF`.

Structured programming is mostly associated with a 'top-down' approach to design. This is based on work-breakdown structure.

Structured programming can be done in any programming language. It is always advisable to use a procedural programming language. After 1970 structured programming gained popularity as a technique of programming. Due to this shift in programming paradigm, new procedural programming languages have features that have encouraged structured programming.

Procedures and Modularity

Modularity is desirable in large and complicated programs. Inputs are usually specified syntactically in the form of *arguments* and the outputs delivered as *return values*.

Scoping is a technique that keeps procedures modular as it prevents access to the variables of one procedure by other. Procedures which are less modular are used in small programs. These programs interact with a large number of variables in the execution environment.

Procedures are convenient means to make pieces of code reusable. These codes may be written as programming libraries. It has the ability to create a simple interface which is self-contained and can be reused.

NOTES

Basic Control Structure

In computer science **control flow** tells about an order of execution of program instructions in an imperative programming paradigm.

In such programming language execution of a **control flow statement** results in a choice between two or more paths. In non-strict functional languages, functions and language constructs achieve same result, but they control flow statements.

Control flow statements of different languages are different, and these are categorized by their effects:

- Unconditional branch or jump
- Execution of statements only on meeting some condition (i.e., conditional branching)
- Execution of statements zero or more times, on meeting some condition (i.e. loop - same as conditional branch)
- Execution of distant statements, returning flow of control (subroutines)
- Halting program, stopping further execution (unconditional halt).

Interrupts and signals are known as low-level mechanisms that altors flow of control, similar to a subroutine. This normally occurs in response to some external event which may occur asynchronously.

In low level programming, control flow instructions usually alters program counter. In some CPUs only control flow instructions result in conditional or unconditional branching which are also termed jumps.

Primitives

a. Labels

A label denotes an explicit name or number for a fixed location within a source code, referenced by control flow statements that appears somewhere in the source code. Label marks a location inside the source code and has no other effect.

Some languages use line numbers a named label instead of Fortran and BASIC, whole numbers use at the beginning of each line of text in source code. Such languages put constraint on line numbers to increase in value in and these need not be consecutive. For example, in BASIC:

```
10 LET X = 3
20 PRINT X
```

In languages such as C and Ada a label is used as an identifier, normally appearing at the start of a line followed by a colon. For example, in C:

Result: printf (“The operation was successful.\n”);

b. Goto

The **goto** statement is most basic that transfers control unconditional by.

Keyword may be written in upper or lower case depending on implementation of the language normally it is written as:

```
goto label
```

When a goto statement is used next statement is executed from the indicated label.

Goto statements is discouraged by many computer scientists, notably Dijkstra.

c. Subroutines

The terminology for subroutines is not standard and it varies. Alternatively subroutines are known as procedures, routines, functions or methods.

In the early stage of development in1950s, computer memories used to be very small hence subroutines were used for reducing program size. For this reason small codes were written and reused at many places in the program.

These days, subroutines make a program more structured, and provide modularity.

Minimal structured control flow

In May 1966, Böhm and Jacopini showed that program with **goto**s could be transformed into a goto-free form using choice IF THEN ELSE and loops such as WHILE condition DO xxx. At a later stage authors suggested the use of loops in place of choice use of Bollean variables with looping and choice statements using true/false flags could make program totally goto-free.

Such minimization is possible, but it does not mean it is desirable in all circumstances. But Böhm and Jacopini had shown that it was possible to make programs goto-free. Research has shown that control structures having one entry and one exit easier in comparison to any other form. They can be used anywhere in a program without disrupting the control flow.

Control structures in practice

Most programming languages use control structures with an initial keyword indicating type of control structure involved. Languages then divide as to whether or not control structures have a final keyword. Given below are language that have no final keyword or have to final keyword.

- No final keyword: Algol 60, C, C++, Haskell, Java, Pascal, Perl, PHP, PL/I, Python, PowerShell. Such languages require ways of grouping statements together:
  - Algol 60 and Pascal : begin ... end
  - C, C++, Java, Perl, PHP, and PowerShell: curly brackets { ... }
  - PL/1: DO ... END
  - Python: uses indentation level (see Off-side rule)
  - Haskell: either indentation level or curly brackets can be used, and they can be freely mixed

NOTES

- Final keyword: Ada, Algol 68, Modula-2, Fortran 77, Visual Basic. The forms of the final keyword vary:
  - Ada: final keyword is end + *space* + initial keyword e.g. if ... end if, loop ... end loop
  - Algol 68: initial keyword spelled backwards e.g. if ... fi, case ... esac
  - Fortran 77: final keyword is end + initial keyword e.g. IF ... ENDIF, DO ... ENDDO
  - Modula-2: same final keyword end for everything
  - Visual Basic: every control structure has its own keyword. If ... End If; For ... Next; Do ... Loop

Loops

A loop contains a sequence of statements, which is specified once but used several times in succession. The code "inside" the *body* of the loop, shown below as *xxx* is executed a specified number of times, until some condition is met.

a. Count-controlled loops

Most programming languages use costructs to repeat a loop a specified number of times. If N is less than 1 in examples given below then the body is skipped completely, and if N = 1 the body is executed just once. In most cases counting goes downwards instead of upwards and step sizes other than 1 are used.

```
FOR I = 1 TO N
    xxx
NEXT I
```

```
for I := 1 to N do begin
        xxx
end;
```

```
DO I = 1,N
    xxx
END DO
```

```
for ( I=1; I<=N; ++I ) {
        xxx
    }
```

Many programming languages, use integers for a reliable count-controlled loop. Use of floating point numbers is not reliable due to hardware constraints. A loop such as

```
for X := 0.1 step 0.1 to 1.0 do
```

might be repeated 9 or 10 times that depends on rounding errors and/or the hardware and/or the compiler version.

b. Condition-controlled loops

Most programming languages have constructs that repeats a loop until some condition changes. Some variations test conditions at the start of the loop, whereas others test at the end. In former case the body may be skipped completely, but in latter case body is executed at least once.

```
DO WHILE (test)
    xxx
LOOP
```

```
while (test) {
```

```
repeat
 xxx
until test;
```

```
do
```

```
    xxx                                    xxx
}                                          while (test);
```

c. Collection-controlled loops

NOTES

Several programming languages (e.g. Ada, Smalltalk, Perl, Java, C#, Visual Basic, Ruby, Python, JavaScript) special constructs allowing implicit looping through every element of an array, or every member of a set or collection.

```
someCollection do: [:eachElement |xxx].
 foreach someArray { xxx }
Collection<String> coll; for (String s : coll) {}
 foreach (string s in myStringCollection) { xxx }
$someCollection | ForEach-Object { $_ }
```

d. General iteration

General iteration constructs in C is **for** statement and in Common Lisp it is **do** used for expressing any of the above loops, as well as others. Looping is also possible over a number of collections in parallel. Specific looping construct is usually preferred over general iteration construct, as it makes the purpose of the expression more clear.

e. Infinite loops

In certain situations, infinite loops are desired. It is desired that looping should continue and terminate only when some exception, such as error occurs. An event-driven program (such as a server) may be required to loop forever handling events, stopping only when the process is killed by the operator.

An infinite loop is mostly due to a programming error in a condition-controlled loop, wherein the loop condition is never changed within the loop.

f. Continuation with next iteration

Sometimes it is desired to skip the remainder of the loop body and continue with the next iteration. Some languages provide statements such as **continue**, **skip**, or **next** for this. This, prematurely terminates the innermost loop body, normal with next iteration. If the iteration is the last one in the loop, it terminates the entire loop.

g. Redo current iteration

Some languages, such as Perl and Ruby, provide a **redo** statement that restarts the current iteration from the beginning.

h. Early exit from loops

In a count-controlled loop that searches through a table, it is desired to stop when the desired item is found. Statement such as **break** or **exit** are provided by some programming languages.Using break or exit the current loop is immediately terminated and control is transferred to the statement immediately following that loop. Things can get a bit messy while searching a multi-dimensional table using nested loops.

NOTES

The following example is done in Ada which supports both *early exit from loops* and *loops with test in the middle*. Both these features have similarity but codes are different. Codes for *early exit* needs combination with an **if** statement whereas a *condition in the middle* is a self contained construct.

```
with Ada.Text IO;
with Ada.Integer Text IO;
procedure Print_Squares is
    X : Integer;
begin
    Read_Data : loop
        Ada.Integer Text IO.Get(X);
    exit Read_Data when X = 0;
        Ada.Text IO.Put (X * X);
        Ada.Text IO.New_Line;
    end loop Read_Data;
end Print_Squares;
```

Python provides support for conditional execution of code and this depends on whether a loop was exited early using a break statement or not by using an else-clause with the loop. For example,

```
for n in set_of_numbers:
    if isprime(n):
        print "Set contains a prime number"
        break
else:
    print "Set did not contain any prime numbers"
```

The else clause in the example above is attached to the 'for' statement, and not to the inner if statement. The for and while loops of Python support an else clause and is executed only if early exit of the loop did not occur.

Structured non-local control flow

Many programming languages, favouring dynamic styles of programming, have constructs for **non-local control flow**. Using such constructs execution jumps out of a given context and then resume at some predeclared point. *Common non-local control constructs are, exceptions, conditions*, and *continuations*.

a. Conditions

PL/I contains 22 standard conditions (e.g. ZERODIVIDE SUBSCRIPTRANGE ENDFILE) which can be RAISEd and which can be intercepted by: ON *condition* action. Programmers can also define and use their own named conditions.

Like *unstructured if* only one statement can be specified, so in many cases, a GOTO is needed for deciding the location from whole flow of control should resume.

Some implementations need substantial overhead in both space and time (especially SUBSCRIPTRANGE), so these conditions are avoided by many programmers.

Common Syntax examples:

ON *condition* **GOTO** *label*

NOTES

b. Exceptions

Modern languages provide a structured construct for handling exception and do not rely on the use of GOTO:

```
    try {
xxx1                          // Somewhere in here
xxx2                          // use: ''throw'' someValue;
xxx3
} catch (someClass& someId) {     //  catch  value  of
someClass
                                   actionForSomeClass
} catch (someType& anotherId) {     // catch value of
someType
                                 actionForSomeType
} catch (...) {         // catch anything not already caught
actionForAnythingElse
}
```

Catch can have many numbers and varieties of clauses for use. In D, Java, C#, and Python a finally clause is added to the try construct. It does not matter how the control comes out of the 'try' clause, the code inside the finally clause is guaranteed to execute. This is useful when writing code that must relinquish an expensive resource when finished processing:

```
    FileStream stm = null;    // C# example
    try {
          stm  =  new  FileStream  ("logfile.txt",
FileMode.Create);
         return ProcessStuff(stm);   // may throw an
exception
    } finally {
        if (stm != null)
            stm. Close();
    }
    Since this pattern is fairly common, C# has a special
syntax:
    using (FileStream stm = new FileStream ("logfile.txt",
FileMode.Create)) {
```

```
        return ProcessStuff(stm);      // may throw an
exception
    }
```

NOTES

On leaving the block, named using, the compiler guarantees the release of the object stm. With statement of Python and block argument to File.open of Ruby are used to get similar effect.

All these languages provide definition for standard exceptions and circumstances under which these are thrown. Users may throw exceptions of their own. C++ and Python permit users to throw and catch of almost any type.

In case there is no catch that matches a particular throw, then control moves back through subroutine calls and/or nested blocks to find a matching catch or reaching the end of the main program and at this, point program is forcibly stopped giving suitable error message.

Proposed control structures

In a spoof Datamation article in 1973, a suggestion was put by R. Lawrence Clark that the COMEFROM statement could replace GOTO and this provides some interesting examples. This found actual implementation in programming language INTERCAL, which is a language designed to make programs as obscure as possible.

Donald Knuth in his article 'Structured Programming with go to Statements', published in 1974, identified two situations that remained uncovered by the control structures listed above. He produced examples of control structures capable of handling these situations. These constructions inspite of their utility failed to find their way into mainstream programming languages.

Loop with test in the middle

Dahl proposed this in 1972.

```
loop                                loop
    xxx1                                read(char);
while test;                     while not atEndOfFile;
    xxx2                                write(char);
repeat;                             repeat;
```

If *xxx1* is omitted we find a loop that tests at the top. If *xxx2* is omitted we find a loop that tests at the bottom. If **while** is omitted, an infinite loop is found. Hence this single construction is capable of replacing several constructions in most programming languages. A possible variant is one that allows more than one **while** test within the loop, but using **exitwhen** covers this case in a better way.

Multiple early exit/exit from nested loops

Zahn proposed this in 1974. Following codes show its modified version.

```
exitwhen EventA or EventB or EventC;
    xxx
```

```
exits
    EventA: actionA
    EventB: actionB
    EventC: actionC
endexit;
```

NOTES

exitwhen is used for specifying events that is likely to occur within *xxx* and their occurrence is indicated by use of name of the event as a statement. On occurrence of some event, relevant action is carried out, and then control moves immediately after **endexit**. Such a construct provides very clear separation between determining situation that applies, and the action corresponding to that situation.

exitwhen has conceptual similarity with exception handling, and exceptions or similar constructs find use in many languages for this purpose.

Following simple example searches a two-dimensional table for a particular item.

```
exitwhen found or missing;
    for I := 1 to N do
        for J := 1 to M do
            if table[I,J] = target then found;
    missing;
exits
    found:   print ("item is in table");
    missing: print ("item is not in table");
endexit;
```

Structured programming

Structured programming may be viewed as a subset or subdiscipline of procedural programming. It is a major programming paradigm. Its popularity is due to the fact that it reduces reliance on GOTO statement.

Historically, there are different methodologies or techniques adopted for structured programming. Three most common techniques are:

1. Structured programming of Edsger Dijkstra in which logic of a program has a structure made up of limited number of similar sub-structures. This reduces efforts to understand a program as each substructure is small and entire program is made up of many such similar structures.
2. A view that has been derived from Dijkstra that advocates splitting a program into many sub-sections with each having one entry point but there is another view that support the concept of one exit point instead of entry points.
3. Data Structured Programming or Jackson Structured Programming, which is based on alignment of program structure with data structures. This approach used fundamental structures proposed by Dijkstra, but used high-level structure to be modeled on the underlying data structures. There are at

NOTES

least three major approaches to data structured program design proposed by Jean-Dominique Warnier, Michael A. Jackson, and Ken Orr.

The latter two are more common. Years after the concept given by Dijkstra in 1969, object-oriented programming (OOP) was developed that could handle complex programs.

Low-level structure: At low level, structured programs consisted simple program flow that followed hierarchical structures. These made use of sequence, selection, and repetition:

- 'Sequence'indicates an ordered execution of statements.
- 'Selection' means execution of one statement from a number of statements depending on the state of the program usually expressed using keywords if..then..else..endif, switch, or case.
- 'Repetition' means execution of statement when program reaches a certain state or operations that applies to every element of a collection. This is mostly expressed using keywords while, repeat, for or do..until. Often it is preferred that each loop should only have one entry point and also only one exit point. Few languages enforce this.

Dijkstra's original Guarded Command Language put emphasis on unity of these structures using a syntax that completely encloses the structure, as in if..fi. This is not in C, and risk of misunderstanding and incorrect modification is associated.

A language is known as 'block-structured' when there is a syntax to enclose structures between bracketed keywords. For example, an if-statement bracketed by if..fi as in ALGOL 68, or a code section bracketed by BEGIN..END, as in PL/I. However, a language is 'comb-structured' having a syntax for enclosing structures within an ordered series of keywords. A 'comb-structured' language has multiple structure keywords for defining separate sections within a block, analogous to multiple teeth or prongs in a comb separating sections of the comb. For example, in Ada, a block is a 4-pronged comb with keywords DECLARE, BEGIN, EXCEPTION, END, and the if-statement in Ada is a 4-pronged comb with keywords IF, THEN, ELSE, END IF.

Design: Structured programming often follows a 'top-down' approach but not always.

Structured programming languages

Structured programming is possible in any programming language, although a procedural programming language is preferred. After 1970 structured programming gained popularity and, most of the new procedural programming languages included features for encouraging structured programming. Better known structured programming languages are ALGOL, Pascal, PL/I and Ada.

Towards the end of 20th century most of the computer scientists favoured the concepts of structured programming. High-level programming languages such

as FORTRAN, COBOL, and BASIC that originally lacked features of structured programming now have such features.

Common deviations: Exception handling

NOTES

Multiple entry to a subprogram is never considered a good practice, yet multiple exists are mostly used in a subprogram. This is to ensure that there is exit without much work when there are circumstance that do not allow execution to continue further.

A simple procedure to read data from a file and to process it, is being given below:

```
open file;
while (reading not finished) {
  read some data;
  if (error) {
    stop the subprogram and inform rest of the program
about the error;
  }
}
process read data;
finish the subprogram;
```

The 'stop and inform' is achieved by throwing an exception, second return from the procedure, with a label loop break, or a goto. Since this procedure contains two exit points, rules of structured programming of Dijkstra is not obeyed. Single point exit rule, will make the coding cumbersome in this case. If possible error conditions, are many, having different rules for clean up, a procedure with single-exit-point will become very difficult to read and understand. This may be more difficult to read and understand than program when control is handled by goto statements. Structural programming without single-exit-point, in such cases, will give very clean and readable code.

For this reason, many programming languages adopted multiple points of exit in structural programming. C allowed multiple exits using 'continue', 'break', and 'return'. Other languages, after C have also adopted 'break' with a label and exceptions.

State machines

Parsers and communications protocols, define many states following each other that it is not easily reduced to basic structures. These systems can be structured by making each state-change a separate subprogram using a variable for indicating active state. Some programmers implement state-changes using a jump to the new state.

Object-oriented comparison

Design of language during sixties was mostly small and based on examples of text book programs, but this changed when large programs were written. Most common

statements in small programs, are assignment statements, whereas in large programs having more than 10 k lines, most common statements are procedure-calls to subprograms.

NOTES

Small programs are handled by writing codes for hierarchy of structures. In large programs, organization contains network of structures. Undue emphasis on hierarchical structuring for data and procedures produces cumbersome code having large amounts of 'tramp data.' For example, a program for displaying text, allowing dynamic change of font size of the entire screen becomes very combursome if font-size data is passed through a hierarchy. As an alternative to this, a system may be used for controlling font data using functions to retrieve data from a common area that is controlled by font-data sub-system.

FORTRAN uses labelled COMMON-blocks for separating global program data into subsystems for permitting program-wide, network-style access to data, such as font-size, by specifying particular COMMON-block name. Confusion is likely in FORTRAN by coding alias names and changing data-types while referencing the same labelled COMMON-block yet mapping alternate variables for over laying the same memory region. Labelled-COMMON concept was extremely valuable while organizing huge software systems. This has led to the use of object-oriented programming for defining subsystems of centralized data controlled by use of accessor functions.Making changes into other data-types were performed by explicitly converting, or casting, data from original variables.

Global subprogram names were found misleading in comparision to global variables or blank COMMON, and subsystems were kept under limit into subprogram names, such as naming using unique prefixes or using name, as used in Java package.

The object-oriented is flexible, as it separates a program into a network of subsystems own data, algorithms, or devices in the whole program, accessible only by specifying named access to the subsystem object-class.Object-oriented programming required a call-reference index for tracing subsystems or classes that are accessed from other locations.

Modern structured systems move toward 'event driven' architectures. Procedural events are designed as independent tasks.

Structured programming led to the concept of object-oriented programming. Memory leaks a program in causes consumption of huge amount of memory and this is due to failure is observing a single exit-point in a subprogram needing memory deallocation.

Structured programming led to a recognition of top-down approach in branching.

Various concepts of structured programming helps in understanding many facets of object-oriented programming.

Benefits of Structured Programming

Structural programming is also called modular programming which is based on work break down structure. It offers lot of benefits.

NOTES

1. Easy to Write

A modular design is more productive. Programmer or developer looks at the overall picture first and then focuses on details, one-by-one. Every module performs one task and this enables several programmers to work on a single, large program in which each works on a particular module that is different from the other. Structured programs take less time in writing when compared to standard programs.

If a procedure has been written for one program, then it can be reused in other programs too, if it performs the same task. Such a procedure can be used in other programs too and is reusable. Such a program can be adapted to similar tasks by making some changes.

Structured programming follows top down design. In this design developer starts with the complete item first and then breaks it down into smaller parts and sub-parts. Thus, a difficult task is broken down into several small tasks. This is also known as 'divide and conquer' strategy. Once a big task is divided into several small tasks, solving these individual pieces makes it easy to control these small programs independently until every step can easily be implemented. This is done with successive refinement.

2. Easy to Debug

Each procedure, in a program, performs just one task. Thus individual procedure can be checked. An unstructured program has a sequence of instructions, not grouped task wise. Such programs are cluttered with details and creates problem in understandability. A structured program, being modularized, it is easy to locate the fault and change or modify it, without the need of looking into the other parts of the program.

3. Easy to Understand

A modular design shows the relationship between the procedures. Procedure names are mostly, meaningful and a clear documentation identifies the task of each module. It is always advisable to give meaningful variable names so as to identify the purpose of each variable.

4. Easy to Change

In a large program which is procedural makes it difficult to understand when other programmers or developers are working on that. Since a structured program is self-documenting, it poses no difficulty for other programmer to understand it.

According to E. Dijkstra, hierarchical systems have a property that something considered as an undivided entity on one level is considered as a composite object on the next lowest level of greater detail. Thus natural grain of space or time, applicable at each level decreases by an order of magnitude when we shift our

attention to the next lower one. A wall can be understood as composed of bricks. Bricks can be thought of as composed of crystals and crystals made up of molecules, etc.

NOTES

Below we give an example of a small house that is to be made for a pet.

Pet House

We first see the overall image. A house is composed of floor, walls and roof. We move down and form details of each. A floor may be of something on which a person can stand. Walls will be front, back, left and write. Every wall will have top section, lower section and door section. Roof also will be left side and right side. Details are given below.

- **1. Floor or Base**
- **2. Walls**
  - 2.1 Back Wall
    - 2.1.1 Top section
    - 2.1.2 Lower section
  - 2.2 Left Wall
  - 2.3 Right Wall
  - 2.4 Front Wall
    - 2.4.1 Top section
    - 2.4.2 Door section
- **3. Roof**
  - 3.1 Left Side
  - 3.2 Right Side

Once each part has been detailed one may prepare all sections. These are described below. Join these sections using nails and glue as needed.

- **1. Floor or Base -** cut to size as specified and smooth edges
- **2. Walls**
  - 2.1 Back Wall
    - 2.1.1 Top section - cut triangular shape
    - 2.1.2 Lower section - cut square shape and glue and staple to upper part of wall
  - 2.2 Left Wall - cut to specified dimensions
  - 2.3 Right Wall - cut to specified dimensions
  - 2.4 Front Wall
    - 2.4.1 Top section - cut triangular piece to specified dimensions
    - 2.4.2 Door section - use jigsaw to cut U shape for the door in square shape, staple and glue to top section

- **3. Roof**
  - o 3.1 Left Side - cut to specified dimensions
  - o 3.2 Right Side - cut to specified dimensions

NOTES

A procedural program is a list of actions as input to computer. When program becomes large and complex, it becomes difficult to organize and control these. Also at a later stage when another programmer has to work on that, it becomes difficult to understand it. A structural programming that way is good since it is composed of small routines or tasks organized in structured manner that is called modular too.

After the concept of structured programming only, the concept of object oriented programming came. In procedural programming data is not controlled, but in object oriented programming data belongs to method and to access data, method has to be accessed. Structured program has great importance in managing large and complex programs.

Procedure Oriented Programming

Procedural programming, at times is used as a synonym for imperative programming. Procedures are also known as routines, subroutines, methods, or functions. These contain a series of steps for computation to be carried out. A procedure is called from any point during execution of a program. In a **procedural programming language** each step is defined precisely for performing a task. Benefits of a procedure programming are:

- Re-usability to code with copying.
- Easy tracking of program flow without using GO TO or 'JUMP' statements.
- Creation of structured or modular programs.

Procedures and modularity

Modularity is desirable in large and complex programs. Inputs have syntactic specification as *arguments* and outputs are delivered as *return values*.

Scoping is another technique for keeping procedures strongly modular. It prevents access to variables of other procedures unless there is explicit authorization for this.

Less modular procedures, are used, but it is for small or quickly written programs.

As a way to provide self-contained, simple interface and re-usable codes, procedures can utilize codes written by different people through programming libraries.

Features of Procedure Oriented Programming

1. A procedure oriented program consists of instructions in groups, known as functions. High level languages like Fortran, Pascal and 'C' are commonly known as procedure oriented programming languages.

NOTES

2. Programs are organised in the form of subroutines and the data items are freely accessible.
3. Data in procedure oriented language is open and can be accessed by any function.
4. Function overloading and operator overloading are not possible in procedure oriented language.
5. In procedure oriented languages local variable can be declared only at the beginning of the block.
6. Program controls are through jumps and calls to sub-routines.
7. Polymorphism, Encapsulation, and inheritance are not possible in procedure oriented languages.
8. For solving the problems, the problem is divided into a number of modules. Each module in procedure oriented language is a sub-program.
9. Data abstraction property is not supported by procedure oriented languages.

3.24 TOP-DOWN AND BOTTOM-UP DESIGN

The two classic ways used to solve the programming problem are known top down design and bottom up design.

Top down design

The top down design starts programming segment from the top point (starting point) in a big and vast project. The following statement is fit in this structure design.

'I am going to control this airplane; I will start with the cockpit layout.'

It basically implies the creation of dummy solution along with definite purpose and interrelationship. A term 'front panel object' is used to solve the programming problem from the top down techniques using data types, connector pane and terminated layout. It refers a different style of programming which starts with high-level description that breaks up the simpler pieces. The top down programming makes complication in testing phase. In this mechanism, stubs are used to remove the complication in programming. Stubs refer to procedures or functions providing suitable interface. These are not the kind of implementation of other piece of program. The function in fact works as abstract data structure. The implementation of data structure can be shared between modules and hence able to define and expose globally. Therefore, this technique creates unwanted dependencies in the various parts of application. The stubs are useful because it is used to test the high-level function as well as abstract data type without implementing the lower level functions or abstract data types. It supports in generating modules but not suitable in reusing these modules in applications. The top down design makes a sequence of simple procedures or functions that can be redefined. The main loophole of top

down design is that all the decisions are taken from starting of the project. The decision is assessed directly or indirectly that corresponds to high-level specification tending to change over time. This technique supports various abstract operations, such as seek, read and write the logical block to maintain toggle activity. The flowchart of top down design is shown in Figure 3.4:

NOTES

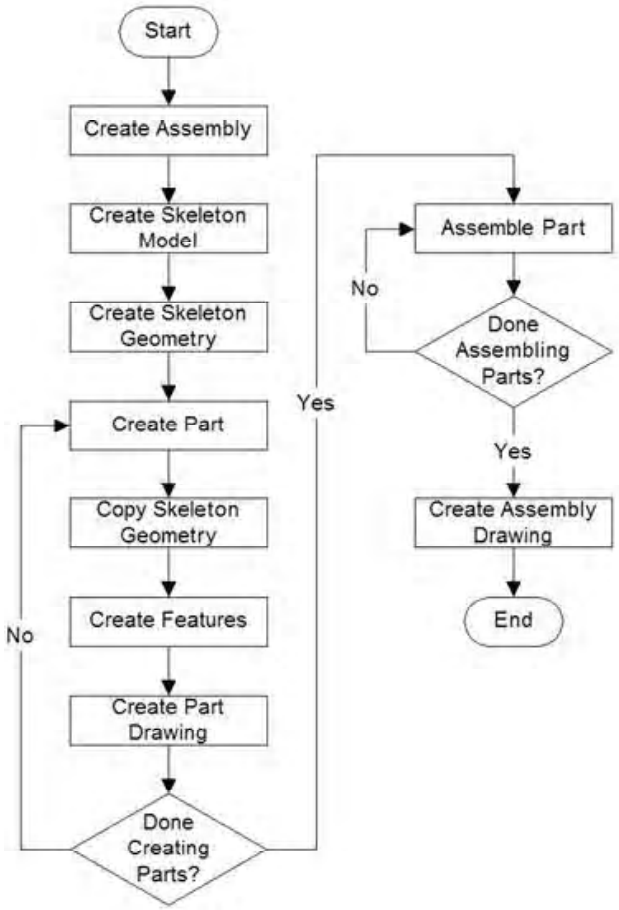


Figure 3.4 *Flowchart of Top Down Design*

In Figure 3.4, an example of top down design has been taken in which part design creates assembly which refers to skeleton model. Then skeleton geometry is created to capture all the interfaces among various components. So, the information is needed for starting point of the geometry. The assemble part is created if the skeleton geometry refers to individual part. Assembling parts make the location where interconnects and other boundaries occur. At last assembly drawing is created after checking the condition **'done assembling parts'** otherwise control goes to iterate the assemble part.

Bottom up design

The bottom up structured design starts to solve the low level bit manipulation, number crunching and timestamp problem. It implies to solve the programming part from lowest level. The lowest level corresponds with the right inputs and

NOTES

outputs linked with the program. It keeps the track of specified input and output maintaining the data dictionary. It incorporates a hierarchical table which provides a list of basic and prime data types. This technique refers the construction of applications that starts with primitive of programming language. It generates complicated features in programming if one application is linked with the other application. This is considered to be more useful than top down design because stubs are not needed to simplify the testing. The test functions are necessarily written as they are easy to understand and write. This technique is basically used in interactive programming environment, for example, Common Lisp. In the Common Lisp, abstract data types are built to use macros in constructing special forms. It uses lists to pass the arguments and maintains a standardized data structure which makes it abstract data type more instead of function. The small modules or pieces of programs are written from bottom end. It is more reusable in application specific language. The specified language must be simple to implement the entire class of applications. It is easy to maintain by adding new features in the applications. It lessens in delaying to take the final decision of exact functionality of applications. The computer languages **C** and Java construct abstract data types from primitive of the language or from the abstract data type.

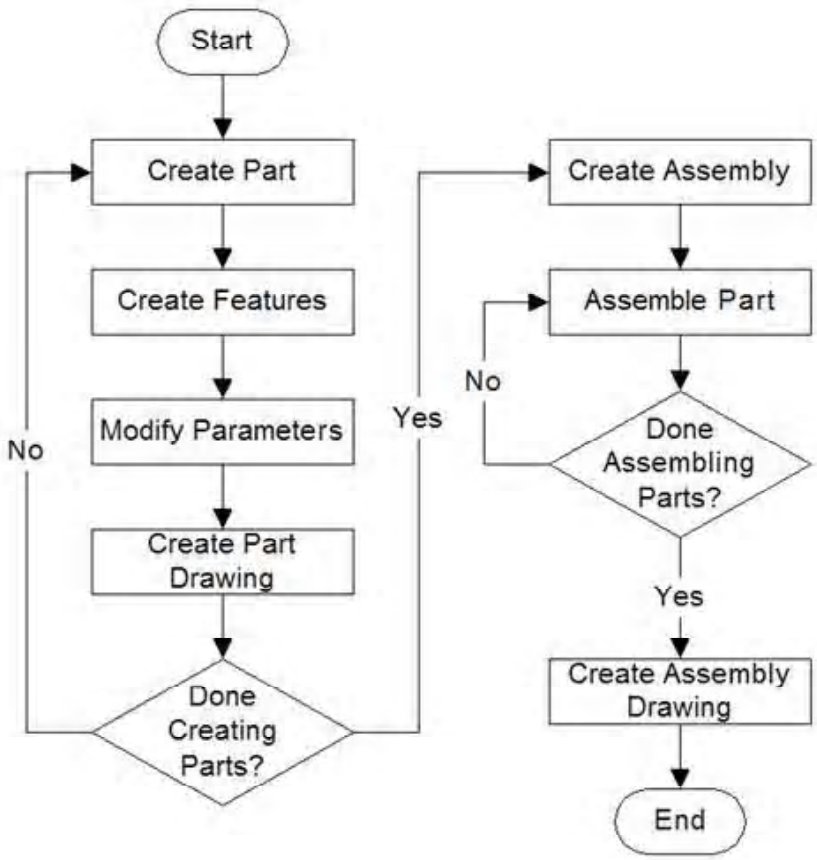


Figure 3.5 *Flowchart of Bottom up Design*

In the bottom up design parts refer to standalone design. It starts to assemble them and converted into assembly file. The drawing files and assembly files make changes. Each part keeps the drawing files and assembly files independently from each other. This is illustrated in Figure 3.5.

NOTES

The above two techniques are the two fundamental ways used in programming techniques. These concepts can be explained with the help of example. Let we take the designing part of Web browser. The top level designing of web browser includes the types of URL, such as **ftp:, http:, file:,** etc., the type of images, types of high level languages for example, JavaScript for validation checking, Java etc. Implementation layer involves main event loop as top level view. This loop is used in the design for collecting, dispatching or waiting for user action. The user action can be clicking on a specified Web link, typing text into the required text field.

3.25 DEVELOPMENT OF EFFICIENT PROGRAMS

Efficient program is developed by using the various tools of computer languages. The two critical factors of system unit are considered as memory and execution time. Efficient program can be developed if perfect coding techniques and accurate design as per client requirement take place in the programming part. The two prime factors decide the development of efficient program as follows:

Memory Requirement

The microcomputer environment keeps memory restriction which is mainly concerned to the programmer. The following steps take place to lessen the memory requirements:

- The program must be simple. It is the key factor which makes program efficient.
- The simple algorithm must be used.
- Arrays, pointers and strings must be declared with suitable memory address and correct sizes.
- Multidimensional arrays must be limited.
- The memory compression features must be evaluated and incorporated.

Execution time

The execution time is considered to be a prime factor in which the following concepts must be stepped out:

- The suitable algorithm is associated with the execution time directly.
- Fastest algorithm must be selected.
- Arithmetic and logical expressions must be simplified.
- Fast arithmetic operations are needed to make efficient of execution time.

NOTES

- Endless loops and multidimensional arrays must be avoided.
- Pointers must be included in the program because it frees the memory space after frees the memory space after successfully running of the program.

The `printf()` function in C source file uses 40960 bytes, where as C++ using `iostream` uses 65536 bytes. The C programs execution time can be measured by checking the clock cycles. These clock cycles are generated from the assembly code. A variable `CLK_TCK` is assigned in `time.h` header file. This header file contains a data type that is used to store the values by calling the `clock()` function. The variable `CLK_TCK` is not considered as standard instead `CLOCKS_PER_SEC` variable can be used. The execution time can be implemented on the following program:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int main()
{
     clock_t start=clock();
//Calculates the overhead from calling clock. The clock()
function is used to obtain the processing time of a
code.

        for ( i = 0; i < 10000000; i++ )
        rand();
//Generator of random number
printf  (  "%f\n",((double)clock()-  start)/
CLOCKS_PER_SEC);
// CLOCKS_PER_SEC is the number of processor clocks
per second.
return 0;
}
```

3.26 PROGRAM CORRECTNESS

Program correctness implements the programming code that is verified online. Online refers to execute coding part successfully on the system unit. It supports the correctness of module and also verifies contracts among modules. Program correctness is needed because testing of the programs is easy and it is helpful and user-friendly to the third-party users. In C, computer programming is done by two methods. The first method is known as mathematical induction and second is known as predicate transformer method. A program is correct **'iif'**

(immediate if that is if and only if). In program correctness method the programs do not contain any unexpected conditions, such as insufficient memory, overflow of words, running out of time etc. during the execution of program. The top down design suggests that a given problem can be broken down into many subtasks which push the statement into the final goal. The preservation of overall structure of the given problem helps to prove the correctness of the specified solution.

NOTES

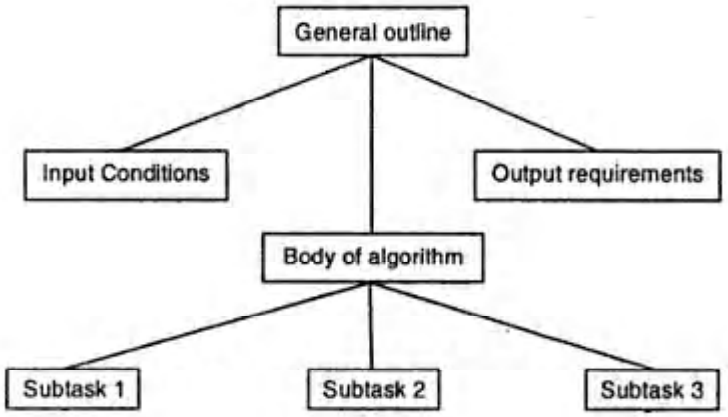


Figure 3.6 *Preservation of Structure Representing Subtasks*

In Figure 3.6, general outline of the program keeps input conditions and output conditions. The body of the algorithm continues between these two requirements and divided into many subtasks, such as subtask 1, subtask 2 and subtask 3. It shows a tree structure because subtasks may have many branches. A set of values are assigned to the parameter x is also known domain of x. For example,

$$GREATER(x, y) = \begin{cases} \text{true} & \text{if } x > y \\ \text{false} & \text{if } x \leq y \end{cases}$$

$$PRIME(x) = \begin{cases} \text{true} & \text{if } x \text{ is a prime number} \\ \text{false} & \text{otherwise} \end{cases}$$

$$SUM(x, y, z) = \begin{cases} \text{true} & \text{if } x + y = z \\ \text{false} & \text{otherwise} \end{cases}$$

In the previous example, there are three functions or modules named as GREATER(x, y), PRIME(x) and SUM(x, y). The function GREATER(x, y) is true if x is greater then y and false if x is less then and equal to y. In the same way, PRIME(x) is true if x is prime number and false if given value x is not a prime number. In the same way, in function SUM(x, y, z), the third parameter z is equal to x + y, otherwise false the function SUM. You can formalize the definition of each statement

Odd(x) means x is odd.

NOTES

Even(x) means x is even.

Div (a, b) means a divides b.

The major question behind program correctness is **"what is the program supposed to do"?**

The 'if-then' Rule is the solution.

The statement '**If a then B explains that B is executed if A is true and B is not executed if A is false.'**

Various interpretations for programs correctness are listed as below:

The program must not contain syntax errors. These errors are detected during translation time performed by the language processor.

- The program must not contain syntax errors. These errors are detected during translation time performed by the language processor.
- The program must not contain invalid operations. The invalid operations are automatically detected if the program is going to be executed.
- A program must contain a set of test data to generate the correct output.
- Program must contain validation checking and required conditions for erroneous inputs.
- For inputted values, an executable program produces correct or reasonable output.
- A program is considered to be incorrect if it does not produce the user's purpose.
- A program must not the global common variables and poor uses of functions.

Program correctness is taken place by resolving the program analysis and program synthesis. For example, take a simple C program that sums the total value as follows:

```
#include <sdio.h>      //Wrong header file is included
#define VALUE 5
void main()              //main() function is declared
{
            int i, //Statement is not terminated by ;
and sum value is not assigned as 0.
for (i=1, i<=VALUE; i++); //For loop will not work because
is terminated by ;
{
        sum = sum + 1;     //The statement must be taken
as sum=sum + i; for getting the sum value from 1 to 5
```

```
}
print("Sum = %d\n", sum); //The function must be printf()
that prints the resulted output.
getch();
}
```

NOTES

The above program can be corrected as per 'C' language syntax norm. The correct program is written in the following way:

```
#include <stdio.h> //Including header file
#define VALUE 5 //Preprocessor directive
void main()     //Main function is declared
{
     int i, sum=0;
for(i=1;i<=VALUE;i++)
//The for loop goes up to defined value 5 for VALUE
{
          sum=sum+i;

//It sums 1+2+3+4+5

}
printf("Sum = %d\n", sum);
getch();
}
```

The result of the program is as follows:

```
Sum = 15
```

3.27 DEBUGGING AND TESTING OF PROGRAMS

Debugging

Debugging refers a process identifying the root cause of programming errors through which users can edit or correct the program errors. This process finds the errors to debug from the starting point of the programming part. User can select any one source to debug the program, such as design, data program etc. The following factors are available in debugging the programs:

- It echoes the programming code on the screen.
- It identifies missing variables, operators or values.
- It identifies those variables which are un-initialized in the program.
- It provides diagnostic capsules, for input and output data sets.

NOTES

In debugging process, the three types of errors mainly occur. They are as follows:

- **Syntax or compilation error** These type of errors occur during compilation. It prevents to get the compile code. For example, delineators (semicolon) used in C, brackets used in C++ or Java, are not detected in the program (Figure 3.7). In Bash scripting language, missing of `'do'` or `'done'` statement creates compiler error.
- **Runtime exception error** The runtime exception error is detected when the compiled code is running. Java language displays those types of error messages that indicate the location of actual errors. The 'segmentation fault' found in C and 'index out of bounds' found in FORTRAN refer to such type of errors. The following table shows the common error messages and common cause found in different computer languages (Table 3.8):

Table 3.8 *Messages and Cause Found in Different Computer Languages*

Error Messages	Common Cause
Null pointer Exception	In Java, instantiated reference is never used and in C, pointer is never assigned.
Out of Memory	Recursion or infinite loop, memory leak takes place in C++.
Index out of Bound	`while` loop is checked in programming.
Difference error	Improper casting used in Java.
Divide by zero	Denominator is not used and compile of C sets the memory to zero.

- **Logical error** The logical errors are not easy task to debug and the outputs of running programs are also not correct. The various tools available in detecting and fixing the logical errors are as follows:
- **Print intermediate tools:** It uses Boolean variables which turn on and off debug printing.
- **Choose case:** The problem solving program must be covered with small cases.
- **Walk through:** It displays the uncommitted code.

Logical errors frequently occur at loops and ifs statement. In loops, these errors check the Boolean expression of `'while'` loop and checks the range for `'for'` loop. The 'ifs' is used to check the Boolean expressions and negations. The Boolean expressions must carry more than one variable.

NOTES

Figure 3.7 *C Debugger*

Testing

A set of required activity with reference to computer programming is known as testing (Figure 3.8). It has the following characteristics:

Figure 3.8 *Life Cycle of Testing of Programs*

- It starts from component levels and works outward to integrate the entire computer based program.
- Various types of listing techniques are approached at certain points on time.
- It is conducted by software developer if the programming part is small but for the large projects an independent test group of experts are required.
- Testing and debugging are considered as two separate activities. Debugging can be accommodated for any testing strategy.

NOTES

- The testing of programs is frequently referred to as verification and validation **(V & V)**. These two prime phases are taken as follows:

Verification: This process follows a set of activities which checks whether the software is correctly implemented to a specific function. The code inspection and testing are the two techniques use din verification.

Validation: This process involves various set of activities which checks whether the developed programming for desired software system is traceable as per customer requirements. Validation uses inspection code for high level that requires software specification.

This process attempts on the following mechanism which are required to enhance the testing criteria:

- **Performance testing:** This mechanism catches the coding effects involved in applications. If architectural defects are caught, it simulates the load testing on application before deploying it. It deals the performances issues before any defect comes.
- **Configuration management:** This mechanism states the artifacts that makes up project for software development and is able to release the latest versions of the developed system. The configuration management is required to test the source control system, for example Rational Clearcase. This is considered as best practice for configuration management.
- **Quality control:** This mechanism inspects the series of programming part, reviews and tests meanwhile the processing of software. It also ensures the work product that fulfils the requirements placed upon it.

The two types of testing techniques are used for computer programming as follows:

Black box (functional testing)

This technique implies to test all types of input types. It checks to test all possible input values that identifies input types. This mechanism is known as **partitioning the inputs**. For example, various operations (addition, subtraction, multiplication and division), numbers (positive or negative numbers, integers, real numbers) etc. are partitioned for the programming. This testing checks the following algorithm:

```
Input
      Interface
Processing
Output
      Interface
```

The tested program is observed on the specified input values. It generates the certain output collected by second interface.

The types of black box testing are as follows:

Stress Testing

This type of testing works with large number of input values, complex numerical values, large number of queries that checks the extra loading of the applications.

Load Testing

This testing works with testing of web sites to find the failure point of web site applications and the performance degrades.

NOTES

The advantages and disadvantages of black box are as follows:

Advantages of black box testing

The advantages of black box are as follows:

- If the testing is done with black box, it is reproducible.
- The overall environment in which program runs is also tested in this method.
- The invested effort is reused multiple times.

Disadvantages of black box testing

The disadvantages of black box are as follows:

- Produced result is overestimated.
- All the properties of software are not tested.
- The reason behind failure is not found.

White box (structural testing)

This technique refers to structural design that implies to access the code or test for all possible paths. These paths are possible path required in structural testing. The program is analysed by double check conditionals. Paths represent various routes through that are included in the program. For example,

```
if (a>b)
{
            p=p+1;
}
else
          {
            p=p+3;
          }
if (c<d)
{
            p=p+3;
}
else
{
            p=p+4;
}
```

NOTES

In the above coding, two paths are specified according to checking conditions, for example, if `(a>b)` then variable `p` is assigned an increased value of 1 to the immediate value otherwise else part would be executed as `p=p+3`. Therefore, it determines the path as per instruction.

Types of white Box Testing

The three types of testing are included in white box testing. They are as follows:

- **Path Testing** This testing refers to typical white box test. Sometimes, it is used in black box test. In the programming, a certain path is chosen that includes all possible input values and then the required correct output is displayed. The produced output is compared to predefined values. The possible faults are noted down for further queries. For example, there is a program which consists of any one of the loop among `while, for, do-while.` The loop is assigned for twenty times execution. The body of the loop contains case or if instructions allowing ten different types of routes. The total number of paths can be determined as 10^{20}. For example,

```
if (Code_C)
{
    instruction block A_1
}
else
{
    instruction block A_2
}

if (Code_C)
{
    instruction block B_1
}
else
{
    instruction block B_2
}
```

In the above code, the given conditions specified in a program are not independent to each other and also all paths are not possible. So, only two possible routes can work in executing the programs. If the instruction blocks either `A_1` or `A_2` is executed because `Code_C` is satisfied but paths `A_1` and `B_2` and `A_2` and `B_1` are never executed if the given condition `Code_C` comes false.

Loop testing

This testing makes path testing complicated. The loop contains bugs that increase the programming complications and not easy to find. The nested loops conceal many bugs. For example, the following set of instruction shows the loop testing:

NOTES

```
label_one:
     instruction block A_1
label_two:
if Code_C1 then goto label_one:
       instruction block A_2
if Code_C2 then goto label_two:
```

In the above set of instruction, the command goto is used.

Domain testing

Domain testing represents white box testing that checks variables, conditions, indexes. It improves the valid range and contains checking to accept the domain structure in which objects are specified within specified domain. This testing is also known as **'garbage in garbage out'** testing. In this testing, variables are initialized correctly and arguments are taken as same types. All Boolean expressions and logical operators, such as AND, OR, NOT are initialized according to domain setup. Domain here represents the proper declaration of classes and objects.

CHECK YOUR PROGRESS

10. What are the various dynamic memory functions?
11. What information does a file pointer point at?
12. List four different types of loops.
13. What are the common sorts of non-local control constructs?
14. What are the steps that can be taken to lessen the memory requirement?
15. What are the two processes to test a program?

3.28 SUMMARY

In this unit, you have learned that:

- A computer program is a set of instructions to a computer to carry out a specified task. A program is written in a language that is understood by the computer. 'C' language has been found to be most suitable for this purpose.
- Data is used in a program to get information. C being a versatile language can handle many different types of data.
- Various aspects of a computer program are integers, variables, constants, and operators.

NOTES

- Every C program has one main function comprising of the declaration part and the executable part.
- String processing represents the various operations involved in string manipulation and string collection. Handling of character strings is the initialization, processing and manipulation of character or string.
- Subprograms are dependant on the main program and the basic idea behind them is to make a group of collection of statements invoked by the name.
- Files used for storing data are popularly known as data files and are stored in the secondary or auxiliary storage devices like floppy disks or hard drive disks.
- Procedural programming is an imperative programming that specifies steps to solve a problem. Such programming paradigm is based on the concept of 'procedure call'.
- The two classic ways to solve the programming problems are known as top down and bottom up design.
- Efficient programs are developed by keeping in mind the two critical factors of memory and execution time.
- Program correctness implements the programming code that is verified online.
- The process of debugging is used to find errors from the starting point of programming.

3.29 KEY TERMS

- **Constants:** Refer to fixed values which do not change during the execution of a program.
- **Variable:** Refers to a symbol that stands for a value which may vary.
- **Operator:** refers to a symbol which indicates an operation to be executed.
- **Recursion:** A powerful technique which is used to call a function itself.
- **Pointer:** A pointer is a variable that points to another variable.
- **Address operator:** An operator which assesses the address of its operand. Also called a substitute variable.
- **Dynamic memory allocation:** Functions of the C library to allocate memory storage space at run-time.
- **Label:** An explicit name or number assigned to a fixed position within the source code, and which may be referenced by control flow statements appearing elsewhere in source code.
- **Loops:** A sequence of statements which is specified once but which may be carried out several times in succession.

- **Debugging:** Refers to the process identifying the root cause of programming errors through which users can edit or correct the program errors.

3.30 ANSWERS TO 'CHECK YOUR PROGRESS'

NOTES

1. C is portable, modular, terse, efficient on most machines and appealing.
2. System level programming, creation of a new language, creating games, developing applications such as library systems, making embedded devices in designing chips and automatic control system in industrial operations are some practical application of C.
3. Character, integer, real numbers, void and enum are the five basic data types.
4. The scope of a variable describes where in the text of a program you can use the variable, whereas the extent describes when during the execution of a program, a variable has a value.
5. The following steps have to be taken to execute a program written in C:
   - Create the program
   - Compile the program
   - Link the program with functions that are needed from the C library
   - Execute the program
6. String processing represents the various operations involved in string manipulation and string collection.
7. A function which invokes itself repeatedly until some condition is satisfied is called a recursive function.
8. Some library functions of C are as follows:

   – printf ( )

   – scanf ( )

   – getcher ( )

   – putcher ( )
9. Function prototype, function definition, function call are the three parts of a function.
10. Certain dynamic memory functions are as follows:

    malloc ( )

    calloc ( )

    realloc ( )

    free ( )

NOTES

11. The information pointed at by a file pointer is:
    - Location of the buffer
    - The position in the file of the character currently being pointed to
    - Whether the file is being read or written
    - Whether an error has occurred or the end of the file has been reached.
12. Count-controlled loop, condition-controlled loop, collection-controlled loop and infinite loops aer the four types of loops.
13. Common non-local control constructs are exceptions, conditions and continuations.
14. The steps that can be taken to lessen the memory requirements are:
    - Program must be simple.
    - Simple algorithms must be used.
    - Arrays, strings and pointers must be declared with suitable memory address and correct sizes
    - Multidimensional arrays must be limited.
    - Memory compression features must be evaluated and incorporated.
15. Verification and validation are the two processes to test program.

3.31 QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write the steps in writing and running a program in C.
2. What are the different data types in C?
3. Define each of the following:
   (a) Variables
   (b) Constants
   (c) Parameters
   (d) Memory allocation
4. What is programming sequencing.
5. What do you mean by right or left associativity of operators?
6. What are file pointers?

Long-Answer Questions

1. What are the benefits of structured programming? Explain with examples.
2. Using flowcharts compare the top-up and bottom-down design.
3. Explain the function of recursion giving examples.
4. Write an explanatory note on ring processing.

5. What are the three types of errors encountered during the process of debugging? Also list the steps involved in the testing procedure.
6. Explain the types and functioning of arrays.
7. Explain the concept of pointers.

NOTES

3.32 FURTHER READING

Friedman, Daniel P. *Essentials of Programming Languages*. Boston, MA: MIT Press.

Manber, Udi. *An Introduction to Algorithms: A Creative Approach*. New York: Addison-Wesley.

Farrell, Joyce. *Computer Programming Logic Using Flowcharts*. New York: Boyd & Fraser.

COMPUTER FUNDAMENTALS AND PROGRAMMING

VENKATESHWARA OPEN UNIVERSITY
www.vou.ac.in