CONTENTS

SYLLABUS

COMPUTER FUNDAMENTAL & PRGRAMMIMG IN C

SECTION A

Number system : decimal, octal, binary & hexadecimal, representation of integer, fixed and floating points, character representation: ASCII, EBCDIC

SECTION B

Functional units of computer,:I/O device, primary and secondary memories.

SECTION C

Programming fundamental: Algorithm development, technique of problem solving, flowcharting, stepwise refinement, alogorithm for searching sorting exchange and insertion merging of order lists.

SECTION D

Representation of integers, characters, reals, data types, constant and variables, arithmetic expression, assignment statement logical expression, sequencing, alteration and iteration, arrays, string processing, sub program , recursion, files and pointers testing and debugging of program.

SECTION A

CHAPTER 1 INFORMATION REPRESENTATION

★ LEARNING OBJECTIVES ★

1.1 INTRODUCTION

A digital computer is a machine that accepts a stream of symbols, stores them, processes them according to precise rules, and produces a stream of symbols as its output. Computers are used in scientific calculations, commercial and business data processing, air traffic control, space guidance, the educational field, and many other areas. The most important property of a digital computer is its generality. It can follow a sequence of instructions, called a *program,* that operates on given data.

Few basic features which are common to all digital processing of information are given below :

- All streams of input symbols to a digital system are encoded with two distinct symbols 0 (zero) and 1 (one), known as *binary digits* or *bits*. Bits can be stored and manipulated reliably and inexpensively with today's electronic circuits.
- The instructions for manipulating the symbols are to be properly specified so that a machine can be built to execute these. The instructions for manipulation are also encoded using *bits*.
- A digital computer has a storage unit which stores the symbols to be manipulated and the encoded instructions for manipulation of the symbols.
- Bit manipulation instructions are realized by electronic circuits.

NOTES

In this chapter we will discuss the representation of information in digital systems. The two main categories of data are: (*i*) Numbers, and (*ii*) Characters. As mentioned earlier that internally in a digital system all data are represented by strings of symbols where each symbol, called a bit, is either a 0 or a 1. The general properties of number systems, methods of their inter-conversions are also discussed in this chapter.

1.2 NUMBER SYSTEMS

Number systems are very important to understand because the design and organisation of a computer depends on the number systems. The modern civilization is familiar with **Decimal Number System,** in which ten digits, namely 0 to 9 are used to represent any number. Now the *decimal number system* is used frequently in the field of Science and Technology. Once the famous Mathematician Laplace stated, "It is India that gave us the ingenious method of expressing all numbers by means of ten symbols, each symbol receiving a value of position as well as an absolute value, a profound and important idea which appears so simple to us now that we ignore its true merit."

Thus, the importance of a number system does not lie in the number of symbols used in it but what is important in it is the concept of face value (absolute value) and the place value (value of position) of a symbol. However, if instead of decimal system the binary number system, using two digits namely, 0 and 1, had been popular everywhere, the understanding of computer working would have been much easier. The binary number system is the one which is used in computers and is based on the fundamental concept of decimal number system. Various other number systems such as Octal System and Hexadecimal System (popularly known as Hex system) also use the fundamental concept of face value and place value.

The knowledge of number systems is essential for understanding of computers. The useful number systems discussed are :

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

1.2.1 Binary Number System

The **Binary Number System,** as the name suggests, consists of two digits namely, 0 and 1. These binary digits are called BITS. Thus, the word **BIT** stands for either of the binary digits, namely 0 or 1. Since this system uses two digits only, it has the **base** or **radix 2.** It may be noted that the base digit namely 2, is not the fundamental or basic digit of the system. Thus, all the numbers in binary system are written with the help of these two digits namely, 0 and 1. The positional value or place value of each digit in a binary

number is twice the positional value of the digit on its right. This number system is identical to decimal number system with the base replaced by 2. The binary numbers are usually written with the base indicated as a subscript on the Least Significant Digit (LSD). For example,

$$(101101.1011)_2$$

NOTES

It can be represented as shown in Figure 1 :

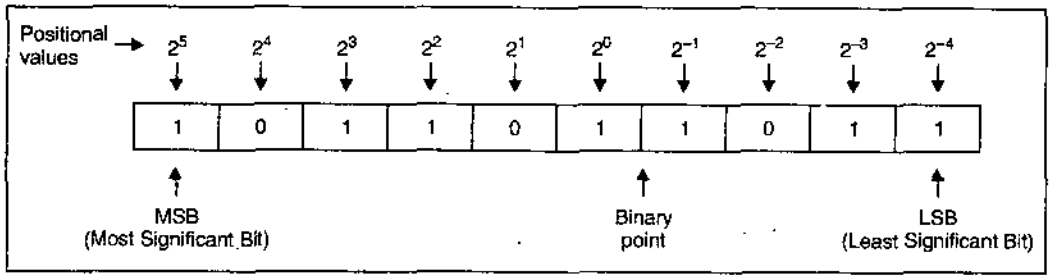


Fig. 1. *Binary number shown with positional values.*

Here, the places to the left of the binary point are positive powers of 2 and places to the right are negative powers of 2.

The commonly used terms in coding of data in computer terminology are :

BIT (BInary digiT). A binary digit is logical 0 or 1.

Nibble. A group of four bits (binary digits) is called a Nibble. It is useful in coding the numeric data to hexadecimal form.

Byte. A group of 8 bits make a byte. A byte is the smallest unit which can represent a data item or a character.

Computer Word. A computer word, like a byte, is a group of fixed number of bits which varies from computer to computer but is fixed for each computer. The number of bits in a computer word is known as the **word size** or **word length.**

Why is Binary Number System used by Computers ?

(*i*) The circuits and switches in a computer have only two states, either on or off, which are represented by 1 and 0.

(*ii*) Handling of two digits, that is, 1 and 0 is simpler, cheaper and more reliable.

(*iii*) Every operation or activity that can be performed by decimal number system can also be done using binary number system, so it does not create any problem.

1.2.2 Octal Number System

This number system has **base** or **radix** 8. The basic digits of this system are 0, 1, 2, 3, 4, 5, 6 and 7. It may be noted that the base 8 is not the basic digit of the system. It is commonly used as a shorthand way of expressing binary quantities. Also the numbers represented in octal number system can be used directly for input and output operations.

NOTES

The octal number system is also a positional value system, wherein each octal digit has its own value or weight expressed as a power of 8. For example,

$$(157246.3174)_8$$

It can be represented as shown in Figure 2 :

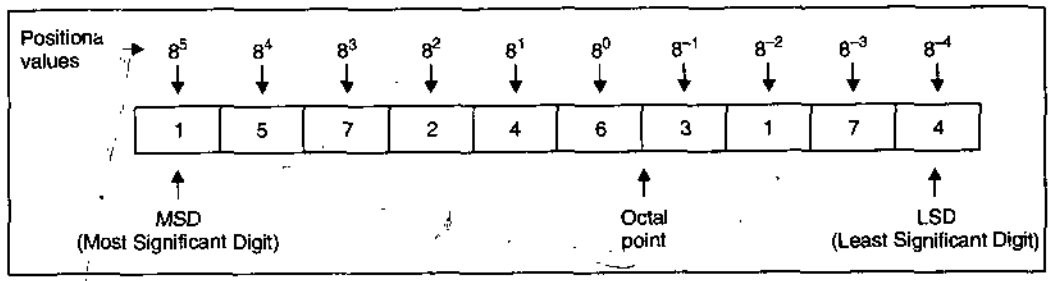


Fig. 2. *Octal number shown with positional values.*

Here, the places to the left of the octal point are positive powers of 8 and places to the right are negative powers of 8.

1.2.3 Decimal Number System

The decimal number system consists of 10 digits namely 0 to 9. A number written using these digits is called a decimal number. For example, the numbers 12876, –1024, 58.74, +768 are decimal numbers. Apart from these digits, the decimal point and ± signs may also be used in writing decimal numbers. The **base** or **radix** of the number system is the number of digits used in it. Since the decimal number system consists of 10 digits, the base of this system is 10. In a number system the base is not the fundamental digit of the system because fundamental digits in this system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The value of each digit in a number depends upon the following :

(*i*) The face value of the digit, that is, the digit itself.

(*ii*) The base of the system.

(*iii*) The position of the digit in the number.

Thus, the magnitude of a number depends upon the digits of which it is made, position of the digits and base of the system. For example,

$$351479.8265$$

Since no base is mentioned, the base is taken as 10.

It can be represented as shown in Figure 3 :

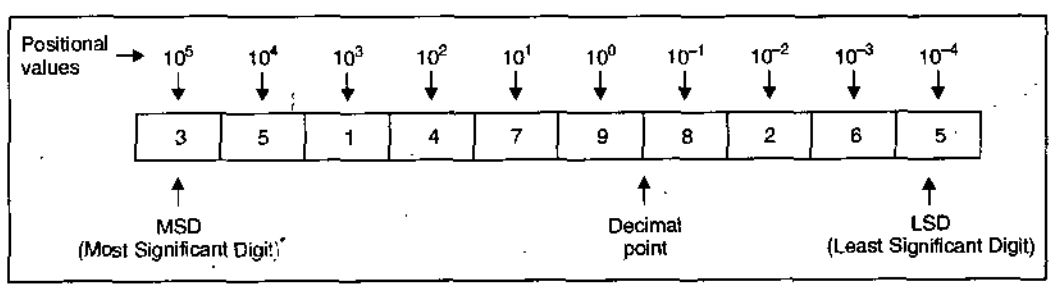


Fig. 3. *Decimal number shown with positional values.*

Here, the places to the left of the decimal point are positive powers of 10 and places to the right are negative powers of 10.

In a decimal number as we move from right to left (starting with the digit before decimal point) the positional or place value of each digit is 10 times the positional value of the digit to its right and as we move from left to right (starting with the digit after decimal point) the positional value of each digit becomes one-tenth of the positional value of the previous digit. The part of the number before the decimal point is called **integral part** and the one after the decimal point is called the **fractional part.**

1.2.4 Hexadecimal Number System

The Hexadecimal Number System, popularly known as Hex System, has sixteen symbols and therefore has the **base** or **radix** as 16 or H. It is very well suited for big computers. The hexadecimal number system represents an information in the concise form. The sixteen symbols used in this system are :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

The equivalence between hex-numbers (hexadecimal numbers) and decimal numbers is given below :

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Hexadecimal number system is also a positional value system, wherein each hexadecimal digit/letter has its own value or weight expressed as a power of 16. For example,

$$(6A9E83.C5BD)_{16}$$

It can be represented as shown in Figure 4 :

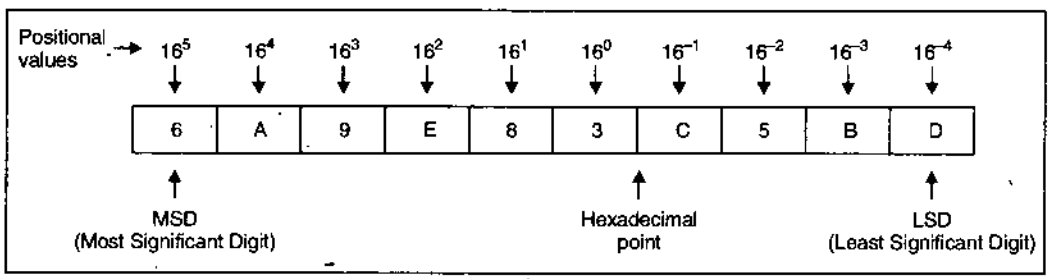


Fig. 4. *Hexadecimal number shown with positional values.*

Here, the places to the left of hexadecimal point are positive powers of 16 and places to the right are negative powers of 16.

Table 1 illustrates the relation between binary, octal, decimal and hexadecimal number systems :

Table 1. Numbers with different bases

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

1.3 CONVERSION BETWEEN TWO DIFFERENT NUMBER SYSTEMS

The decimal number system is known as **International System** of numbers. This base (base 10) was used initially perhaps for the reason that man has 10 fingers. However, this system is unsuitable for computers because a computer uses electrical and electronical components which can exist only in two states. Hence, for computers the binary number system is required. But, the binary number system at the moment is indispensable for computers as it suffers from the defect of expansion. For example, a number in decimal system requiring only one digit for its representation may require more than one bits in binary form.

To overcome this problem various other number systems such as Octal (base 8) and Hexadecimal (base 16 or H) were developed. A base greater than 10 is preferred because it will require even lesser number of digits. The choices of bases 8 and 16 are useful because of their being multiple of two.

1.3.1 Conversion from Binary to Decimal

A binary number can be converted to its decimal equivalent by adding the weights of the various positions in it which have a 1.

Example. *Find the decimal equivalent of the following binary numbers :*

(i) 10110 (ii) 11011100.

NOTES

Solution :

(*i*) $(10110)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$= 16 + 0 + 4 + 2 + 0$

$= (22)_{10}$.

It must be noted that the binary number 10110 has five digits in all. The most significant digit (MSD) has the fifth position (starting from rightmost digit) so it is multiplied by 2^4 and each digit on its right will be half of it in its positional value, so these are multiplied by 2^3, 2^2, 2^1, 2^0 respectively and the products so obtained are added to get the required decimal equivalent.

(*ii*) $(11011100)_2 = 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

$= 128 + 64 + 0 + 16 + 8 + 4 + 0 + 0$

$= (220)_{10}$

$= 220.$

1.3.2 Conversion from Decimal to Binary

A positive decimal integer can be converted to binary form by successive division by 2. The procedure is given below :

Divide the given number N by 2 and let the quotient be q_1 and the remainder be R_1. Again divide the quotient q_1 by 2 and let the remainder be R_2. Continue the procedure of division till the quotient becomes 0 and in this case let the remainder be R_n.

Then, the binary representation by N is given as

$$N = R_n\ R_{n-1} \ldots\ldots R_3\ R_2\ R_1,$$

where each of the R's is either 0 or 1.

Example. *Convert the following decimal integers into their binary equivalents :*

(i) 25 (ii) 283.

Solution : (*i*) Start dividing 25 by 2 and continue the procedure till the quotient is 0. The procedure is shown below :

2 |	25	
2 |	12 – 1	$(= R_1)$ ↑
2 |	6 – 0	$(= R_2)$ |
2 |	3 – 0	$(= R_3)$ |
2 |	1 – 1	$(= R_4)$ |
|	0 – 1	$(= R_5)$ |

The required number in the binary number system

$= R_5\ R_4\ R_3\ R_2\ R_1$

$= 11001$

Thus, $(25)_{10} = (11001)_2$

NOTES

(*ii*) Start dividing 283 by 2 and continue the procedure till the quotient becomes 0.

2	283	
2	141 – 1	$(= R_1)$
2	70 – 1	$(= R_2)$
2	35 – 0	$(= R_3)$
2	17 – 1	$(= R_4)$
2	8 – 1	$(= R_5)$
2	4 – 0	$(= R_6)$
2	2 – 0	$(= R_7)$
2	1 – 0	$(= R_8)$
	0 – 1	$(= R_9)$

The required binary equivalent of 283

$$= R_9\ R_8\ R_7\ R_6\ R_5\ R_4\ R_3\ R_2\ R_1$$
$$= (100011011)_2$$

Thus, $(283)_{10} = (100011011)_2$

The decimal digits and their binary equivalents are given in Table 2.

Table 2. Decimal digits and their binary equivalent

Decimal Digit	*Binary Equivalent*	
0	0	(0×2^0)
1	1	(1×2^0)
2	10	$(1 \times 2^1 + 0 \times 2^0)$
3	11	$(1 \times 2^1 + 1 \times 2^0)$
4	100	$(1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0)$
5	101	$(1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)$
6	110	$(1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0)$
7	111	$(1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)$
8	1000	$(1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0)$
9	1001	$(1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)$

1.3.3 Conversion of Decimal Fractions to Binary Fractions

While converting decimal fractions to binary fractions, instead of dividing by 2, multiply successively by 2 and instead of noting the remainders as in case of integers, keep track of the overflow. If overflow digits are $f_1, f_2, f_3, ..., f_n$ then the equivalent binary fraction is

$$0 \cdot f_1\ f_2\ f_3 \ldots\ldots f_n$$

while multiplying by 2, it must be noted that every time only fractional part is to be multiplied and not the integral part. The following examples illustrate the procedure.

Example 1. *Convert the decimal fractional number 0.8125 into its binary equivalent.*

Solution :

	0.8125	
	× 2	
f_1	← 1 .6250	
	× 2	
f_2	← 1 .2500	(Multiply only the fraction part)
	× 2	
f_3	← 0 .5000	
	× 2	
f_4	← 1 .0000	

Further multiplication gives only zero digits as overflow.

Thus, $(0.8125)_{10} = 0.f_1 f_2 f_3 f_4$

$= (0.1101)_2$

Example 2. *Convert 0.33 into its binary equivalent.*

Solution.

	0.33	
	× 2	
f_1	← 0 .66	
	× 2	(Multiply only the fraction part)
f_2	← 1 .32	
	× 2	
f_3	← 0 .64	
	× 2	
f_4	← 1 .28	
	× 2	
f_5	← 0 .56	
	× 2	
f_6	← 1 .12	and so on.

Thus, $(0.33)_{10} = 0.f_1 f_2 f_3 f_4 f_5 f_6 \ldots\ldots$

$= (0.010101\ldots)_2$

1.3.4 Conversion of Mixed Numbers (from Decimal to Binary)

A mixed number consists of an integral part as well as a fractional part. For example, 38.625 is a mixed number. A mixed number in decimal system can be converted to its binary equivalent by converting the integral and fractional parts separately into their binary equivalent.

NOTES

The following example illustrates this concept :

Example. *Convert the decimal number 38.625 into its binary equivalent.*

Solution : The given decimal number has two parts, namely an integral part 38 and a fractional part 0.625. These are to be converted into their binary equivalents separately, as given below :

2 | 38		
2 | 19 – 0	$(= R_1)$	↑
2 | 9 – 1	$(= R_2)$	↑
2 | 4 – 1	$(= R_3)$	↑
2 | 2 – 0	$(= R_4)$	↑
2 | 1 – 0	$(= R_5)$	↑
0 – 1	$(= R_6)$	↑

Thus, the binary equivalent of $(38)_{10} = R_6\ R_5\ R_4\ R_3\ R_2\ R_1$

$= (100110)_2$

Also

	0.625	
	× 2	
f_1 ↓	← [1] .250	
	× 2	(Multiply only the fraction part)
f_2 ↓	← [1] .500	
	× 2	
f_3 ↓	← [0] .000	

Further multiplication gives only zero digits as overflow.

Thus, $(0.625)_{10} = 0.f_1\ f_2\ f_3$

$= (0.101)_2$

Hence, $(38.625)_{10} = (100110.101)_2$

1.3.5 Conversion of Decimal Numbers to Octal Numbers

For converting integer decimal numbers into their equivalent octal numbers, divide the given number repeatedly by 8 till the quotient obtained is zero. The following example illustrates this concept :

Example. *Convert the following Decimal numbers into their Octal equivalents : (i) 759 (ii) 1598.*

Solution : (*i*) Start dividing 759 by 8 and continue the procedure till the quotient is 0. The procedure is shown below :

8 | 759		
8 | 94 – 7	↑	LSD
8 | 11 – 6	↑	
8 | 1 – 3	↑	
0 – 1	↑	MSD

Thus, $(759)_{10} = (1367)_8$

(*ii*)

8 | 1598		
8 | 199 – 6	↑	LSD
8 | 24 – 7		
8 | 3 – 0		
0 – 3		MSD

Thus, $(1598)_{10} = (3076)_8$

For converting decimal fractions into their equivalent octal fractions, multiply the fractional part repeatedly by 8 and keep track of the overflow. The process of multiplication continues till the fractional part becomes zero or upto required number of digits.

1.3.6 Conversion of Octal Numbers to Decimal Numbers

A mixed octal number can be converted to decimal number by the formula given below :

$$M = d_{n+1}8^n + d_n8^{n-1} + d_{n-1}8^{n-2} + \ldots\ldots d_3 8^2 + d_2 8^1 + d_1 8^0 + d_{-1}8^{-1} + d_{-2}8^{-2} + d_{-3}8^{-3} + \ldots + d_{-n}8^{-n}$$

Here, M is the mixed number

d_{n+1} is the digit in the $(n + 1)$th position of the integral part

d_{-1} is the digit immediately after the octal point.

The following examples illustrate this concept :

Example 1. *Convert the following octal numbers into their decimal equivalents :*

(i) $(47)_8$ *(ii)* $(564)_8$

Solution. (*i*) $(47)_8 = 4 \times 8^1 + 7 \times 8^0$

$= 32 + 7 = 39$

Thus, $(47)_8 = (39)_{10}$

(*ii*) $(564)_8 = 5 \times 8^2 + 6 \times 8^1 + 4 \times 8^0$

$= 5 \times 64 + 6 \times 8 + 4 \times 1$

$= 320 + 48 + 4$

$= 372$

Thus, $(564)_8 = (372)_{10}$

Example 2. *Convert the following octal fractions to their decimal equivalents :*

(i) $(0.34)_8$ *(ii)* $(0.542)_8$

Solution. (*i*) $(0.34)_8 = 3 \times 8^{-1} + 4 \times 8^{-2}$

$= 3 \times 1/8 + 4 \times 1/64$

$= 3 \times 0.125 + 4 \times 0.015625$

$= 0.375 + 0.062500$

$= (0.4375)_{10}$

(*ii*) $(0.542)_8 = 5 \times 8^{-1} + 4 \times 8^{-2} + 2 \times 8^{-3}$

$= 5 \times 0.125 + 4 \times .015625 + 2 \times 0.001953125$

$= 0.625 + 0.062500 + 0.003906250$

$= (0.691406250)_{10}$

NOTES

1.3.7 Conversion from Octal to Binary

The octal number system is widely used as a shorthand way of expressing binary values. The octal number system groups three binary bits together into one digit (0 to 7) as given in Table 3.

Table 3. Octal numbers and their binary equivalent

Octal	*Binary Equivalent*
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

To convert an octal number to its binary equivalent, each digit of the octal number is converted to its 3 bits binary equivalent.

The following examples illustrate this concept :

Example 1. *Convert the following octal numbers into their binary equivalents :*

(i) $(746)_8$ *(ii)* $(5043)_8$

Solution : *(i)* $(746)_8 = \frac{111}{7}\frac{100}{4}\frac{110}{6}$ (Replace each octal digit by its 3 bits binary equivalent)

$= (111100110)_2$

Thus, $(746)_8 = (111100110)_2$

(ii) $(5043)_8 = \frac{101}{5}\frac{000}{0}\frac{100}{4}\frac{011}{3}$

$= (101000100011)_2$

Thus, $(5043)_8 = (101000100011)_2$

Example 2. *Convert the following octal numbers into their binary equivalents :*

(i) $(35.216)_8$ *(ii)* $(417.25)_8$

Solution : *(i)* $(35.216)_8 = \frac{011}{3}\frac{101}{5}.\frac{010}{2}\frac{001}{1}\frac{110}{6}$

$= (11101.01000111)_2$

(Discarding the leftmost and right most zero)

Thus, $(35.216)_8 = (11101.01000111)_2$

(ii) $(417.25)_8 = \frac{100}{4}\frac{001}{1}\frac{111}{7}.\frac{010}{2}\frac{101}{5}$

$= (100001111.010101)_2$

Thus, $(417.25)_8 = (100001111.010101)_2$

1.3.8 Conversion from Binary to Octal

For converting a binary number into its octal equivalent the following steps are followed :

(*i*) Divide the given binary number before the binary point into groups of three bits each (from right to left) and after the binary point into groups of three bits each (from left to right) by adding 0 bits for completing the groups (if needed).

(*ii*) Replace each group by its octal equivalent.

The following examples illustrate this concept :

Example. *Convert the following binary numbers into their octal equivalents :*

(i) $(11101)_2$ *(ii)* $(101010011011.10100011)_2$

Solution : (*i*) $(11101)_2 = \overleftarrow{\frac{011}{3}\frac{101}{5}}$ (Replacing each group by its octal equivalent, 0 added on leftmost position for completing the group)

Thus, $(11101)_2 = (35)_8$

(*ii*) $(101010011011.10100011)_2 = \overleftarrow{\frac{101}{5}\frac{010}{2}\frac{011}{3}\frac{011}{3}} . \overrightarrow{\frac{101}{5}\frac{000}{0}\frac{110}{6}}$

(Replacing each group by its octal equivalent, 0 added on rightmost position for completing the group).

Thus, $(101010011011.10100011)_2 = (5233.506)_8$

1.3.9 Conversion of Decimal Numbers to Hexadecimal Numbers

For converting integer decimal numbers into their equivalent hexadecimal numbers, divide the given number repeatedly by 16 (if the remainder is greater than or equal to 10 then write its symbol, that is, A to F, otherwise the digit 0 to 9) till the quotient obtained is zero.

The following example illustrates this concept :

Example. *Convert the following Decimal numbers into their Hexadecimal equivalents :*

(i) $(28)_{10}$ *(ii)* $(1795)_{10}$

Solution. (*i*) Start dividing 28 by 16 and continue the procedure till the quotient is 0. The procedure is given below :

16	28
16	1 – C
	0 – 1

Thus, $(28)_{10} = (1C)_{16}$

NOTES

(*ii*)

16	1795
16	112 – 3
16	7 – 0
	0 – 7

Thus, $(1795)_{10} = (703)_{16}$

For converting decimal fractions into their equivalent hexadecimal fractions, multiply the fractional part repeatedly by 16 and keep track of the overflow. If the overflow is greater than equal to 10 then write its symbol, that is, A to F, otherwise the digit 0 to 9. The process of multiplication continues till the fractional part becomes zero or upto required number of digits.

1.3.10 Conversion of Hexadecimal Numbers to their Decimal Equivalents

The conversion of hexadecimal numbers to their decimal equivalents is performed by using the concept of the positional value of each digit in the number, whether it is an integer, a fraction or a mixed number. The following examples illustrate this concept :

Example. *Convert the following hexadecimal numbers to their decimal equivalents :*

(i) $(9D)_{16}$ *(ii)* $(517)_{16}$

Solution : (*i*) $(9D)_{16} = 9 \times 16^1 + D \times 16^0$

$= 144 + 13 \times 1$ ($\because$ D = 13)

$= 144 + 13 = 157$

Hence, $(9D)_{16} = (157)_{10}$

(*ii*) $(517)_{16} = 5 \times 16^2 + 1 \times 16^1 + 7 \times 16^0$

$= 5 \times 256 + 1 \times 16 + 7 \times 1$

$= 1280 + 16 + 7 = 1303$

Hence, $(517)_{16} = (1303)_{10}$

1.3.11 Conversion from Hexadecimal to Binary

For converting a hexadecimal number to its binary equivalent, each digit of the hexadecimal number is converted to its 4-bits binary equivalent. For reading convenience, usually each nibble (4-bits binary equivalent) is written with a little space in between. The following examples illustrate this concept :

Example 1. *Convert the following hexadecimal number into their binary equivalent :*

(i) $(B9F)_{16}$

Solution. (*i*) $(59F)_{16} = \underset{5}{0101}\ \underset{9}{1001}\ \underset{F}{1111}$ (Replace each hexadecimal digit by its 4 bits binary equivalent)

$= (0101\ 1001\ 1111)_2$

Thus, $(59F)_{16} = (0101\ 1001\ 1111)_2$

Example 2. *Convert the following hexadecimal number into their binary equivalent :*

(i) $(F3A.CB)_{16}$

Solution : *(i)* $(F3A.CB)_{16} = \underset{F}{1111}\ \underset{3}{0011}\ \underset{A}{1010}\ .\ \underset{C}{1100}\ \underset{B}{1011}$

(Replace each hexadecimal digit by its 4 bits binary equivalent)

$= (1111\ 0011\ 1010\ .\ 1100\ 1011)_2$

Thus, $(F3A.CB)_{16} = (1111\ 0011\ 1010\ .\ 1100\ 1011)_2$

NOTES

1.4 REPRESENTATION OF INFORMATION

We are familiar about different types of number systems. In order to talk to computers one has to convert the information, numeric or non-numeric, into binary form. Therefore, one must know how the information is stored in computer memory.

1.4.1 Binary Representation of Integers

The different ways of integer representation in computer memory are :

(i) Sign and Magnitude representation

(ii) One's complement representation

(iii) Two's complement representation

(i) Sign and Magnitude Representation

It is the conventional form for number representation. Every integer has a sign (+ or –) and a string of digits representing the magnitude.

For example,

+ 241 or 745 are positive integers (+ sign may be omitted)

– 127, – 82 are negative integers

The sign of a number is represented by the MSB (Most Significant Bit). If it stores value 0, the sign is +, and if stores 1, the sign is –.

Let us assume that the word size of the computer is 16 bits, then +93 will be represented as given below :

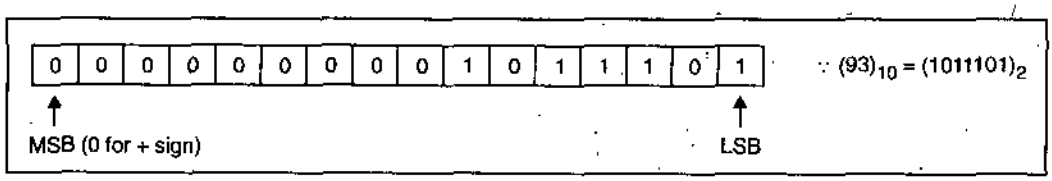


Fig. 5. *Representation of Integer Number in binary form.*

NOTES

Since, bit pattern of 93 consists of 7 bits, it is expanded to 15 bits by adding the required number of zeros to the left as 000000001011101.

–33 will be represented as given below :

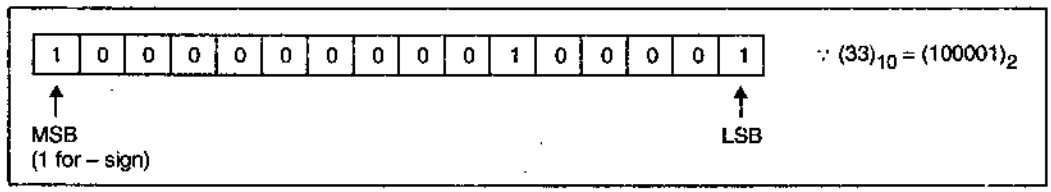


Fig. 6. *Number in binary form.*

Since bit pattern of 33 consists of 6 binary digits, it is expanded to 15 bits by adding the required number of zeros to the left as 000000000100001.

Integer Range for N Bit Word. The range for integer numbers using sign and magnitude representation is given by the formula :

$$-2^{N-1} \text{ to } 2^{N-1}-1$$

as 1 bit is used for sign notation so (N–1) bits represent the magnitude.

For N = 16, the range is

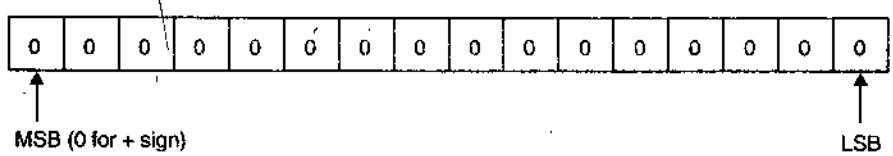
$$-2^{16-1} \text{ to } 2^{16-1}-1$$

that is -2^{15} to $2^{15}-1$

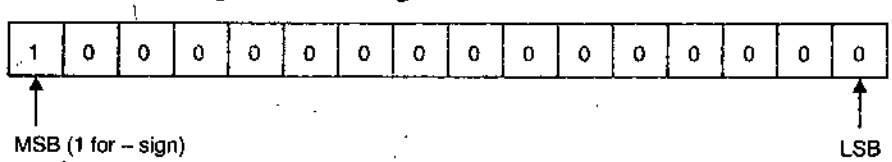
or –32768 to 32767

Note: *In this method of representation, we obtain two representations for 0, one for + 0 and other for – 0.*

The number + 0 is represented as given below :



The number – 0 is represented as given below :



(ii) One's Complement Representation

Using one's complement positive numbers are represented by their binary equivalents (also known as true forms) and negative numbers by their 1's complements (also known as 1's complement forms). For an *n*-bit number, the maximum positive number which can be represented in 1's complement form is $(2^{n-1}-1)$ and the minimum negative number $(2^{n-1}-1)$ is .

1's complement of a binary number is found by replacing every 0 with 1 and every 1 with 0.

NOTES

For example,

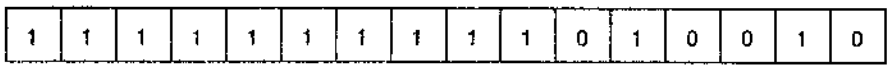
1's complement of binary number 100111 will be 011000

The number – 45 using a 16-bit word is represented as given below :

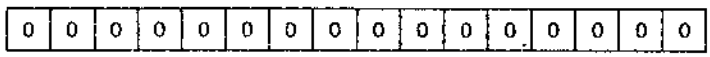
$\because \quad (45)_{10} = (101101)_2$

Now the 16-bit pattern is 0000000000101101

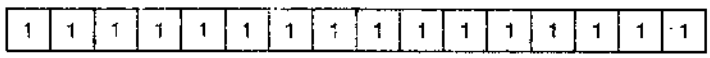
1's complement of the above pattern is 1111111111010010, which is stored as :



Note. In this method of representation, like sign bit magnitude representation, we obtain two representations of 0, one for +0 and other –0. These representations are:



Representation of +0



1's complement representation of —0

(iii) Two's Complement Representation

Using two's complement positive numbers are represented by their binary equivalents (also known as true forms) and negative numbers by their 2's complement form. For an n-bit number, the maximum positive number which can be represented in 2's complement form is $(2^{n-1}-1)$ and the minimum negative number is -2^{n-1}.

2's complement of a number is found by adding 1 to its 1's complement.

or

For finding the 2's complement of a number, start from the LSB and write the bits as it is till the first 1 appears (**do not** change the first 1 to 0), then write the 1's complement of the remaining bits to the left side of it.

For example, 2's complement of binary number 1110001 will be calculated as given below :

1's complement of 1110001 = 0001110

$$\underline{\qquad\qquad + 1}$$

2's complement of 1110001 = 0001111

Note. Rules for binary addition are :

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

NOTES

The number – 45 using a 16-bit word is represented as given below :

$$(45)_{10} = (101101)_2$$

Now the 16-bit pattern is 0000000000101101

1's complement of the bit pattern is 1111111111010010

The 2's complement is obtained as under :

1's complement is 1111111111010010

$$\underline{\qquad\qquad +1}$$

2's complement is 1111111111010011

The above shown pattern is stored as :

1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1

Note. *Unlike sign-bit magnitude and 1's complement representation, both + 0 and – 0 have the same representation under 2's complement method.*

In general, the negative numbers are stored in the computer memory in 2's complement form and positive numbers in sign and magnitude form.

Comparison between 1's and 2's Complements

A comparison between 1's and 2's complements reveals the advantages and disadvantages of each.

(*i*) The 1's complement has the advantage of being easier to implement by digital components (*viz.* inverter) since the only thing to be done is to change the 1's to 0's and vice versa. To implement 2's complement we can follow two ways : (1) by finding out the 1's complement of the number and then adding 1 to the LSB of the 1's complement, and (2) by leaving all leading 0's in the LSB positions and the first 1 unchanged, and only then changing all 1's to 0's and vice versa.

(*ii*) During subtraction of two numbers by complement method, the 2's complement is advantageous since only one arithmetic addition is required. The 1's complement requires two arithmetic additions when an end-around carry occurs.

(*iii*) The 1's complement has an additional disadvantage of having two arithmetic zeros: one with all 0's and one with all 1's. The 2's complement has only one arithmetic zero. The fact is illustrated below :

We consider the subtraction of two equal binary numbers 1010 – 1010.

Using 1's complement :

$$\begin{array}{rl} & 1010 \\ \underline{+} & 0101 \text{ (1's complement of 1010)} \\ + & 1111 \text{ (negative zero)} \end{array}$$

We complement again to obtain (–0000) (positive zero).

Using 2's complement :

$$\begin{array}{rl} & 1010 \\ + & 0110 \text{ (2's complement of 1010)} \\ \hline \text{(Carry over)} & \boxed{1}0000 \end{array}$$

After dropping the carry over, the result is +0000.

In this 2's complement method no question of negative or positive zero arises.

NOTES

1.4.2 Fixed-point Representation of Numbers

In a fixed-point system of number representation all numbers are represented as integers or fraction. Signed integer or BCD numbers are known as fixed-point numbers because they contain no information about the binary point or decimal point. The binary or decimal point is assumed at the extreme right or left of the number. If the binary or decimal point is at the extreme right or left of the computer word, all numbers are positive or negative numbers. If the radix point is assumed to be at the extreme left, all numbers are positive or negative fraction. Suppose one has to multiply 7.35 × 80.64. This will be represented as 735 × 8064. The result will be 5927040. The decimal point has to be placed by the programmer to get the correct result, that is 592.7040. Thus in fixed-point of representation the user has to keep the track of radix point which is a tedious (very difficult) job.

In scientific applications of computers fractions are frequently used. So a system of representation which automatically keeps track of the position of the binary or decimal point is needed. Such a system of representation is known as floating point representation of numbers. It is discussed in the next section. Many computers and all electronic calculators use floating-point arithmetic operations. The computers which do not have internal circuitry for floating-point operations can solve the scientific problems involving fractions with the help of floating-point software.

1.4.3 Binary Representation of Real (Floating Point) Numbers

A number having both integer part as well as a fractional part is called real number or floating point number. It may be either positive or negative. For example, 975.88, 0.586, – 0.866 represent real decimal numbers and 1101.001, 0.0101, – 100.1001 represent binary real numbers.

Real numbers are represented in the memory of the computer by their **mantissa** and **exponent.** In the general form,

$$N = M \times R^e$$

The mantissa M and the exponent e are actually stored in a register of a computer. But, the base or radix R and the radix-point (decimal or binary point) are not present in the register. An assumption is made about these and the electronic circuitry taking these things into account performs the computation and manipulation.

NOTES

Let us assume a 16-bit word for a computer having two parts : a 10-bit mantissa and a 6-bit exponent. The mantissa is in two's complement form ; the left most bit represents a sign bit. The binary point is assumed to be to the right of the sign bit.

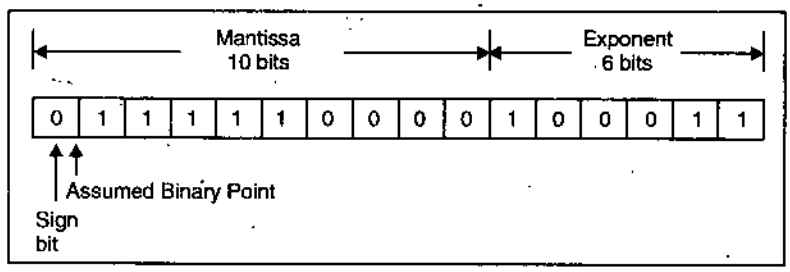


Fig. 7. *Floating point format.*

The 6 bits of the word store the exponent without any sign. The reason for not having the place for storage of the sign of exponent is that the exponent is represented in **biased** form.

Now, let us understand the meaning of biased form of the exponent.

The minimum number represented by 6 bits is (-2^5), that is, –32. While representing align the floating point numbers, this number 32 is added to the actual exponent, thus leaving no space for negative exponent. The new exponent (after adding 32 to the actual exponent) is called **Biased Exponent.** This code for floating point representation is called as the base 2 excess 32 code.

Some examples of this system for exponent part are given in Table 4 given below :

Table 4. Illustration of Biased Exponents

Actual Exponent	*Binary Representation (Biased Exponent)*
–32	0 0 0 0 0 0
–2	0 1 1 1 1 0
0	1 0 0 0 0 0
+5	1 0 0 1 0 1
+31	1 1 1 1 1 1

Now, using the above definitions, the floating point number in figure 7 is :

Mantissa part is + 0.111110000

Exponent part is 100011

Subtracting 100000 from exponent

($\because (100000)_2 = (32)_{10}$ was added to it)

NOTES

The value of the number is

$$N = +(0.11111)_2 \times 2^3 \quad (N = M \times R^e)$$
$$= (111.11)_2$$
$$= 7.75$$

There are many formats of storage of floating point numbers on different computers. Some use two words for the mantissa and one for the exponent; others use one and one-half word for the mantissa and one-half word for the exponent. Few systems even allow to select a format out of many, depending on the accuracy desired. Some use excess-n notation for the exponent; some use 2's complement. Some even use signed magnitude for both the mantissa and the exponent.

Floating point numbers are used to express very large and very small numbers. Using above mentioned system of storage for numbers, we can have 9-bit accuracy (1 bit allowed for sign of mantissa). The exponent bits add nothing to accuracy, only to magnitude. The accuracy of fixed point 2's complement numbers expressed in 16 bits is of 15 bits. So, the floating points are less accurate than an equivalent length fixed point number.

For achieving the full 10 bits of accuracy in the floating point number, the most significant bit of the mantissa is made non-zero and the number so obtained is said to be in the **normalized** (or standard) form. The process of shifting of mantissa left to make the most significant bit non-zero is called **normalization.**

1.4.4 Alphanumeric Codes

Many applications of computer require not only of handling numbers, but also of letters. To represent alphabet it is necessary to have a binary code for the alphabet. In addition the same binary code must represent the decimal numbers and some other special characters. An alphanumeric code is a binary code of a group of elements consisting of ten decimal digits, the 26 letters of the alphabet (both in upper-case and lower-case), and a certain number of special symbols such as #, /, &, %, etc. The total number of elements in an alphanumeric code is greater than 36. Therefore it must be coded with a minimum number of 6 bits ($2^6 = 64$, but $2^5 = 32$ is insufficient). One possible six bit alphanumeric code is given in Table 5. It is used in many computers to represent alphanumeric characters and symbols internally and therefore can be called *"internal code"*. Frequently there is a need to represent more than 64 characters including the lower case letters and special control characters. For this reason the following two codes are normally used.

ASCII

The full form of ASCII (pronounced "as-kee") is "American Standard Code for Information Interchange", used in most microcomputers. It is actually a seven bit code, where a character is represented with seven bits. The character is stored as one byte with one bit remaining unused. But often the extra bit is used

NOTES

to extend the ASCII to represent an additional 128 characters. Some of the codes are shown in Table 5.

Now let us represent the word CODES in ASCII-7 code :

1000011	1001111	1000100	1000101	1010011
C	O	D	E	S

The same word CODES in ASCII-8 code can be represented as given below :

10100011	10101111	10100100	10100101	10110011
C	O	D	E	S

EBCDIC

It is pronounced as "ebb-see-dick". It is a 8 bit code and can represent 256 different characters. All of the 256 bit combinations are not meaningful, so the code can still add new characters if required.

The full form of EBCDIC is "Extended Binary Coded Decimal Interchange Code". It is also an alphanumeric code generally used in IBM equipments and in large computers for communicating alphanumeric data. For the different alphanumeric characters the code grouping in this code is different from the ASCII code. It is actually an eight bit code and a ninth bit is added as the parity bit. Out of the 8 bits, the first 4 bits are known as zero bits and the remainder 4 bits represent digit values.

Now let us represent the word CODES in EBCDIC Code :

11000011	11010110	11000100	11000101	11100010
C	O	D	E	S

Table 5. Partial list of alphanumeric codes

Character	*6-bit Internal Code*	*7-bit ASCII Code*	*8-bit EBCDIC Code*	*12-bit Hollerith Code*
A	010001	1000001	11000001	12,1
B	010010	1000010	11000010	12,2
C	010011	1000011	11000011	12,3
D	010100	1000100	11000100	12,4
E	010101	1000101	11000101	12,5
F	010110	1000110	11000110	12,6
G	010111	1000111	11000111	12,7
H	011000	1001000	11001000	12,8
I	011001	1001001	11001001	12,9
J	100001	1001010	11010001	11,1
K	100010	1001011	11010010	11,2
L	100011	1001100	11010011	11,3

M	100100	1001101	11010100	11,4
N	100101	1001110	11010101	11,5
O	100110	1001111	11010110	11,6
P	100111	1010000	11010111	11,7
Q	101000	1010001	11011000	11,8
R	101001	1010010	11011001	11,9
S	110010	1010011	11100010	0,2
T	110011	1010100	11100011	0,3
U	110100	1010101	11100100	0,4
V	110101	1010110	11100101	0,5
W	110110	1010111	11100110	0,6
X	110111	1011000	11100111	0,7
Y	111000	1011001	11101000	0,8
Z	111001	1011010	11101001	0,9
0	000000	0110000	11110000	0
1	000001	0110001	11110001	1
2	000010	0110010	11110010	2
3	000011	0110011	11110011	3
4	000100	0110100	11110100	4
5	000101	0110101	11110101	5
6	000110	0110110	11110110	6
7	000111	0110111	11110111	7
8	001000	0111000	11111000	8
9	001001	0111001	11111001	9
Blank	110000	0100000	01000000	No punch
.	011011	0101110	01001011	12,3,8
(	111100	0101000	01001101	12,5,8
+	010000	0101011	01001110	12,6,8
*	101100	0101010	01011100	11,4,8
$	101011	0100100	01011011	11,3,8
)	011100	0101001	01011101	11,5,8
/	110001	0101111	01100001	0,1
,	111011	0111100	01101011	0,3,8
=	001011	0111101	01111110	6,8
–	100000	0101101	01100000	11

NOTES

NOTES

1.5 SUMMARY

- Numbers are represented by a string of digit symbols.
- The binary number system has the base or radix 2, the octal 8 and hexadecimal 16.
- Complements are used in digital systems for simplifying the subtraction operation and for logical manipulations.
- Codes are used to represent information, error detection and error correction.
- Integers can be represented in computer memory using sign and magnitude, 1's complement and 2's complement forms.
- In general, the negative numbers are stored in the computer memory in 2's complement form and positive numbers in sign and magnitude form.

1.6 TEST YOURSELF

1. Fill in the blanks :

(*i*) $(94.00625)_{10}$ = (.....)$_2$

(*ii*) $(11011.0101)_2$ = (.....)$_{10}$ = (.....)$_8$

(*iii*) ASCII stands for ..... .

2. What is the advantage of using hexadecimal numbers ?

3. Find the 1' and 2's complement of the following binary numbers :

(*i*) 11001 (*ii*) 11110

4. Write the 1's complement of the binary numbers given below :

(*i*) 1100_2 (*ii*) 1101_2

(*iii*) 10011_2 (*iv*) 10101_2

(*v*) 000_2 (*vi*) 1111_2

(*vii*) 0.101_2 (*viii*) 100.01_2

5. Write the 2's complement of the following binary numbers :

(*i*) 1011_2 (*ii*) 10111_2

(*iii*) 101.101_2 (*iv*) 111.011_2

(*v*) 000_2 (*vi*) 111_2

(*vii*) 101.01_2 (*viii*) 10.001_2

6. Write a short note on the following :

(*i*) ASCII code (*ii*) EBCDIC code

7. Why was BCD code extended to EBCDIC ? Write EBCDIC code for the following words :

(*i*) EARTH (*ii*) LION

(*iii*) FOX (*iv*) PROCESS

How many bytes are needed for each of these representations ?

SECTION B

NOTES

CHAPTER 2 COMPUTER FUNDAMENTALS

★ LEARNING OBJECTIVES ★

2.1 INTRODUCTION

Computer is perhaps the most powerful and versatile tool ever created by man. Computers have made a serious foray into every nook and corner of our everyday lives. Their presence can be felt at almost every working place viz. schools, colleges, homes, offices, industries, hospitals, banks, airways, railways, research organisations and so on. Computers, large and small, are used nowadays by all kinds of people for a variety of purposes.

A digital computer is a digital device which processes digital data. Thus computer (digital) can be defined as a multipurpose, programmable machine built by logic circuits which accepts binary data as input, processes the data according to the binary instructions, read from its memory and provides result in the form of binary or analog as its output.

A digital computer is basically an electronic device that can transmit, store and manipulate information *i.e.,* data. Several different types of data can be processed by a computer. These include numeric data, character data, (name, address, etc.), graphics data (charts, drawings, photographs, etc.), and sound (music, speech pattern, etc.). The two most common data types are numeric

NOTES

data and character data. Scientific and technical applications are concerned primarily with numeric data, whereas business applications usually require processing of both numeric and character data.

Digital computer operate essentially by counting. All quantities are expressed as discrete digits or numbers.

2.2 CHARACTERISTICS OF COMPUTERS

Computers have some remarkable features which have made them so very popular. These features are basically the reasons for which the computers were originally built. These features are :

1. **Automatic.** The computers are automatic machines in the sense that once started on a job, they carry on, until the job is finished, normally without any user's help. But computers cannot start themselves. They have to be instructed, which (the instructions) specify the way of the job completion.
2. **Speed.** The computers can work at enormously high speeds. They are capable of taking logical decisions, performing arithmetic and non-arithmetic operations on alphabets and copying at unbelievable speed. While talking about the speed of a computer, we do not talk in term of seconds or even milliseconds (10^{-3}). For a computer the units of speed are microseconds (10^{-6}), nanoseconds (10^{-9}) and even picoseconds (10^{-12}). A powerful computer can perform 3 to 4 million simple arithmetic operations per second. The reason for this extremely fast speed of a computer can be attributed to the fact that a computer is an electronic device that operates on electrical signals known as electric pulses. These pulses travel at extremely high speeds and hence the fast speed of computers.
3. **Accuracy.** The computers produce highly accurate and reliable results. In majority of cases the accuracy is close to cent per cent. However, errors can't be ruled out but these are mainly due to human rather than technological weaknesses, that is, due to error in logic developed by the programmer or due to inaccurate data *i.e.*, garbage-in-garbage-out (GIGO).
4. **Versatility.** A computer is capable of performing a wide variety of functions :
   (*i*) It can accept data and produce results.
   (*ii*) It can perform the fundamental arithmetic operations of addition, subtraction, multiplication and division.
   (*iii*) It can perform logical operations.
   (*iv*) It can transfer data internally *i.e.*, data can flow from one part to the other in the machine.
5. **Diligence.** A computer is capable of performing the same task over and over again with the same degree of accuracy and reliability as the first

one. This is because unlike human beings, a computer is free from monotony, tiredness, lack of concentration, etc., and hence can work for hours together without creating any errors.

6. **Large and Perfect Memory.** As a human being our ability to acquire and retain knowledge is limited. But this is not the case with computers. A computer can store and recall any amount of information because of its secondary storage capability with perfect accuracy. Even after several years, the information recalled will be as accurate as on the day when it was fed into the computer. A computer loses information only when it is asked to do so.
7. **No I.Q.** A computer is not intelligent on its own. It cannot think on its own. It can only perform tasks that a human being can. The difference is that it performs these tasks with unthinkable speed and accuracy. It cannot take decisions on its own. Only the user can determine what tasks the computer will perform.
8. **No Feelings.** Computers being machines have no feelings. A human being has feelings and can take decisions but the computers take decisions on the instructions provided to them in the form of programs written by the user.

2.3 FUNCTIONAL UNITS OF COMPUTER

Figure 1 illustrates the organisation of *computer system components.* In this figure, the solid lines are used to indicate the flow of instructions and data, and the dotted lines represent the control exercised by the control unit.

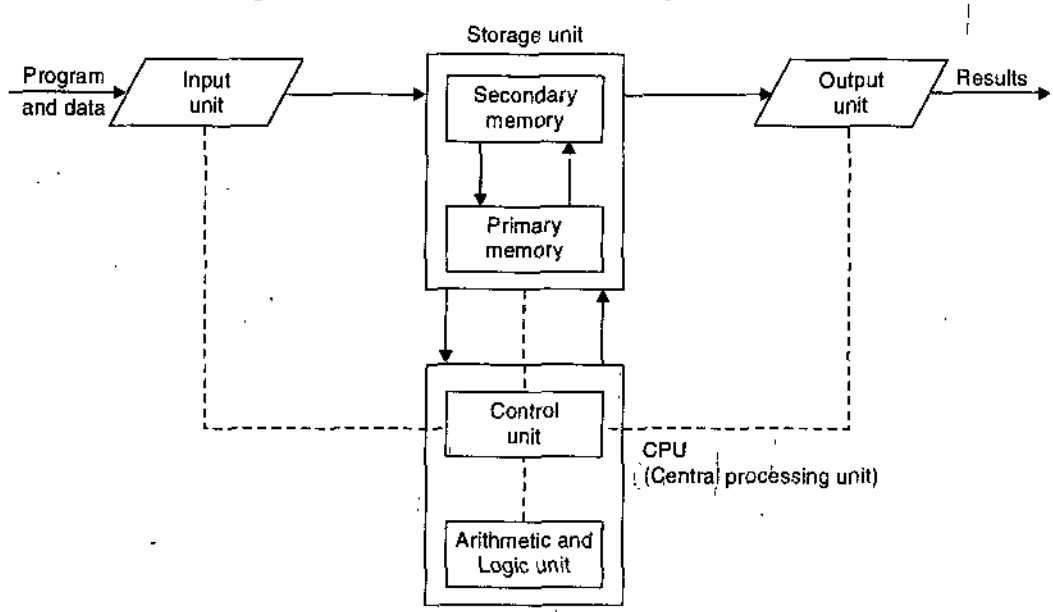


Fig. 1. *Illustration of basic functional units of a digital computer.*

NOTES

The function of each of these units is described below :

Input Unit

Information is entered into a computer through input devices. An INPUT DEVICE reads the data and program into the computer. The program contains instructions about what has to be done with the data. It provides a way of man to machine communication. An input device converts input information into suitable binary form acceptable to the computer. Some popular input devices are listed below :

1. Keyboard
2. Mouse
3. Joystick
4. Floppy and Hard Disk
5. Punched Cards
6. Optical Mark Reader.

In short, the INPUT UNIT performs the following functions :

1. It accepts or reads the data and program (set of instructions).
2. It converts these instructions and data in computer acceptable form.
3. It supplies the converted instructions and data to the computer system for further processing.

Output Unit

The output devices receive results and other information from the computer and provide them to the users. The computer sends information to an output device in the binary form. An output device converts it into a suitable form convenient to users such as printed form, display on a screen, voice output, etc. Some of the popular output units are :

1. Computer screen called VDU (Visual Display Unit)
2. Printer
3. Plotter

In short, the following functions are performed by an output unit :

1. It accepts the results produced by the computer which are in coded form.
2. It converts these coded results to human acceptable form.
3. It supplies the converted results to the outside world.

Storage Unit. The function of storage unit is to store information. The data and instructions that are entered into the computer system through input units have to be stored inside the computer before the actual processing starts. Similarly, the results produced by computer after processing must be kept somewhere before they are passed onto the output unit for display. Moreover, the intermediate results produced by the computer must also be preserved. The storage unit or the primary/main memory of the computer provides support for these storage functions. The main memory is a fast memory. It stores programs along with data. The main memory is directly accessed by the CPU.

The secondary memory, also called the *auxiliary memory*, is used to store the information, data and program instructions permanently. These may be used later on or deleted whenever not required.

To sum up, the storage unit performs following functions :

1. It stores the data and the program (set of instructions).
2. It holds the intermediate results of processing.
3. It stores the final results of processing before they are passed onto the output unit.

NOTES

Central Processing Unit

The CPU is the brain of a computer. Its primary function is to execute programs. Besides executing programs, the CPU also controls the operation of all other components such as memory, input and output devices. The major sections of a CPU are :

(*i*) Arithmetic and Logic Unit (ALU)

(*ii*) Control Unit (CU).

(*i*) **ALU.** The function of an ALU is to perform arithmetic and logic operations such as addition, subtraction, multiplication and division : AND, OR, NOT, EXCLUSIVE OR Operations. It also performs increment, decrement, left shift and clear operations.

(*ii*) **Control unit.** The control unit is the most important part of the C.P.U. as it controls and co-ordinates the activities of all other units such as ALU, memory unit, input and output unit. Although, it does not perform any actual processing on the data, the CU acts as a central nervous system.

To sum up, it performs the following functions :

1. It can get instructions out of the memory unit.
2. It can decode the instructions.
3. It sets up the routing, through the internal wiring, of data to the correct place at the correct time.
4. It can determine the storage location from where it is to get the next instruction after the previous instruction has been executed.

2.4 INPUT DEVICES

Any device that allows information from outside the computer to be communicated to the computer is considered an input device. Since the Central Processing Unit (CPU) of a digital computer can understand only discrete binary information, all computer input devices and circuitry must eventually communicate with the computer in this form. Many devices are capable of performing this task. Some common computer input devices are :

NOTES

1. Punched cards
2. Card-readers
3. Key-punching machines
4. Keyboard
5. Mouse
6. Joystick
7. Trackball
8. Magnetic Tablet (DIGITIZER)
9. Voice-recognition
10. Optical-recognition
11. Scanners

These can be mainly divided into two basic categories :

(*i*) Analog Device

(*ii*) Digital Device

(*i*) **Analog Device.** An analog device is a continuous mechanism that represents information with continuous physical variations. For example, mercury thermometers and record players are all analog devices.

(*ii*) **Digital Device.** A digital device is a discrete mechanism which represents all values with a specific number system. For example, digital watches and computers all process discrete information and use the binary number system.

- **ANALOG INPUT DEVICES.** The Joystick, Trackball, Mouse and Paddle Controls are all transducers that convert a graphics system user's movement into changes in voltage. A transducer is a device that converts energy from one form to another.
- **DIGITAL INPUT DEVICES.** Some of the digital input devices are Keyboard, Lightpen, Digital Cameras and Digitizing Video Images, an Acoustic Tablet, A Magnetic Pen and Tablet (DiGitiZer).
- **GRAPHICAL INPUT TECHNIQUES.** The use of graphical input devices should not be influenced purely by the way the user uses pens and pencils. The user should instead consider the following three factors when he/she writes program for these devices :

  (*a*) What is the user trying to do ?

  (*b*) What input information does the application program need ?

  (*c*) How can the display and computer help the user ?

Each of these questions has many different answers according to the situation, and for each set of answers the user programs the input devices in different ways. The result is that he/she develops certain programming techniques for the use of input devices in each environment.

When we want to design the different characters in a natural and systematic way, the graphical feedback (such as cursor on screen) can play an important role in the input process. It helps the user in using an unfamiliar program. There are many graphical input techniques such as :

(*i*) The use of selection points,

(*ii*) Defining a bounding rectangle,

(*iii*) Multiple keys for selection,

(*iv*) Modes,

(*v*) Multiple selection,

(*vi*) Menu selection.

The most important fact that should be considered while choosing any input device is that the CPU of the computer must not be overloaded.

NOTES

Punched Cards

In the early years of computer evolution, the punched cards were the most widely used input medium for most computer system. These days they are not extensively used in computer industry as number of fast input devices are available, but in order to understand the potential of currently available input devices, it is necessary to understand the concept behind the working of punched cards. There are two types of punched cards—one has eighty columns and the other has ninety six columns.

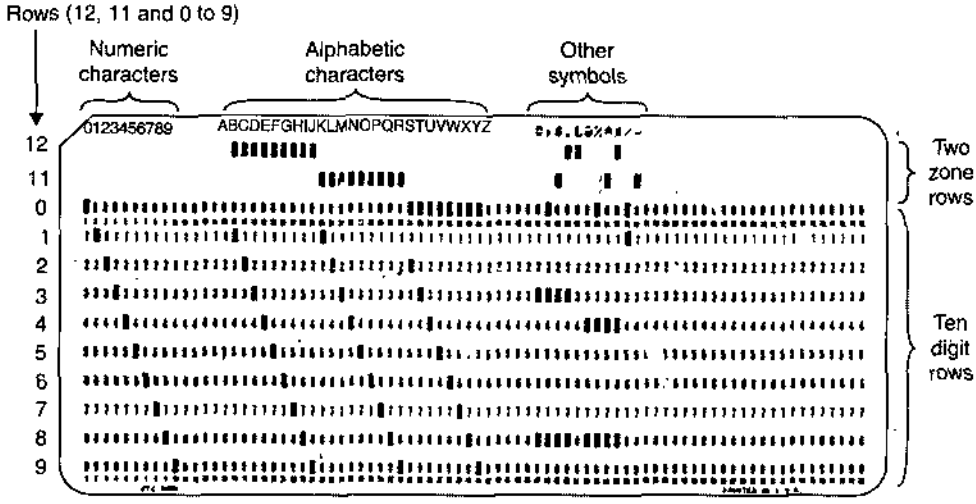


Fig. 2. *A modern Hollerith card can handle a maximum of 80 alphanumeric characters. As this sample shows, numbers are coded by making a single punch in the appropriate card column; letters and special characters are coded by using double or triple punches. Note that the upper left-hand corner is missing from the card. This helps the machine operators make sure all the cards are turned the same way and that the deck itself has the proper orientation.*

NOTES

The 80-column card is divided from left to right into 80 vertical columns numbered from 1 to 80. It is again divided into 12 rows numbered 12,11,0,1,2,3,4,5,6,7,8 and 9 from top to bottom. Each column of this card can be represented as one character, so a maximum of eighty characters can be represented on one card. The digit 0 to 9 are represented by punching one hole in the corresponding row position. The alphabet A to Z are represented by a combination of two holes in two of the row positions. The top three rows—12, 11 and 0 are zone punching positions and the rows 0 to 9 are numeric punching position. A logical combination of zone and numeric punches is required to represent alphabets.

For example, letter A through I are coded by using a 12 zone punch and numeric punches 1 through 9; letter J through R are coded by using a 11 zone punch and numeric punches 1 through 9 and letters S to Z are coded by using a 0 zone punch and numeric punches 2 through 9 respectively. Special characters are coded by punching one, two or three column cards. This coding system is known as **Hollerith code** after the name of the Herman Hollerith who first used punched cards.

The 96-column card, which was developed to store 20 percent more data in a small amount of space, never found widespread use. It is only one-third the-size of an 80-column card. The 96-columns are separated into three 32 column sections or tiers. The upper portion of card, which is not used, for punching holes is used as the print area. These cards have round holes instead of rectangular holes of 80-column cards. Moreover the standard 6 bit code is used instead of Hollerith code for recording the data on 96 column card. Each of the 96-columns has 6 punch positions and remaining four are numeric positions. The presence of a hole in a punch position indicates 1 bit.

Punched cards are rarely used today. However, you may occasionally encounter them in large companies such as public utilities, where they are still used for billing. When the customer returns the card with payment, the keypunch operator uses the keypunch machine to record new data on the card, then he or she runs the card through a card reader to input the data into the computer (usually a mainframe).

Card Readers

A card reader is an input device. It transfers data from the punched card to the computer system. The card reader will read each punch card by passing light on it. Each card will be passed between a light source and a set of light detectors. The presence of hole causes the light to produce a pulse in the director. These pulses are transformed into binary digits by the card reader and sent to the computer for storage. Card is then submitted in the output stack. Card readers can read upto 2000 cards per minute. Figure 3 illustrates a card reader and figure 4 a card verifier.

NOTES

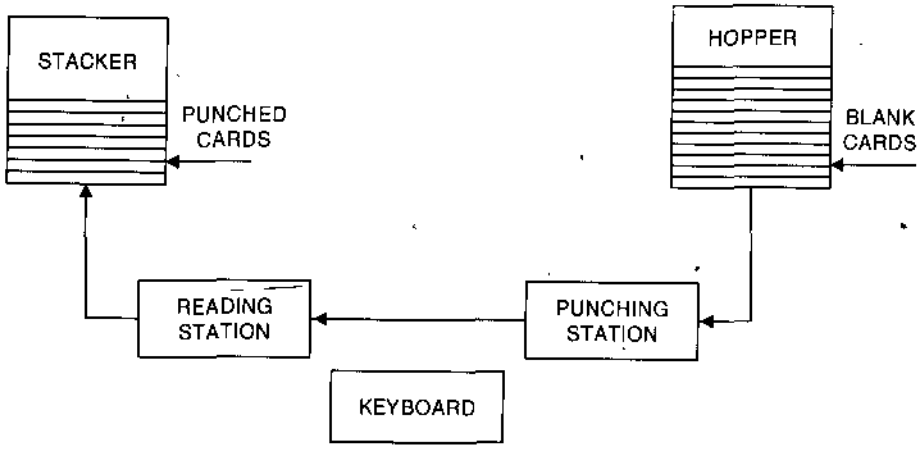


Fig. 3. *Card reader.*

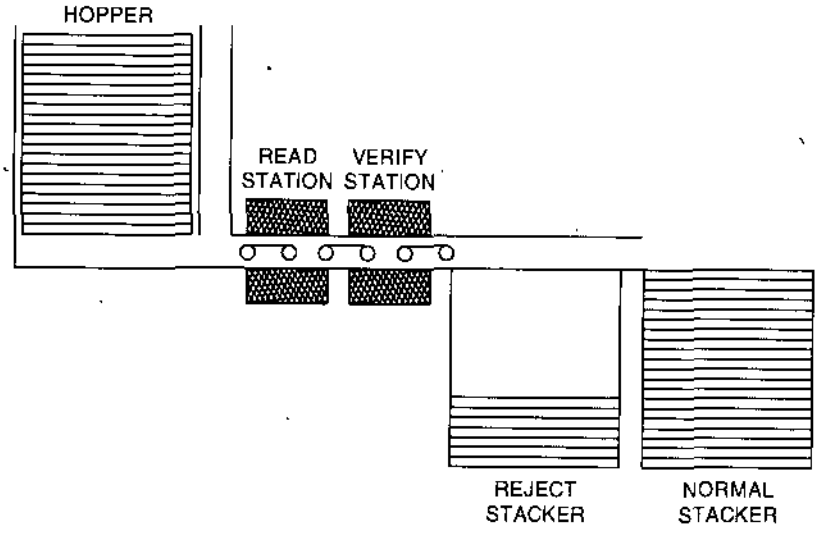


Fig. 4. *Card verifier.*

Figure 5 shows the illustration of keypunching and verification procedure :

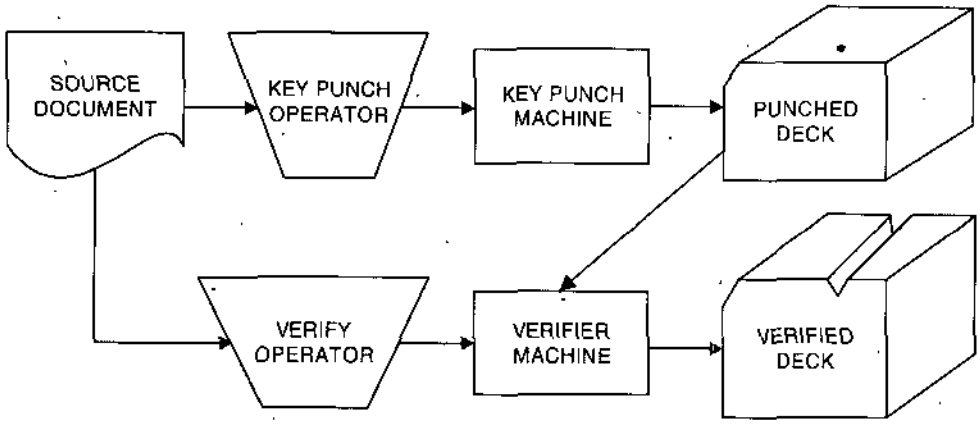


Fig. 5. *Illustrating keypunching and verification procedure.*

NOTES

There are two types of card readers depending upon the mechanism used for sensing the punched holes in the cards :

- **Photoelectric card reader.** In this type of card reader, light passing through the punched holes are detected by photoelectric cells. These are faster and accurate in comparison of other type of card reader.
- **Wire brush card reader.** In this type of card reader, a card is passed between a wire brush and a metal roller. If a punch exists, the brush makes electric contact with the roller and then sends signals to the computer to which the card reader is connected.

Key-Punching Machines

There is another device to punch data on a punch card—*the key punch.* It contains a keyboard which looks like a typewriter keyboard. When characters are typed in the keyboard, corresponding holes are punched in the blank card. Then these cards will be sent to card reader to feed the information to the computer. Commonly used key-punch machines were IBM 026 and IBM 029. These machines have the following components :

- Keyboard
- Card hopper
- Punching station
- Card stacker
- Column indicator
- Backspace key
- Program control unit
- Program drum
- Reading section
- Switches
- Printing mechanism, etc.

Some of these above mentioned components have been shown in Figure 3 and Figure 4.

Keyboard

Keyboard is perhaps the most popular and widely used device for entering data and instructions in a computer system. A keyboard is similar to the keyboard of a typewriter. It contains alphabets, digits, special characters and some control keys. A general purpose keyboard normally contains cursor control keys and function keys. Function keys allow user to enter frequently used operations in a single keystroke, and cursor-control keys can be used to select displayed objects or co-ordinate positions by positioning the cursor on the screen.

NOTES

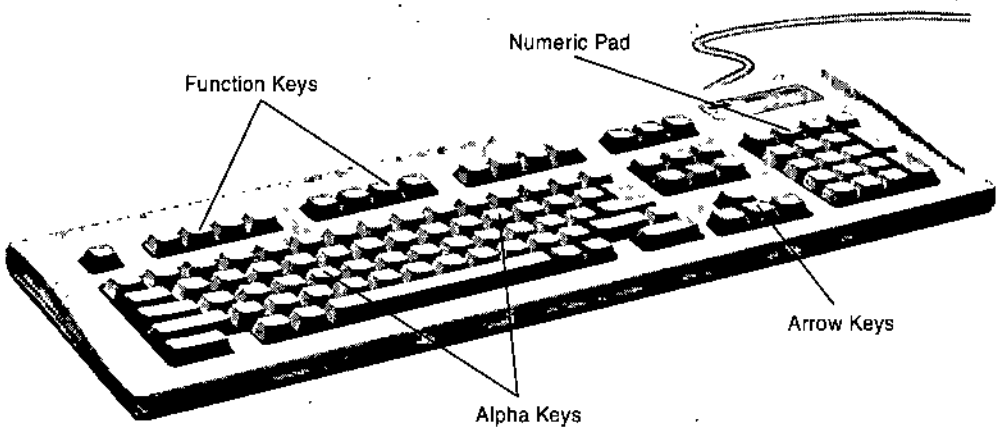


Fig. 6. *A keyboard.*

When a key on the keyboard is pressed an electrical signal is produced. This signal is detected by an electronic circuit called keyboard encoder which is a special IC or a single-chip microcomputer. The encoder detects which key is pressed and sends the binary code, corresponding to the pressed key, to the computer. The encoder contains a lookup table in ROM. The binary code is obtained from the lookup table.

Some of the special keys on a keyboard are given in Table 1.

Table 1. Special keys and their functions

Type	*Purpose*
Arrow Keys	To move the cursor in the top, down, left and right directions in a document.
Backspace Key	To delete the character on the left of the cursor.
Caps Lock	To capitalise letters.
Del	To delete the character from the current position of the cursor.
End Enter	To move the cursor to the end of the line. To start a new paragraph in a document.
Esc	To cancel a command.
Home	To move the cursor to the beginning of the line.
Ins	To insert characters.
Shift	To type the special characters above the numeric keys. If you press this key along with a number key, the special character above that number will be typed. For example : To type "#", you have to press the shift key and the number key 3.
Space Bar	To enter a space.
Tab	To enter multiple spaces between two words in a document.

NOTES

Mouse

A mouse is a pointing device. It is a small hand held box and it is used to position the cursor on the screen. The amount and the direction of movement can be detected by the wheels or the rollers on the bottom of the mouse. The wheels have their axes at right angles. Each wheel is connected to a shaft encoder and whenever the wheel moves this shaft encoder emits electrical pulses. The distance moved is determined by the number of pulses emitted by the mouse.

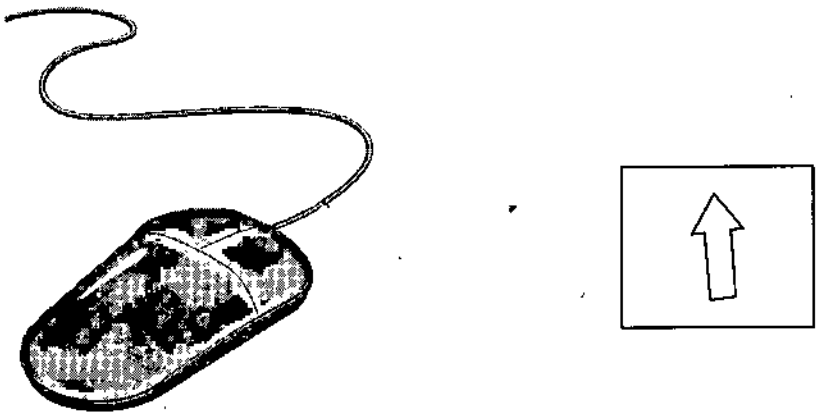


Fig. 7. *A mouse and standard mouse pointer.*

A mouse can be picked up and put down at any position on the screen without change in cursor movement. By moving the mouse the user can point to a menu on the screen. The mouse generally has two or three buttons on its top for indicating the execution of some operation, such as recording cursor position or invoking a function. By pressing the button the user indicates his/her choice to the computer.

The movements of the mouse cursor always match that of the mouse. There are three kinds of clicks. They are left-click, right-click and double click. The mouse can be used to drag and drop objects on the screen.

Note : *It is a good practice to use the mouse pad instead of just any flat surface for movement of the mouse.*

Joystick

A joystick is also a pointing device (See Figure 8). It is used to move the cursor position on the screen. It consists of a small, vertical lever fitted on a base. This lever is used to move the cursor on the screen. The screen-cursor movement in any particular direction is measured by the distance that the stick is shifted or moved from its center position. The amount of movement is measured by the potentiometers that are plugged at the base of the joystick. When the stick is released, a spring brings it back to its center position. The joystick can move right or left, forward or backward.

Trackball

A trackball is also a pointing device (See Figure 9). It consists of a ball which is fitted on a box. The ball can be rotated with the fingers or palm of the hand to move the cursor on the screen. The amount and direction of rotation can be detected by the potentiometers which are attached to the ball. Trackballs are generally fitted on keyboards.

NOTES

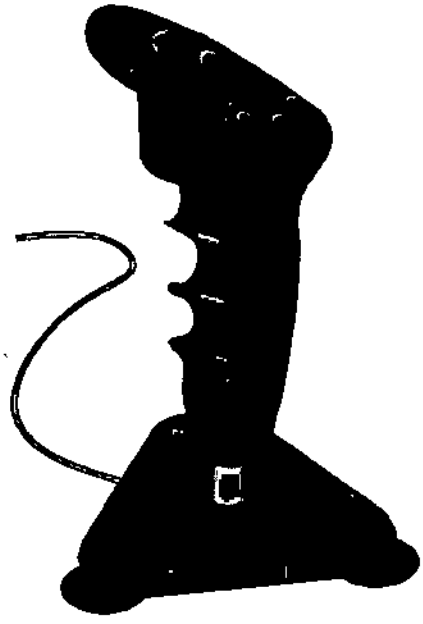


Fig. 8. *A joystick.*

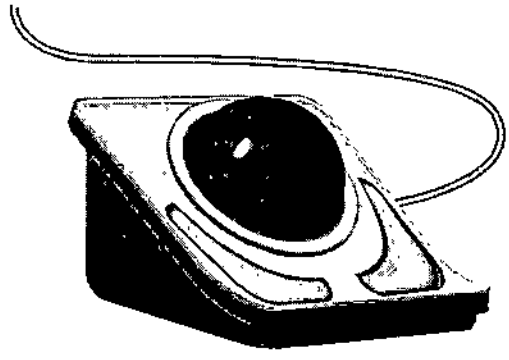


Fig. 9. *A trackball.*

While a trackball is a two divisional positioning devices, a spaceball provides six degrees of freedom. A spaceball does not actually move. The amount and direction of movement is determined by the strain gauges that measure the amount of pressure applied to the spaceball as the ball is pushed and pulled in various directions.

Touch Panels. A touch panel is a very sophisticated and user friendly input devices. It allows the displayed objects or screen positions to be selected by the touch of a finger. There are three methods by which an input to the touch panel can be recorded namely optical, electrical or acoustical methods.

An optical touch panel has a line of infrared light emitting diodes (LEDs) along one vertical edge and along one horizontal edge of the frame. Light detectors are fitted along the opposite vertical and horizontal edges. Now when the panel is touched these detectors take a note of all the beams that are disturbed by the touch. The two crossing beams that are interrupted identify the horizontal and vertical co-ordinates of the screen position selected.

An electrical touch panel is constructed by placing two transparent plates at a small distance. One of the plates is coated with a conducting material, and the

NOTES

other with a resistive material. Now when the outer plate is touched, it is forced into contact with the inner plate. This contact creates a voltage drop across the resistive plate that is converted to the co-ordinate values of selected screen position.

The touch panels are generally used for applications where the processing options are represented with graphical ions.

Light Pen. It is yet another pointing device (See Figure 10). It is used to select screen positions by detecting the light coming from points on the CRT screen. It is a penlike device which is photosensitive. When the tip of the pen touches

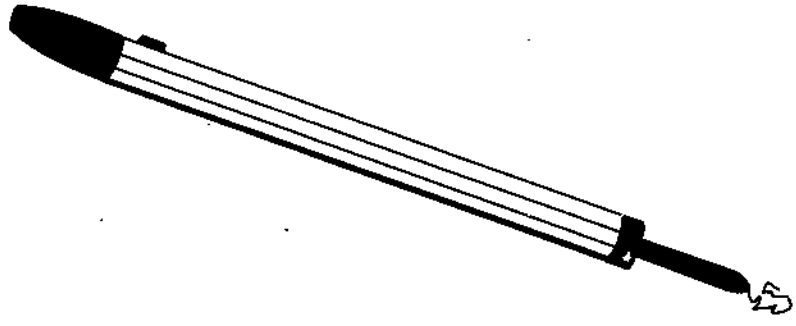


Fig. 10. *An activated light pen with a button switch.*

the screen then the position on the CRT screen is detected by the pen. An activated light pen, pointed at a spot on the screen as the electron beam lights up that spot, generates an electrical pulse that causes the co-ordinate position of electron beam to be recorded. The light pen can be used to draw directly on the CRT screen.

Digitizer

A digitizer is an input device that can also be called a graphics tablet, *i.e.,* a digitizer and a graphics tablet can be one and the same device. There are several versions of the graphic tablet. The basic type consists of an even surface containing a series of parallel wires in the X and Y directions. Conceptually it is very similar to a piece of graph paper. A tablet or digitizer consists essentially of three interconnected parts :

1. A thin flat plate (known as the platen or, confusingly, the tablet) which forms the work surface or active area.
2. A pointing device which can be moved about the platen.
3. A controller which converts the electrical signals arising from the interaction of the pointer and the platen into location information relative to some origin.

The interaction comes about generally through electromagnetic induction, occasionally through differences in electrical resistance or, more rarely, by means of acoustic range-finding techniques. Platens come in a number of sizes, varying

from 11 inches (about 280 mm) square to as much as 60 inches (about 1525 mm) square.

The most usual pointing devices are either a stylus or a puck (multiple button cursor). The puck is probably better for digitizing drawings whilst the stylus is better for pointing, picking and choosing.

NOTES

Digitizing. It is the process of indentifying, locating or selecting a menu item, entity or point through an input device.

Digitizing is used to input a drawing produced on paper into the graphics system. This is accomplished by taping the drawing onto the digitizing tablet as per size and using the input device to locate end points of lines, arcs, etc.

Voice-recognition

A voice-recognition system, using a microphone (or a telephone) as an input device, converts a person's speech into digital signals by comparing the electrical patterns produced by the speaker's voice with a set of prerecorded patterns stored in the computer.

Data entry into a computer manually using keyboard is a time-consuming and laborious task. It will become very easy if we can talk to a computer. Attempts have been made to develop a computer that can listen to the users and talk to them. The voice input to the computer *i.e., voice-recognition* by a computer is much more difficult than the voice output. It is because of the fact that the rules for generating voice through a speaker or a telephone system can easily be defined compared to the rules for interpreting words spoken by a person. The tones of speech, speed, accent and pronunciation differ from person to person. These differences in speech makes voice-recognition a difficult job. In a voice input system the speech is converted into electrical signals employing a microphone. The signals are sent to a processor for processing. The signal pattern is compared with the patterns already stored in the memory. A word is recognised only when a choice match is found, and then the computer gives a corresponding output. At present a voice-recognition system is costly. In future it is expected to become cost effective and will be widely used for direct entry of data. IBM has developed a Talkwriter with 6000 words. It is capable of detecting words with 95% accuracy. It is meant for business correspondence. A voice-recognition system can be used in factories at places where both hands of worker are engaged in the job he is doing and he wants to input some data into the computer. It can also be used to assist bedridden and handicapped persons in a number of tasks; to control access to restricted areas; to identify a customer in a bank etc. Today's programs reach about 98% accuracy at conversational speeds. Two major voice-recognition systems are IBM's Via Voice and L&H's Naturally Speaking.

Touch Screen

Some computers have touch screen which is sensitive to user's touch. One can use finger to point the command displayed on the screen. It is popular on

NOTES

laptops. Many techniques have been used to make the screen sensitive to touch as described below :

(*i*) Capacitive screen uses a device which can sense changes in capacitance when and where the user touches the screen with a stylus or finger.

(*ii*) Infrared screens have light-emitting diodes and photo detector cells to cover the screen with invisible light. LEDs emit infrared light and photo detectors receive it. When the user touches the screen, some light beams are interrupted and the computer then senses the position of the finger.

(*iii*) Pressure-sensitive screens of Mylar, separated by a small space are used. Each sheet of Mylar contains rows of invisible wires. The sheets are placed in such a way that the wires run horizontally in one sheet and vertically in the other. When the user applies pressure on the screen, the wires at that point make contact and a circuit is closed. This is sensed and fed to the computer.

Optical Recognition

Optical recognition occurs when a device scans a printed surface and translates the image the scanner sees into a machine-readable format that is understandable by the computer. The three types of optical recognition devices are given below :

(*i*) Optical Character Recognition (OCR)

(*ii*) Optical Mark Recognition (OMR)

(*iii*) Optical Bar Recognition (OBR)

(*i*) **Optical Character Recognition (OCR).** It uses a device that reads preprinted characters in a particular font (typeface design) and converts them to digital code. OCR characters appear on utility bills and price tags on departmental store items.

(*ii*) **Optical Mark Recognition (OMR).** It uses a device that reads pencil marks and converts them into computer-usable form. The best known example is the OMR technology used to read the various competitive examinations test.

(*iii*) **Optical Bar Recognition (OBR).** It is slightly more sophisticated type of optical recognition. The *bar codes are the vertical zebra-striped marks you see on most manufactured retail products*—every candy to cosmetics to comic books. The usual barcode system in use is called the *Universal Product Code* (UPC). Bar codes represent data, such as name of the manufacturers and the type of product. The code is interpreted on the basis of the width of the lines rather than the location of the bar code. The bar code does not have the price of the product. Bar code readers are photoelectric (optical) scanners that translate the symbols in the bar code into digital code. In this system, the price of a particular item is set within the store's computer. Once the bar code has been scanned, the corresponding price appears on the sales clerk's point-of-sale (POS) terminal and on your receipt.

Scanners

NOTES

They are a kind of Input Device. Scanners are capable of entering information directly into the computer and it is not required to key the information. This makes data entry more interactive, faster and accurate. Examples of scanners are : Optical scanners and Magnetic-ink character readers.

(I) Optical Scanner. Optical scanners use light source and light sensors to read information recorded on a paper. Commonly used optical scanners include Optical Character Reader (OCR), Optical Mark Reader (OMR) and Optical Bar Code Readers (OBCR).

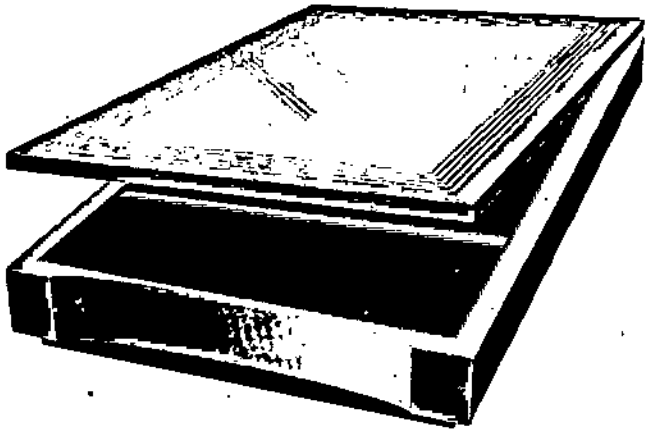


Fig. 11. *A scanner.*

Optical Character Readers (OCR). It is used to recognize alphanumeric characters printed or typewritten on paper. The scanner detects the light reflected from the paper. The change in the reflected light is converted to binary data which is sent to the processor. The paper or text to be scanned is illuminated by low frequency light source. The dark areas on the paper absorb the light whereas light is reflected by lighted areas. The reflected light falls on the photocells which provide binary data corresponding to dark and lighted areas. OCRs are generally used in large-volume applications such as computer-oriented bills.

The ANSI (American National Standard Institute) has adopted a standard type font called OCR-A for use with OCRs shown in Figure 12 :

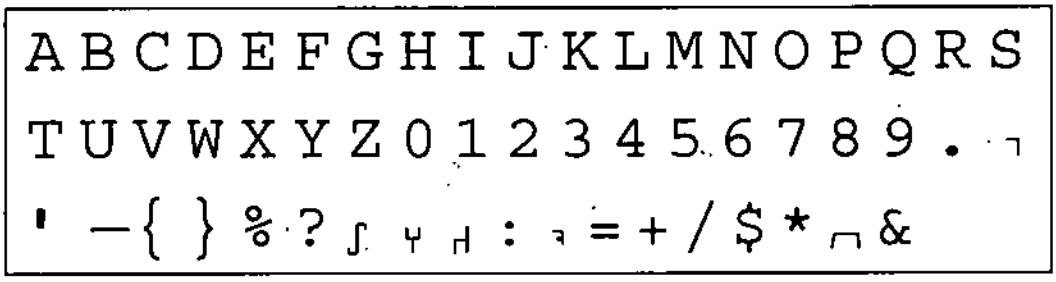


Fig. 12.

NOTES

Optical Mark Readers (OMR). They are commonly used to check special examination answer sheets or questionnaires. The answer sheets contain special marks such as a square or a bubble. These squares or bubbles can be filled with soft pencil or ink. These kind of marked answer sheets are used where one out of a few number of alternatives is to be selected and marked. The sheets are illuminated by a light source. The reflected light is detected by OMR and corresponding signals are sent to the processor. The change in the reflected light is used to detect the presence of a mark. Figure 13 shows the simplest form of optical recording – –Optical marks :

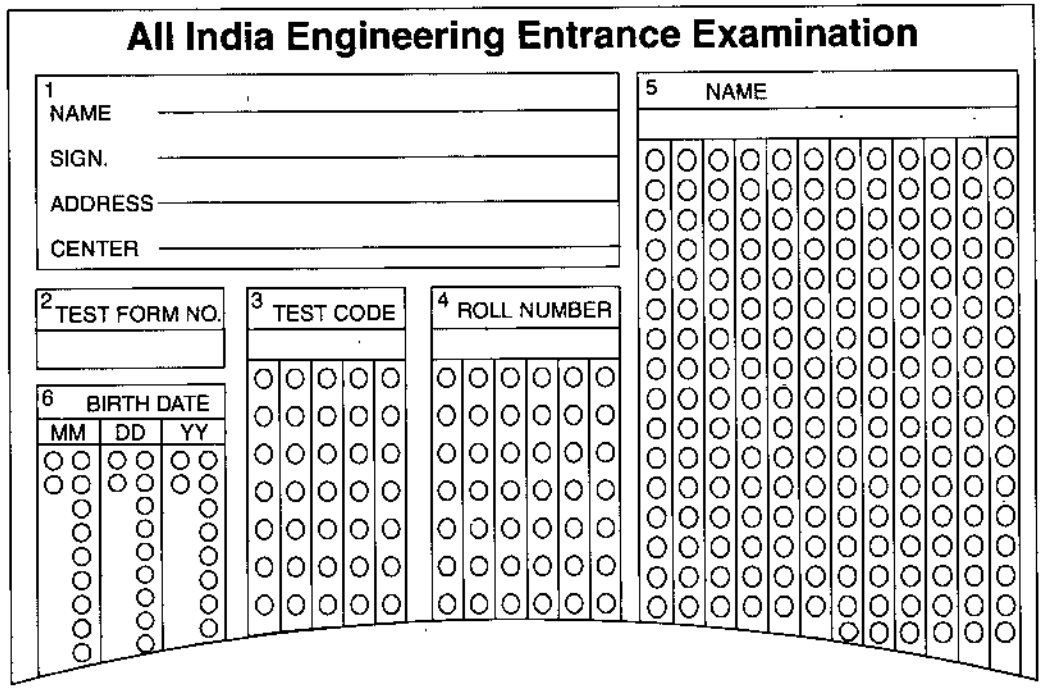
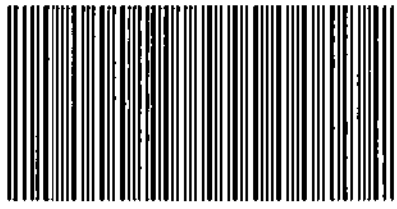


Fig. 13. *Examination answer form for marking (using a no. 2 lead pencil).*

OMRs come in a variety of sizes and shapes depending on the sizes of the forms to be read and the required loading and processing capacity of the reader. OMR devices are easily available for Apple and IBM Compatible personal computers.

Optical Bar Code Readers. This method uses a number of lines (bars) of varying thickness and spacing between them to represent the desired information. Bar codes are used on most grocery items. An OBCR can read such bars and convert them into electrical pulses to be processed by a computer. The most commonly used bar code is Universal Product Code (UPC). The UPC code uses a series of vertical bars of varying widths. These bars are detected at ten digits. The first five digits identify the supplier or manufacturer of the item. The second five digits identify individual product. The code also contains a check digit to ensure that the information read is correct or not.

NOTES



81-87522-11-9

Fig. 14. *A bar-code.*

Magnetic Ink Character Readers (MICR). This device was developed in the late 1950s to assist the banking industry in automation of the process of accounting the bank cheques. MICR devices speed up the processing of input cards and paper documents which are written with a magnetic ink which contains iron oxide particles in it. The characters have a standard configuration which makes them recognizable to humans and at the same time provides signals produced from the read head to electronic circuitry. These signals are analysed to sense the characters used and then these are transmitted to the memory unit. MICR is an example of patter recognition technique and has successfully replaced the time consuming and expensive punched card processing. Human involvement is required to encode the cheque amount and other descriptions—thus, some room for error does remain. Figure 15 shows the layout of MICR encoding on a personal cheque.

PAY _____ OR BEARER

RUPEES _____ Rs.

A/c. NO. | TR NO. | INTLS

Bank of Punjab Ltd.
10166–67, Gurdwara Road,
Karol Bagh,
New Delhi - 110 005

DHS - KRB

"222287" 110237003: 10

Fig. 15. *MICR encoding on a cheque.*

Terminals. Terminals are much more limited than the personal computers although they look like them. Terminals have only a screen and a keyboard and the electronics that allow them to communicate with the computer to which they are connected and are used only to send information to the computer and receive information from it. Here we will discuss *dumb terminal, intelligent terminals* and *Internet terminals.*

NOTES

- **Dumb terminal.** It is also known as Video Display Terminal (VDT), has a display screen and a keyboard and can input and output but not process data. Usually the output is text only. For instance, airline reservations clerks use these terminals to access a mainframe computer having flight information. Dumb terminals cannot perform functions independent of mainframe to which they are linked.
- **Intelligent terminal.** It has its own memory and processor, as well as a display screen and keyboard. Such a terminal can perform some functions independent of any mainframe to which it is linked. For example, an *automated teller machine (ATM)*, a self service banking machine that is connected through a telephone network to a central computer. Another example is the *point-of-sale (POS)* terminal, used to record purchase at a store's checkout counter.
- **Internet terminal.** It provides access to the Internet. There are several variants which are given below :
  1. The *set-top box* or *web terminal* : It displays web pages on a TV set.
  2. The *network computer* : It is a cheap stripped-down computer that connects people to networks.
  3. The *online game player* : It not only lets you play games but also connects to the Internet.
  4. The full-blown *PC/TV* (or *TV/PC*) : It merges the personal computer with the television set.
  5. The *wireless pocket PC* or *personal digital assistant (PDA)* : It is a hand held computer with a tiny keyboard that can do two-way wireless messaging.

Figures 16 and 17 show a monitor (or VDU) and Terminal respectively :

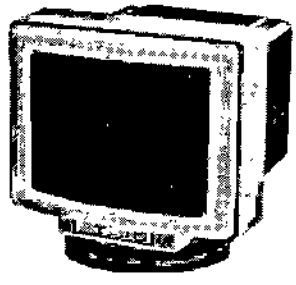


Fig. 16. *Monitor (or VDU).*

Fig. 17. *Terminal.*

2.5 OUTPUT DEVICES

An output device is a device which accepts results from the computer and displays them to the user. The output device also converts the binary code obtained from the computer into human readable form.

NOTES

The Meaning of Hard Copy and Soft Copy

Hard copy output is a computer output, which is permanent in nature and can be kept in paper files, or can be looked at a later stage, when the person is not using the computer. For example, output produced by printers or plotters on paper.

Soft copy output is a computer output, which is temporary in nature, and vanishes after its use. For example, output shown on a terminal screen, or spoken out by a voice response system.

The commonly used output devices are : CRT screen, printers and plotters.

2.5.1 Hard Copy Devices

Hard copy is printed output. For example, printouts, whether text or graphics, from printers. Film, including microfilm and microfiche is also considered hard copy output. The *hard copy output* devices are printers.

Print Quality

We can have a considerable variety in the quality of hard copy. Some of the print qualities are given below :

- **Near-typeset quality.** This type of print is similar to what is produced by a typeset machine, such as the print of a magazine.
- **Letter-quality.** This type of print is the equivalent of good typewriter print. It is made using solid-line (fully formed) characters rather than characters made up of dots or lines. This is used mainly in business letters and in formal correspondence between persons.
- **Near-letter quality.** All the printers don't produce the fully formed characters but print high-quality documents using a near-letter quality print. This is done by some printers when the print head makes multiple passes over the same letters, filling in the spaces between the dots or lines being printed.
- **Standard-quality.** This type of print is provided by the printer when characters composed of dots or lines are formed by a single pass of the print head. In general, standard-quality suits for most informal applications.
- **Draft-quality.** The *draft-quality* print lies at the low end of the quality scale and sometimes known as compressed print. It is sometimes used for rough drafts, informal correspondence, or computer program printouts. The characters are formed with a minimum number of dots or lines and are smaller in size than the standard-quality characters.

Printers

A printer is an output device that prints characters, symbols and perhaps graphics on paper or another hard copy medium. The resolution or quality of sharpness of the image, is indicated by *dpi (dots per inch)*, which is a measure of the dots

NOTES

that are printed in a linear inch. For PC printers, the resolution is in the range 60–1,500 *dpi*. They provide information in a permanent readable form. There are a variety of printers available for various types of applications. Depending upon the speed and approach of printing, printers can be classified as :

(*i*) Character printers

(*ii*) Line printers

(*iii*) Page printers

There is yet another classification depending upon the technology used for printing (whether or not the image produced is formed by physical contact of the print mechanism with the paper). According to this classification, printers are of two types :

- **Impact printers**—do have contact with paper.
- **Non-impact printers**—do not have contact with paper.

Character Printers. Character printers print only one character at a time. They are low-speed printers and are generally used for low volume printing work. Characters to be printed are sent serially to the printer. Three of the most commonly used character printers are described below :

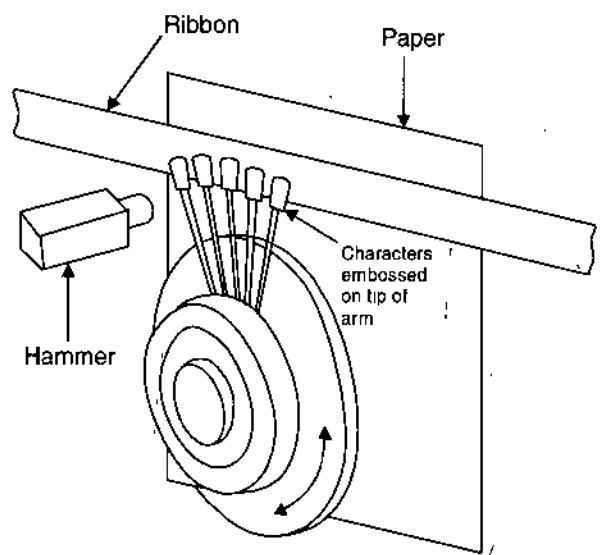


Fig. 18. *A daisy wheel printer.*

Better Quality Printers. Better quality printers are used where good printing quality is needed. These printers use a print wheel font known as a daisy wheel. There is a character embossed on each petal of the daisy wheel. The wheel is rotated at a rapid rate with the help of a motor. In order to print a character, the wheel is rotated. When the desired character spins to the correct position, a print hammer strikes it to produce the output. Thus, daisy wheel printers are impact printers. Its speed is in the range 10–90 CPS (Characters Per Second). It has a fixed font type. Normally 2 or 3 fonts are available. It cannot print graphics.

NOTES

Dot-matrix Printers (DMPs). A dot-matrix printer prints the characters as a pattern of dots. The print head contains a vertical array of 7, 9, 14, 18 or even 24 pins. A character is printed in a number of steps. One dot column of the dot-matrix is taken up at a time. The selected dots of a column are printed by the print head at a time as it moves across a line. The shape of each character *i.e.,* the dot pattern is detained from the information held electronically in the printer.

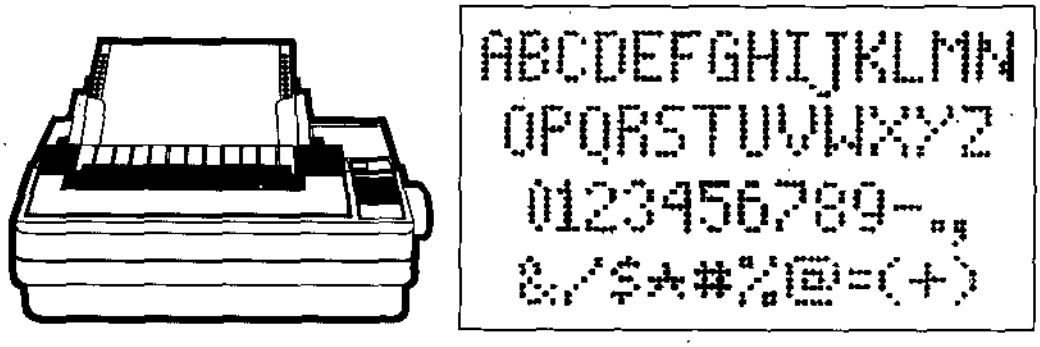


Fig. 19. *A dot-matrix printer and its character pattern.*

The dot-matrix printers are faster than daisy wheel printers. This speed lies in the range of 30–600 cps. But the print quality of a dot-matrix printer is low as compared to that of a daisy wheel printer. Dot-matrix do not have fixed character font. So they can print any shape of character. This allows for many special characters, different sizes of print and the ability to print graphics such as graphs and charts. *These are the only printers that can use multilayered forms to print "carbon copies".*

Inkjet Printers. Inkjet printers are non-impact printers. They employ a different technology to print characters on the paper. They print characters by spraying small drops of ink onto the paper. Special type of ink with high iron content is used. Each droplet is charged when it passes through the valve. Then it passes through a region having horizontal and vertical deflection plates. These plates deflect the ink drops to the desired spots on the paper to form the desired character.

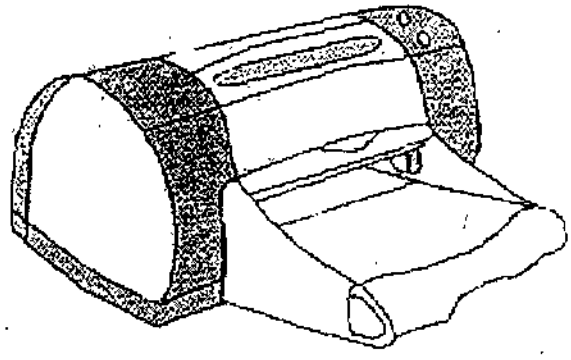


Fig. 20. *An inkjet printer.*

Inkjet printers produce high quality printing output. The speed of inkjet printers lies in the range of 40–300 cps. They allow all sorts of fonts and styles.

NOTES

Therefore, the document printed may contain multiple character styles and a variety of font sizes. Colour printing is also possible by using different coloured inks.

Line Printers. As the name suggests, a line printer prints one line of the text at a time. They are impact printers and are used for producing high volume paper output. They are fast printers and the printing speed lies in the range of 300–3000 lines per minute. Drum printer and chain printer are the most commonly used line printers.

Drum Printer. A drum printer consists of a solid, cylindrical drum which contains complete raised characters set in each band around the cylinder. The number of bands is equal to the number of printing positions. Each band contains all the possible characters. The drum rotates at a rapid speed. There is a magnetically driven hammer for each possible print position. The hammers hit the paper and the ribbon against the desired character on the drum when it comes in printing position. The speed of a drum printer is in the range of 200 to 2000 lines per minute.

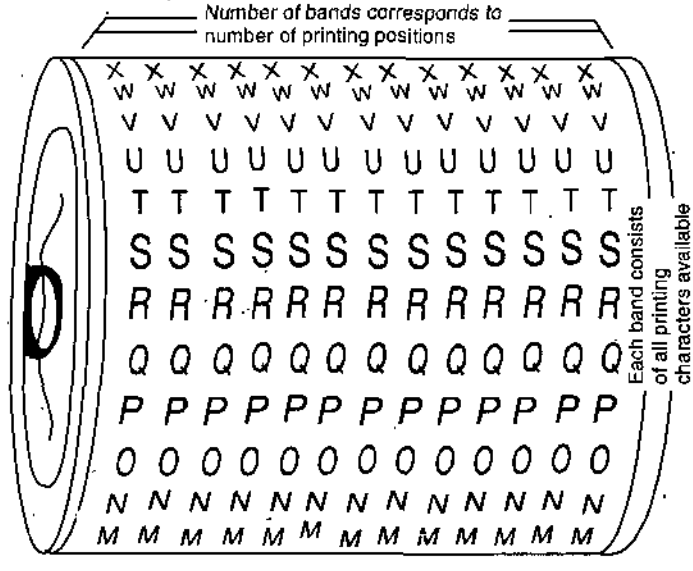


Fig. 21. *A drum printer.*

Chain Printers. Chain printers use a rapidly rotating chain which is called print chain. The print chain contains characters. Each link of the chain is character font. There is a magnetically driven hammer behind the paper for each print position. The processor sends all the characters to be printed in one line to the printer. When the desired character comes in the print position the hammer strikes the ribbon and paper against the character. A chain may contain more than one character set, for example, 4 sets. The speed lies in the range of 400–2400 lines per minute.

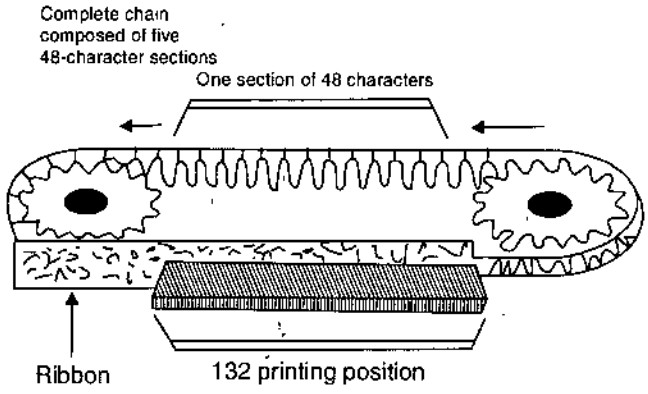


Fig. 22. *The print chain of a chain printer.*

Page Printers. Page printers are non-impact printers. They print one page at a time at a very high speed of 2,000 lines per minute. They are very costly and are economical only when printed volume is very high. Page printers are based on a number of technologies like electronics, xenography and lasers. These techniques are called Electro-photographic techniques. In these printers, an image is produced on a photosensitive surface using a laser beam of other light source. The laser beam is turned off and on under the control of a computer. The areas that are exposed to the laser attract toner, which is generally an ink power. Thereafter, the drum transfer the toner to the paper. Then the toner is permanently fused on the paper with heat or pressure in a fusing station. After this drum is discharged, cleaned so that it is ready for next processing. They can produce 300 pages per minute.

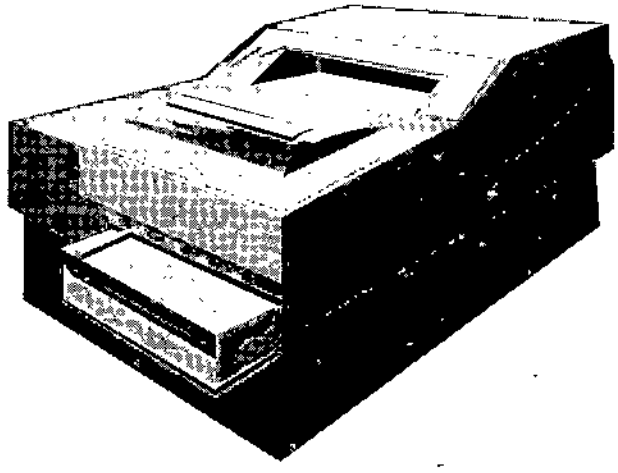


Fig. 23. *A laser printer.*

The differences between a dot-matrix printer and a laser printer are given in Table 2.

NOTES

Table 2. Differences between dot-matrix and laser printer

Dot-matrix printer	*Laser printer*
A dot-matrix printer prints characters using dots.	A laser printer prints characters completely.
The speed is measured in characters per second.	The speed is measured in pages.
It prints approximately 200–300 characters in one second.	It prints approximately 4–20 pages in one minute.
It is quite noisy.	It is not very noisy.
It is cheap.	It is expensive.

Thermal Printers

These printers are a variation of the non-impact dot-matrix type in which selected needles are pressed against heat sensitive paper in a dot-matrix method for formation of characters. It is not possible to have mass printing with ordinary dot-matrix or other impact printers. However, there is a very little noise associated with thermal printers. The advantage of this type of printer over the dot-matrix type is that the thermal unit is much quieter. These provide high quality colour output. The disadvantages are that a special type of paper must be used and it is not possible to produce multiple copies. These are expensive and slow also.

LED Printers

The Light Emitting Diode (LED) or Liquid Crystal printers use LEDs which are cheaper alternatives of laser printers. Here, LEDs are used to produce image on the drum rather than a laser beam. For example, Epson EPL 5200 Printer.

Plotters

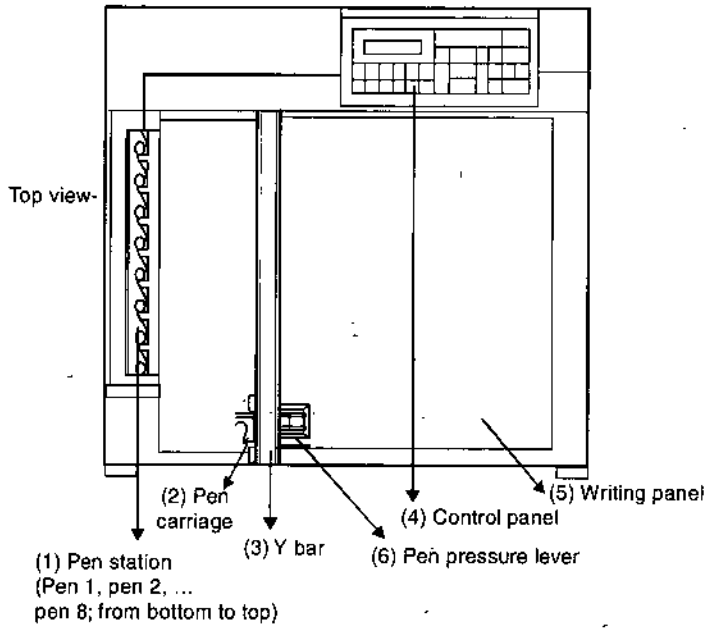
A plotter is an output device used to produce hard copies of graphs and designs. They use ink pen or inkjet to draw graphics and drawings. Pens could be monochrome or multi-coloured. Plotters are slow devices because a lot of mechanical movement is required during plotting. The graphics and designs produced by pen plotters are uniform and precise and are of very good quality. Pen plotters are basically of two types : Drum plotters and Flatbed plotters (use pens) and Electrostatic plotters (do not use pens).

Drum Plotters. In case of a drum plotter there is a drum that moves back and forth to produce vertical motion. The paper on which the design has to be made is placed on this drum. A pen is mounted horizontally across the drum in a pen carriage. The pen moves horizontally along with the carriage

left to right or right to left on the paper to produce drawings. Coloured drawing can also be produced by using multi-coloured pens.

Flatbed Plotters : In the case of a flatbed plotter, a paper is spread and fixed over a rectangular flatbed table. This paper is fixed and does not move. A pen-holding mechanism is designed to provide all the motion. Multi-coloured graphs and designs can be produced by using pens with multi-coloured inks.

NOTES



(1) Pen station—holds pens ready for use.

(2) Pen carriage—holds the pen used for plotting.

(3) Ybar—moves the pen carriage to left and right.

(4) Control panel—has keys to control the operation of the plotter and lamps which indicate the status of the plotter.

Fig. 24. *A flatbed plotter and its different parts.*

Electrostatic Plotters. These use electrostatic charges to create images out of very small dots on specially treated paper. The paper is run through a developer to allow the image to appear. These are faster than pen plotters and can produce images of very high resolution. Figure 24shows a flatbed plotter. The cost of a plotter can range from about $1000 to more than $100000 depending on the machine's speed and quality of images. Large plotters, used with large computer systems, can produce drawings upto 8' × 8', or sometimes even larger.

2.5.2 Softcopy Devices

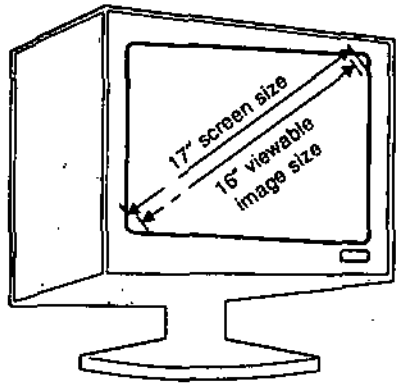
Output hardware consists of devices that convert machine-readable information as the result of processing, into human-readable form. The principal kinds of output are *softcopy* and *hardcopy*. The softcopy devices are CRT display screens, Flat-panel display screen (for example, liquid-crystal display).

NOTES

- **Softcopy.** Softcopy is data that is shown on a display screen or is in audio or voice form. This kind of output is not tangible; it cannot be touched. (Actually, you almost never hear the word "softcopy" used in real life).

The *hardcopy* and the related output devices have been discussed earlier. Let us discuss the softcopy devices.

Monitors. These are also known as display screens, CRTs, or simply screens—are output devices that show programming instructions and data as they are being input and information after it is processed. The size of a computer screen is measured diagonally from corner to corner in inches. For desktop microcomputers, the most common sizes are 13, 15, 17, 19 and 21 inches; for laptop computers, 12.1, 13.3 and 14.1 inches. Figure 25 illustrates some sizes of computer screen :



Monitor screen size	Viewable image area
15 inches	14 inches
17 inches	16 inches
21 inches	20 inches

Fig. 25. *Illustrating sizes of computer screen.*

In deciding which display screen to opt, you will require issues of

screen clarity (dot pitch, resolution and refresh rate),

types of monitor (CRT versus flat panel, active-matrix flat panel versus passive-matrix flat panel), and

color and resolution standards (SVGA and XGA).

- **Screen clarity—dot pitch, resolution, and refresh rate.** Major factors affecting screen clarity (often mentioned in ads) are *dot pitch, resolution* and *refresh rate.* These relate to the individual dots known as pixels, which represent the images on the screen. A ***pixel,*** for "picture element", is the smallest unit on the screen that can be turned on and off or made different shades.

  Dot pitch (dp) is the amount of space between the centers of adjacent pixels; the closer the dots, the crisper the image. For a .28dp monitor, for instance, the dots are 28/100ths of a millimeter apart. Generally, a dot pitch of .28dp will provide clear images.

  Resolution is the image sharpness of a display screen; the more pixels there are per square inch, the finer the level of detail attained. Resolution is expressed in terms of the formula *horizontal pixels* × *vertical pixels.* Each pixel can be assigned a colour or a particular shade of gray. Standard

resolutions are 640 × 480, 800 × 600, 1024 × 768, 1280 × 1024, 1600 × 1200 and 1920 × 1440 pixels.

Refresh rate is the number of times per second that the pixels are recharged so that their glow remains bright. In general, displays are refreshed 45–100 times per second. *The higher the refresh rate, the more solid the image looks on the screen—that is, the less it flickers.* Refresh rate is measured in *hertz;* a high-quality monitor has a refresh rate of 75 hertz—the screen is redrawn 75 times per second.

NOTES

Type of Monitors

The two types of monitors are CRT and flat-panel.

CRT. A CRT, for cathode-ray type, is a vaccum tube used as a display screen in a computer or video display terminal. Figure 26 illustrates CRT display.

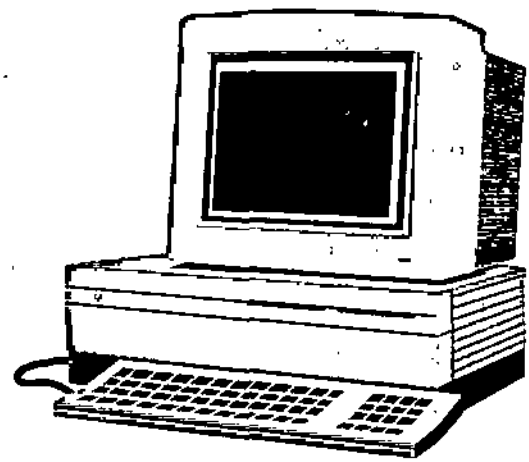


Fig. 26. *CRT display.*

The same kind of technology is found not only in the screens of desktop computers but also in television sets and flight-information monitors in airports. The CRTs are considerable cheaper (5–10 times) than flat-panel displays.

Figure 27 illustrates the basic operation of a CRT. A beam of electrons, emitted by an electron gun, passes through focussing and deflection systems that direct the beam towards specified positions on the phosphor-coated screen. The phosphor then emits a small spot of light at each position contacted by the electron beam. The light emitted by the phosphor fades away rapidly. Therefore, some mechanism is needed to maintain the screen picture. One way to retain the picture is to redraw the picture repeatedly by quickly directing the electron beam back over the same points. This type of display is also called refresh CRT.

A CRT display comes is two varieties. Monochrome (only one colour) and colour (multicolour). Monochrome displays come in green, blue, orange, yellow, pink, amber, red and white depending upon the type of phosphor material used.

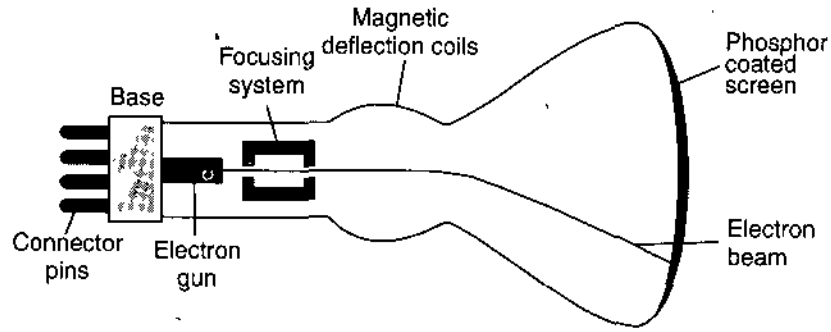


Fig. 27. *Basic design of magnetic deflection CRT.*

Coloured displays are developed by using a combination of phosphors that emit different coloured light. To produce colour display three phosphors : red, blue and green are used. These three phosphor colour dots are put at each pixel position. One phosphor dot emits a red light, another emits a green light and the third emits a blue light. Three separate electron beams are employed to illustrate the dots of three different phosphors. By varying the intensity of the three electron beams the intensity of red, blue and green dots is varied. This gives the appearance of a triangular spot of desired colour.

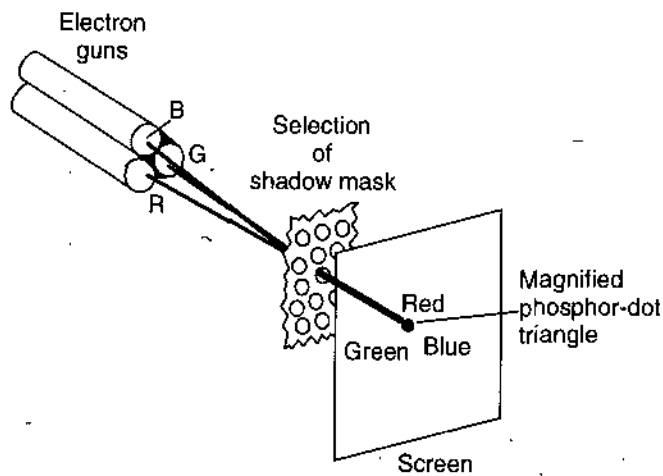


Fig. 28. *Operation of three electron gun CRT.*

2.5.3 Flat-panel Displays

Compared to CRTs, flat-panel displays are much thinner, weigh less and consume less power. Thus, they are better for portable computers, although they are available for desktop computers as well.

Figure 29 illustrates a flat-panel display :

Flat-panel displays are made up of two plates of glass separated by a layer of a substance in which light is manipulated. One technology used is *liquid crystal display (LCD), in which molecules of liquid crystal line up in a way that alters their optical properties, creating images on the screen by transmitting or blocking out light.*

NOTES

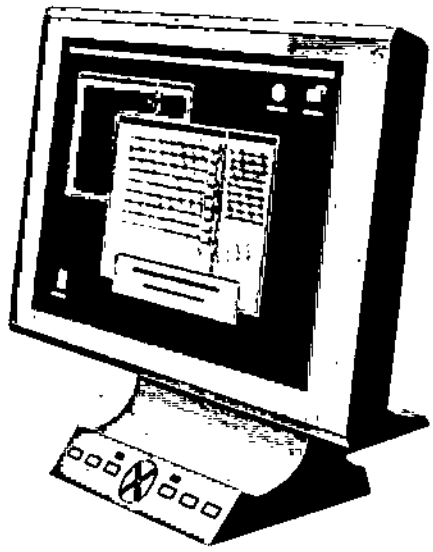


Fig. 29. *A flat-panel display.*

Advantages

- Lower power consumption.
- Cover less space than conventional monitors.
- Reduction in cooling load (as these radiate less heat).
- High performance monitors.
- Flexibility of usage.
- More viewing area.

Types of Flat-panel Displays

- **Active-matrix versus passive-matrix flat-panel displays.** Flat-panel screens are either active-matrix or passive-matrix displays, according to the location of their transistors.

  In an *active-matrix display, also known as TFT (thin-film transistor) display, each pixel on the screen is controlled by its own transistor.* Active-matrix screens are much brighter and sharper than passive-matrix screens, but they are more complicated and thus more expensive. They also need more power, affecting the battery life in laptop computers.

  In a *passive-matrix display, a transistor controls a whole row or column of pixels.* Passive matrix provides a sharp image for one-colour (monochrome) screens but is more subdued for colour. The advantage is that passive-matrix displays are less expensive and use less power than active-matrix displays, but they aren't as clear and bright and can leave "ghosts" when the display changes quickly. Passive-matrix displays go by the abbreviations HPA, STN or DSTN.

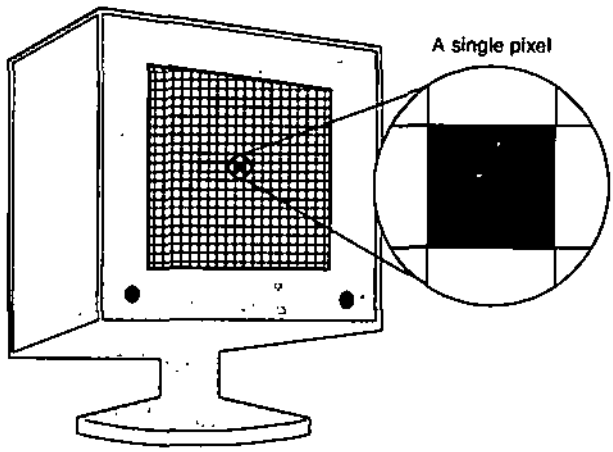
NOTES

Video Standards

PCs have *graphics cards* (also known as *video cards* or *video adapters*) that convert signals from the computer into video signals that-can be displayed as images on a monitor. The monitor then separates the video signal into three colour : red, green and blue signals. Inside the monitor, these three colours combine to make up each individual pixel. Video cards have their own memory, *video RAM* or *VRAM*, which stores the information about each pixel.

The common colour and resolution standards for monitors are *VGA, SVGA, XGA, SXGA* and *UXGA*. Figure 30 illustrates comparison of video standards :

VGA (Video Graphics Array). It was developed by IBM for PCs. In graphics mode, the resolution is either 640 × 480 or 320 × 200 with 16 colours and 256 colours respectively. In text mode, VGA systems provide a resolution of 720 × 400 pixels. The total palette of colours is 2,62,144. It uses analog signals rather than digital signals.



Video standard	Principal resolution in pixels
VGA	640 × 480
VGA	320 × 200
VGA	720 × 400
SVGA	800 × 600
XGA	1024 × 768
SXGA	1280 × 1024
UXGA	1600 × 1200

Fig. 30. *Video graphics standards compared for pixels.*

SVGA (Super Video Graphics Array). *It supports a resolution of 800 × 600 pixels, or variations, producing 16 million possible simultaneous colours,* but the number of colours than can be displayed simultaneously depends upon the amount of video memory installed in a computer. SVGA is the most common standard used today with 15-inch monitors.

XGA (Extended Graphics Array). *It has a resolution of up to 1024 × 768 pixels, with 65,536 possible colours.* It is used mainly with 17-inch and 19-inch monitors.

SXGA (Super Extended Graphics Array). *It has a resolution of 1280 × 1024 pixels.* It is often used with 19-inch and 21-inch monitors.

UXGA (Ultra Extended Graphics Array). *It has a resolution of 1600 × 1200 pixels.* It is expected to become more popular with graphic artists, engineering designers and other using 21-inch monitors.

2.6 PRIMARY AND SECONDARY MEMORIES

NOTES

A computer is capable to storing bulk of data and retrieving or accessing the stored data as and when required. A personal computer may store a few thousand of characters whereas a mainframe may store billion of characters. The bulk of data can't be stored in the main memory as this memory is costly and naturally some other cheaper memory devices are required. These cheaper memory devices, called SECONDARY STORAGE DEVICES can store bulk of data at very less cost. Data are stored in secondary storage in the same binary codes as in the main storage and are made available to main storage as needed. The commonly used secondary storage devices are Magnetic Tape, Floppy Disk, Hard Disk and CD-ROM.

From the above discussion on memory we conclude that computers use two types of storage or memory. Figure 31 illustrates the two types of memories.

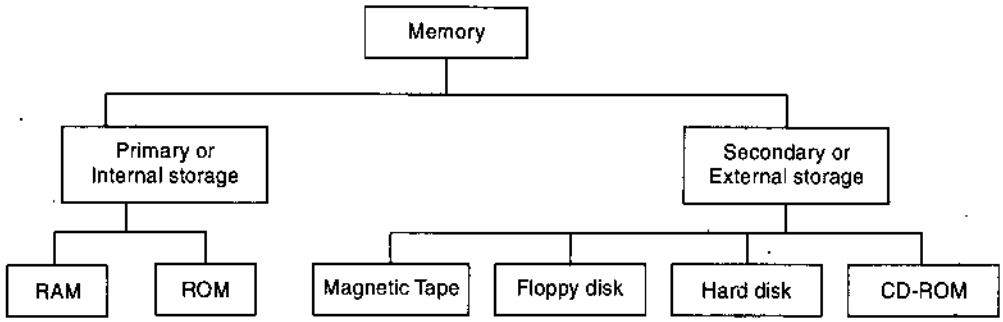


Fig. 31. *Illustration of memory in a computer.*

The choice of a particular secondary storage device for a given application mainly depends upon how the stored information needs to be accessed. Basically there are two methods of accessing information.

(*i*) Sequential or Serial Access

(*ii*) Direct or Random Access

A Sequential Access Device is one in which the data can be retrieved in the same sequence in which it is stored, so that the access time varies according to the location of storage. Sequential processing is quite suitable for applications like preparation of monthly payslips, monthly electrically bills etc. In these applications, each record needs to be processed at scheduled intervals. Magnetic tape and punched paper media are widely used examples of sequential access storage devices.

In many applications we need to retrieve a record directly rather than going through all the records to reach the desired one. For example : In a computerised bank, we might need to check the account status of a particular customer at any instant of time. In such applications, it is inefficient to use a sequential storage device as most of the time is wasted in looking for the particular

NOTES

record. For such online real time processing applications Direct Access Storage Devices are used. These direct access devices are also called Random-Access devices because the information is available randomly. Thus, a Direct Access Storage Device is a device in which the data can be stored randomly and it can be accessed directly. Magnetic Disk and Magnetic Drum are typical examples of Direct-Access Storage Devices.

2.6.1 Primary Memories

Let us first discuss the primary memories :

RAM—Random Access Memory

Random Access Memory is also known as primary storage; and it temporarily store

1. Software instructions and
2. Data before and after it is processed by the CPU.

Because its contents are temporary, RAM is said to be **volatile** as *the contents are lost when the power goes off or is turned off.* This is the reason why you should frequently—every 5–10 minutes, say—transfer (save) your work on a secondary-storage device such as your hard disk, in case the electricity goes off while you are working.

Four types of RAM chips are used in PCs which are given below :

- **DRAM.** Pronounced *"dee-ram", DRAM (dynamic RAM)* must be constantly refreshed by the CPU or it will lose its contents.
- **SDRAM.** The type of dynamic RAM used in most PCs today is SDRAM *(synchronous dynamic RAM),* which is synchronized by the system clock and is much faster than DRAM. Often in computer advertisements, the speed of SDRAM is expressed in megahertz.
- **SRAM.** Pronounced *"ess-ram", SRAM (static RAM)* is faster than any DRAM and will retain its contents without having to be refreshed by the CPU.
- **RDRAM.** *Rambus dynamic RAM, or RDRAM,* is faster and more expensive than SDRAM and is the type of memory used with Intel's P4 chip.

Microcomputers come with different amounts of RAM, which is usually measured in megabytes. The more RAM you have, the faster the computer operates, and the better your software performs. *Having sufficient RAM is a critical matter. Microsoft* Office 2000, for example, states that a minimum of 16 megabytes of RAM is required.

If you are short on memory capacity, you can usually add more RAM chips by plugging them into the motherboard. Chips can be brought single or in so-called *memory modules,* circuit boards that can be plugged into expansion slots on the motherboard. There are two types of such modules :

NOTES

SIMMs and DIMMs

Both of these are DRAM chips. A SIMM *(single inline memory module)* has RAM chips on only one side. A DIMM *(dual inline memory module)* has RAM chips on both sides.

ROM—Read Only Memory

Unlike RAM, to which data and instructions are constantly being added and removed, *ROM (Read Only Memory) cannot be written on or erased by the computer user without special equipment. ROM chips have fixed start-up instructions.* That is, ROM chips are loaded, at the factory, with programs having special instructions for basic computer operations sometimes called *firmware,* such as those that start the computer or put characters on the monitor. These chips are *nonvolatile; their contents are not lost when power to the computer is switched off.*

In computer terminology, *read means to transfer data from an input source into the computer's memory or CPU.* The opposite is *write-to transfer data from the computer's CPU or memory to an output device.* Thus, with a ROM chip, "read-only" means that the CPU can retrieve programs from the ROM chip but cannot modify or add to these programs.

The ROM can be further classified as :

PROM. A PROM is a programmable ROM. ROM chips are provided by the computer manufacturers and it is not possible for the user to change the contents of a ROM chip. However, in a PROM the contents are decided by the user. The user can store permanent programs, data or any other kind of information in a PROM. PROMs are programmed to store information using a facility known as PROM programmes. However, once the chip has been programmed the recorded information cannot be changed *i.e.,* PROM becomes a ROM. So PROM is also a permanent storage.

EPROM. A variation to PROM is EPROM which stands for erasable PROM. As the name suggests it is possible to erase the contents of a EPROM chip unlike a PROM chip. The stored data in EPROMs is erased by exposing it to high intensity short wave ultraviolet light for about 20 minutes. EPROMs are used to store programs which are permanent but need updating.

EEPROM. EEPROM is an electrically erasable PROM. The chip can be erased and reprogrammed on byte by byte basis. Hence selective erasing is possible. Its disadvantage is that it requires different voltages for erasing (21 V), writing (21 V) and reading (5V) the stored information. It also has high cost and low reliability.

FLASH EPROM. This is the latest type of ROM, which is becoming very popular. Using a special program, a manufacturer can modify the contents of the flash EPROM while it remains in the computer.

NOTES

2.6.2 Secondary Memories

Now let us discuss the secondary memories.

Floppy Disks

A floppy disk is a very popular direct access secondary storage medium for micro and mini computers.

A floppy disk, often called a diskette or simply a disk, is a removable flat piece of mylar plastic packaged in a 3.5-inch plastic case. Data and programs are stored on the disk's coating by means of magnetized spots, following standard on/off patterns of data representation (such as ASCII). The plastic case protects the mylar disk from being touched by human hands. Originally, when most disks were larger (5.25 inches), the disks actually were **"floppy",** not rigid; now the plastic disk inside is flexible or floppy.

Floppy disks are inserted into a floppy-disk drive, a device that holds, spins, reads data from, and writes data to a floppy disk. **Read** means taking data from secondary storage (converted to electronic signals) to the computer's memory (RAM). **Write** means copying the electronic information processed by the computer to secondary storage.

On the diskette, data is recorded in concentric circles called **tracks.** On a formatted disk each track is divided into **sectors, invisible wedge-shaped sections used for storage reference purpose.** The read/write head is used to transfer data between the computer and the disk. Figure 32 illustrates a 3½ inch floppy disk :

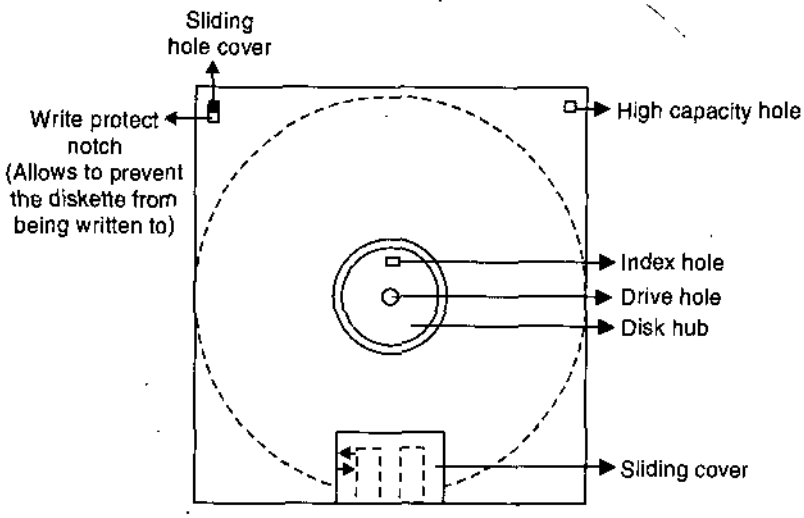


Fig. 32. *3½ inch floppy disk.*

The Floppy Drive is of a moving head variety and therefore the floppy can be removed and replaced by another. The head actually contacts the surface during reading/writing, though in other times it is lifted up from the surface. The hole at the centre is to allow a spindle to lock the floppy so that it can

NOTES

rctate. The Index hole is used to recognize the starting sector of any track. The purpose of write permit notch is to protect valuable information recorded on the floppy from accidental damage. If this notch is covered, writing is not allowed on the floppy, only reading is possible. If the notch is not covered reading as well as writing is possible.

A new floppy can't be used without formatting it. Formatting a disk is to create a set of magnetic concentric circles called tracks. Tracks are further divided into sectors. Most high density disks have 80 tracks.

Floppies offer a number of advantages. They are exchangeable. The storage capacity is high as compared to its size and weight. These are portable. Floppies are inexpensive also as compared to hard disks. The most common uses of floppy disks are as follows :

(*i*) Moving files between computers that are not connected through communication channel.

(*ii*) Loading new programs on to a system.

(*iii*) Backing up data or programs, the primary copy of which is stored on hard disk.

***Note** : Do not remove the disk when the access light is on.*

Let us compare the 3.5 inch floppy disk with some 3.5 inch **floppy-disk cartridges, or higher-capacity removable disks**—Zip disks, SuperDisks and HiFD disks :

- **3.5 inch floppy-disks—1.44 megabytes :** The current standard for traditional floppy disks is 1.44 megabytes, the equivalent of 400 type-written pages. Today's floppy carries the label 2HD, in which the 2 stands for "double-sided" (it stores data on both sides) and the HD stands for "highdensity" (which means it stores more data than the previous standard—DD, for "double density").
- **Zip disks—100 or 250 megabytes :** These are special disks with a capacity of 100 or 250 megabytes, produced by Iomega Corp. At 100–250 megabytes, this is at least 70 times the storage capacity of the standard floppy. These are used to store large spreadsheet files, database files, image files, websites and multimedia representation files. These require their own Zip disk drives, which may come installed on new computers, although external Zip drives are also available in market.
- **Super Disks—120 megabytes :** These are disks with a capacity of 120 megabytes, produced by Imation. The Super Disk drive can also read standard 1.44 megabyte floppy disks, which Zip drives cannot do.
- **HiFD Disks—200 megabytes :** These are disks with a capacity of 200 megabytes, produced by Sony Corp. The disk drive can also read standard 1.44 megabytes floppies. These have 140 times the capacity of today's standard floppy disks.

NOTES

Hard Disk

Magnetic disk is the most popular direct access storage medium. A magnetic disk is made of aluminium or other metals or metal alloys instead of plastic. The disk is coated on both sides with magnetic material (iron oxide). Unlike a floppy disk, a hard disk cannot be inserted or removed from the hard disk drive. A disk drive is a device that writes information on recording platters that resemble gramophone records. Disk drive reads information written on to the disk. In order to increase the storage capacity a large number of disks or platters are grouped together and are mounted on a common drive to form a disk pack. A term cylinder is usually used in case of a disk pack. A disk pack generally contains 6 platters. One platter has two recording surfaces one above and the other below it. No data is recorded on the topmost and the bottommost surfaces.

Hard disks are quite sensitive devices. The read/write head does not actually touch the disk but rather rides on a cushion of air about 0.000001 inch thick. The disk is sealed from impurities within a container, and the whole apparatus is manufactured under sterile conditions. Otherwise, all it would take is a human hair a finger print smudge, a dust particle, or a smoke particle to cause what is called head crash. A **head crash happens when the surface of the read/write head or particles on its surface come into contact with the surface of the hard-disk platter, causing the loss of some or all the data on the disk.** A head crash can also happen when you bump a computer too hard or drop something heavy on the system cabinet. So, always take up backup of data.

Each surface has concentric circles dividing the disk into tracks. Disk drives have read/write heads for writing to and reading from the disks. Some disk drives have fixed read/write heads. In this case, each track has a read/write head associated with it and therefore, the only delay in accessing a specific record is the rotational delay.

Each track is divided into a number of fixed length physical blocks called sectors. These are like blocks on magnetic tape. Sector is the smallest unit of data for transfer. The sectors are separated by inter record gaps.

Bits of character are recorded serially in each sector. Disks are available in different sizes and with different speeds. Different type of disks have different number of tracks and sectors. Number of tracks is generally 800 and number of sectors per track is 64.

Movable head disk drives have single read/write head per recording surface. The arm having read/write head moves on the recording surface so that it can be positioned on any track for reading or writing. All the read/write heads move together and therefore, all the read/write heads are positioned on the tracks on surfaces in the same plane. The same number of the track on each of the surfaces together are said to form a **cylinder.**

NOTES

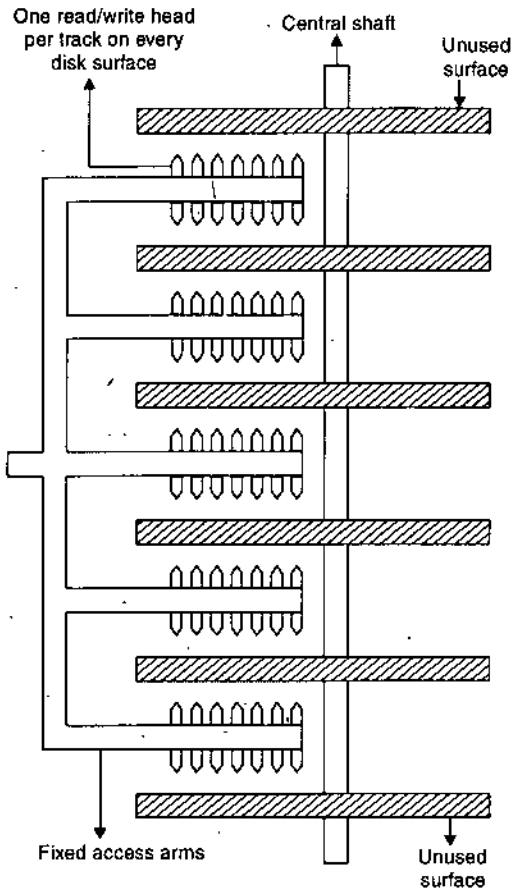


Fig. 33. *A disk drive having fixed head.*

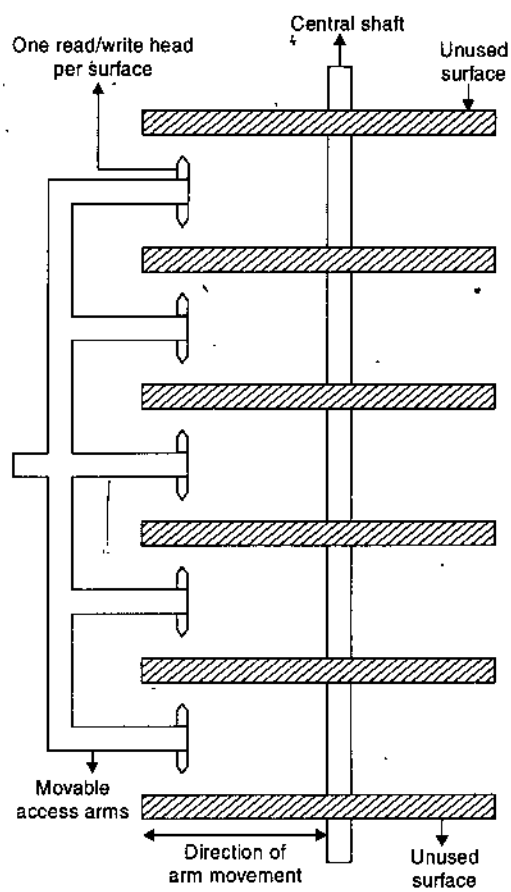


Fig. 34. *A movable head disk.*

Microcomputer hard drives with capacities measured in tens of gigabytes—upto 40 gigabytes, according to current ads—are becoming essential because today's programs are so huge. Microsoft Office alone is 500 megabytes. These allow faster access to data than floppy disks do, because a hard disk spins

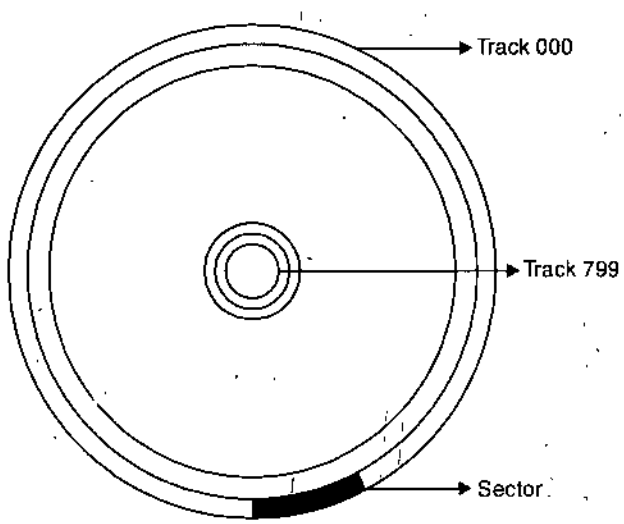


Fig. 35

NOTES

many times faster. Computer ads frequently specify speeds in revolutions per minute. A floppy disk drive rotates at only 360 rpm; a 7200-rpm hard drive is going about 300 miles per hour.

Access Time : The access time of a record on a disk consists of 3 factors viz. Seek Time, Latency Time and Data Transfer Time.

Seek Time : It refers to the time taken to position the read/write head at the desired track on the disk.

Latency Time : It refers to the time taken to position the read/write head at the desired sector of the track. Latency time depends on the speed of rotation of the disk.

Data Transfer Time : It is the actual time required to transfer the data. The data transfer time depends upon density of stored data and rotational speed of the disk.

Differences between a Floppy Disk and a Hard Disk

The differences between a floppy disk and a hard disk are given below :

Floppy disk	*Hard disk*
Floppy disks are also known as floppies or microdisks.	Hard disks are also known as fixed disks.
The computer takes more time to read from a floppy disk.	The computer takes less time to read from hard disk.
More prone to damage by heat, dust and improper handling as it is made of a flexible material.	Less prone to damage as it is within the system unit.
Can be used to store 1.44 MB of data.	Can be used to store far more data than floppy disks. They can be used to store data in the range a few GBs.
It is cheap.	It is costly.

2.6.3 For Magnetic Disk Numerical Problems

Storage capacity of one surface = No. of tracks × no. of sectors × bytes per sector

Storage capacity of the disk pace = Storage capacity of one surface × no. of surfaces

Number of cylinders = No. of tracks per surface

Transfer rate = No. of bytes per track × rotational speed

Example. *A 6 disk pack has 600 tracks per surface. There are 10 sectors per track and 512 bytes per sector.*

(i) What is the storage capacity of the disk pack ?

(ii) How many cylinders does the disk pack have ?

(iii) How many tracks are there per cylinder ?

Solution :

(*i*) Storage capacity of one surface

= No. of tracks × No. of sectors × bytes stored per sector

= 600 × 10 × 512 = 3072000 bytes

Storage capacity of the disk pack

= Storage capacity of one surface × no. of surfaces

= (3072000 × 10) bytes

(∵ upper and lower surfaces are not used)

= (30720000) bytes

$$= \frac{30720000}{1024 \times 1024} \text{ Mb}$$

= 29.29 Mb ≃ 30 Mb

(*ii*) No. of cylinders = No. of tracks on each disk

= 600

(*iii*) No. of tracks per cylinder = No. of usable surfaces on the disk = 10

NOTES

Compact Disk (Optical Disk)

Everyone who has ever played an audio CD is familiar with optical disks. *An optical disk is a removable disk, usually 4.75 inches in diameter and less than one-twentieth of an inch thick, on which data is written and read through the use of laser beams.* An audio CD holds upto 74 minutes of high-fidelity stereo sound. Some optical disks are used strictly for digital data storage, but many are used to distribute multimedia programs that combine text, visuals and sound.

Optical disk storage system consists of a rotating disk which is coated with a thin metal or other material that is highly reflective. Optical storage techniques make use of pinpoint precision which is possible with laser beams. A laser uses a concentrated and narrow beam of light. Hence, a laser beam is used to write information to or read information from an optical disk.

Optical storage devices focus a laser beam on the recording medium, which is a spinning disk. Some areas of the disk reflect the laser light into sensor, whereas others scatter the light. As the disk rotates past the laser and the sensor, a spot that reflects the laser beam into the sensor is interpreted as binary one, and the absence of reflection is interpreted as binary zero.

The storage density of optical disks is enormous, the storage cost is extremely low and the access time is relatively fast. An optical disk can hold over 4.7 gigabytes of data, the equivalent of 1 million typewritten pages. A typical shortcoming of optical storage devices is that they are permanent storage devices. Data once recorded cannot be erased and hence the disk cannot be reused. Extensive research is being carried out to develop erasable optical disks. The types of optical disks are :

NOTES

(*i*) CD-ROM — Compact Disk Read Only Memory

(*ii*) CD-R (Compact Disk Recordable)

(*iii*) CD-RW (Compact Disk Rewritable)

(*iv*) DVD-ROM (Digital Versatile or Digital Video Disk, with Read Only Memory)

(*i*) **CD-ROM.** For PC users, the best known type of optical disk is the CD-ROM. *CD-ROM (Compact Disk Read Only Memory) is an optical-disk format that is used to hold pre-recorded text, graphics and sound.* The disk's content is recorded at the time of manufacture and cannot be written on or erased by the user. CD-ROM uses the same technology that is used in music CDs. The disk is made of a resin, such as polycarbonate and is coated with a material that is highly reflective, usually aluminium. Data is recorded by focussing a laser beam on the surface of the disk. The laser beam is turned on and off at a varying rate because of which tiny holes (or pits) are produced on the metal coating. In order to read the stored data, a less powerful laser beam is focussed on the disk surface. This beam is strongly reflected by the coated surface and weakly reflected by the pits, thereby producing patterns of on-off reflections that can be converted into electronic signals.

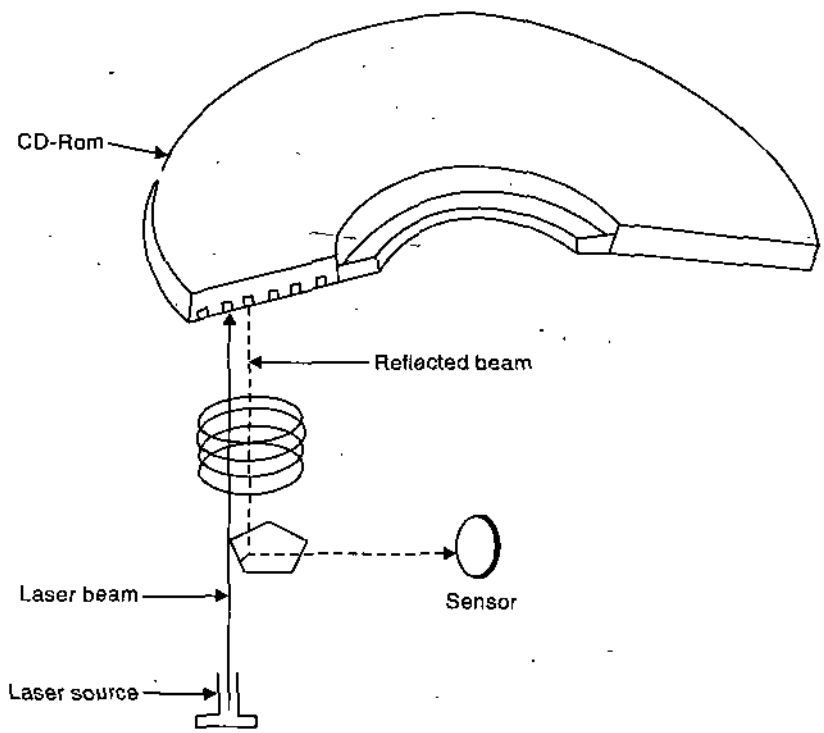


Fig. 36. *CD-ROM.*

CD-ROMs do not use concentric circles. Rather they use simple spiral tracks. The capacity of CD-ROM is 650 MB, equal to over 300,000 pages of text.

NOTES

(*ii*) **CD-R (Compact Disk Recordable).** *CD-R can be written to only once but can be read many times.* This allows users to make their own CD disks, though it is a slow process. (Recording a complete disk takes 20–60 minutes). The information recorded once cannot be erased. CD-R disks are most commonly used for archival applications. Its advantage is high capacity, better reliability and longer life.

(*iii*) **CD-RW (Compact Disk Rewritable).** *A CD-RW disk, also known as an erasable optical disk, allows users to record and erase data, so that the disk can be used over and over again.* Special CD-RW drives and software are needed. CD-RW disks are useful for archiving and backing up large amount of data or work in multimedia production or desktop publishing. CD-RW disks cannot be read by CD-ROM drives.

(*iv*) **DVD-ROM.** *A DVD-ROM* **(Digital Versatile Disk or Digital Video Disk, with Read Only Memory)** *is a CD-style disk with extremely high capacity, able to store 4.7–17 gigabytes.* The surface of a DVD contains microscopic pits, which represent the 0s and 1s of digital code than can be read by a laser.

The DVD drives can also take standard CD-ROM disks, so a user can watch DVD movies and play CD-ROMs using these. DVDs have enormous potential to replace CDs for archival storage, mass distribution of software and entertainment. DVDs not only store far more data but are different in quality from CDs. The variants of DVDs are :

DVD-R	(DVD–Recordable)—Permits one-time recording by the user.
DVD-RW	(DVD–Rewritable)
DVD-RAM	(DVD–Random Access Memory)
DVD + RW	(DVD + Rewritable)

All of these three types are reusable, that is these can be recorded on and erased many times.

Differences between a Hard Disk and a CD-Rom

The differences between a hard disk and a CD-ROM are given below :

Hard disk	*CD-ROM*
Hard disks are also known as fixed disks.	CD-ROMs are also known as optical disks.
Data is stored in the form of concentric circles.	Data is stored in the form of a single spiral track.
The computer takes less time to read from hard disk. It is in the range of 10 to 30 milliseconds.	The computer takes more time to read from CD-ROM. It is in the range of 100 to 300 milliseconds.
Data can be read or written as and when required. These can be reused.	It is a permanent storage medium. Data once recorded, cannot be erased and hence, the CD-ROMs cannot be reused.

NOTES

Hard disks are not portable.	CD-ROMs are portable.
Hard disks require a less complicated drive mechanism.	CD-ROMs require a more complicated drive mechanism.
Hard disks have a very large storage capacity (Disk packs have virtually unlimited storage capacity).	It has a storage capacity of about 650 Megabytes.
It is costly.	It is cheap.
Not a better storage medium for data archiving as compared to CD-ROMs.	CD-ROMs have a data storage life in excess of 30 years. These are a better storage medium for data archiving as compared to Hard disks.

Magnetic Tape

Magnetic tape is a secondary storage device which can hold large volumes of data on it. The tape is a sequential access media and data on it can be accessed sequentially. Large files are stored on them. It is one of the most popular storage media because of lower cost.

The Magnetic tape is made up of thin plastic ribbon coated on one side with ferromagnetic material. It is ½ inch in width and over 2500 ft. long. The coated side of the tape is usually divided into nine horizontal rows called tracks. Along the width of the tracks every character is stored with one bit in each track. The eight bits of EBCDIC or ASCII code of the character occupy 8 tracks while the ninth track is used as parity bit.

The density of recording on a tape is the number of characters per inch on the tape. Typical tape densities are 800 bpi (bytes per inch), 1600 bpi and 6250 bpi. Records are written one after the other on the tape. The processing program reads one record at a time into the primary memory for the data to be processed. During the time this record is being processed the tape drive stops. It will be moving the tape under the magnetic read/write head when

Track 1	1
Track 2	1
Track 3	0
Track 4	0
Track 5	1
Track 6	0
Track 7	1
Track 8	0
Track 9 (Parity Bit)	0

Fig. 37. *Structure of a nine track tape.*

NOTES

the next record is to be read. So between reading two consecutive records the tape drive stops and starts again. During the time that the drive deaccelerates to stop and accelerates again to a constant speed at which data from the tape is read a certain length of tape is empty. This empty portion of tape which occurs between every two records is called **Inter Record Gap** (IRG).

The IRGs are a wastage of storage space of the tape and cause slowing down of the program execution. Hence, a number of records are grouped together into a block and one block is transferred at a time into the buffer area in the computers primary memory. The individual records are read one by one by the program from this buffer memory for processing. The tape drive has to stop and start between reading of blocks. Once again gaps called **Inter Block Gaps** (IBGs) are generated between blocks of data. Since the IBG occur less frequently than the IRGs the empty space in the tape is reduced.

The initial and final few ft. of the magnetic tape are used for winding on a reel and hence can't be used. The beginning and end of the usable part of the tape are marked by markers called "Load Point" and "End of Reel" markers. The first record after load point is known as the Header Control Label. This record gives information about the contents of the tape. Several sequential files may be stored in the tape. Each file has its own header and trailer markers to identify the beginning and end of each file.

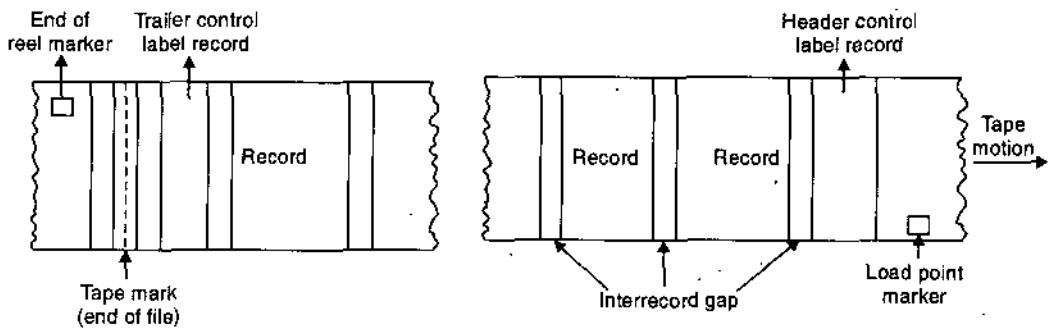


Fig. 38. *Magnetic tape markers and labels.*

The rate at which data is transferred between the tape and the CPU depends on the tape drive speed and the density of recording. The actual rate will, however, be lower than the value due to presence of IBGs and the start/stop time lost.

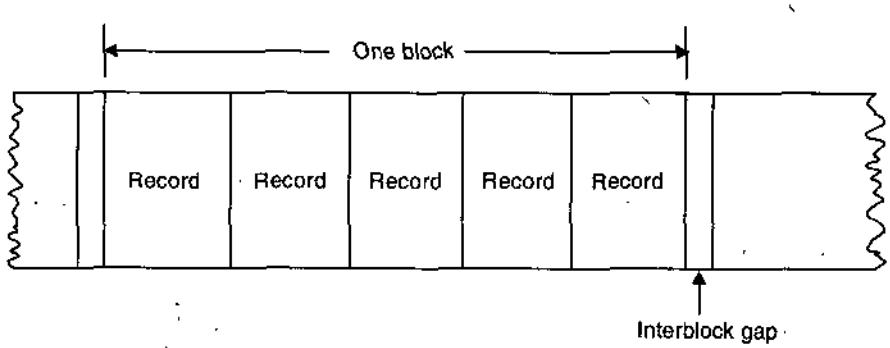


Fig. 39. *Multiple record block.*

NOTES

For Magnetic Tape Numericals :

$$\text{Length of a block} = \frac{\text{Block size}}{\text{Tape density}}$$

$$\text{Maximum no. of blocks on a tape} = \frac{\text{Tape length}}{\text{Block length} + \text{IBG}}$$

where IBG means Inter Block Gap.

$$\text{Number of records on tape} = \text{No. of blocks} \times \text{Blocking factor}$$

$$\text{where Blocking factor} = \text{No. of records in a block}$$

$$\text{Transfer rate} = \text{Tape density} \times \text{Tape speed}$$

$$\text{Effective transfer rate} = \frac{\text{Transfer rate} \times \text{length of a block}}{\text{Length of a block with IBG}}$$

Example. *Determine the amount of data which can be stored on a nine track tape, 4800 feet in length, having tape density 800 bytes per inch, IBG 0.5 inch and block size = 2000 bytes.*

Solution : Total length of tape = 4800 × 12 = 57,600 inches (1 ft = 12 inch)

$$\text{Length of a block} = \frac{\text{Block size}}{\text{Tape density}} = \frac{2000}{800} = 2.5 \text{ inch}$$

Actual length of a block (Block size + IBG) = 2.5 + 0.5 = 3 inch

$$\text{No. of blocks on the tape} = \frac{\text{Tape length}}{\text{Block length} + \text{IBG}}$$

$$= \frac{4800 \times 12}{3} = 19{,}200$$

$$\text{Amount of data on tape} = \text{No. of blocks} \times \text{block size}$$

$$= 19{,}200 \times 2{,}000$$

$$= 3{,}84{,}00{,}000 \text{ bytes}$$

Concepts of Virtual and Cache Memory

Pronounced "cash", *cache temporarility stores instructions and data that the processor is likely to use frequently. Thus, cache speeds up processing.* Cache memory uses special chips, often SRAM (static RAM) chips. On some systems, these chips are four times as fast as regular memory. However, the chips cost six times as much. It's this cost that keeps them from being used for the entire system's memory.

The technique used to access cache memory is very different from that of accessing the main memory. When the CPU accesses main memory, it outputs the data contained at the specified address. On the other hand the cache memory first compares the incoming address to the address stored in the cache. If the address matches, it is said that a **'hit'** has occurred. Then the corresponding data are read. If the address does not match it is said that a **'miss'** has occurred. In this case data is read from the main memory. The

NOTES

data read from main memory is also provided to cache memory so that when this specific address is accessed next time, a **hit** may occur.

Caches are sometimes described by their logical and electrical proximity to the microprocessor's core logic. The closed physically and electrically to the microprocessor's core logic is the *primary cache*, also called a *Level One Cache or L1.* A *secondary cache* (or *Level Two Cache i.e., L2)* lies between the primary cache and the main memory. The secondary cache or L2 is generally larger than the primary cache or L1 but it operates at a lower speed (to make its larger mass of memory more affordable).

The currently designed microprocessors have both the primary and secondary caches as part of the microprocessor itself. Earlier designs have the secondary cache in a separate part of a microprocessor module or in external memory.

L1 and L2 caches differ in the way they connect with the core logic of the microprocessor. L1 invariably operates at the full speed of the microprocessor's core logic with the widest possible bit-width connection between the core logic and the cache. L2 often operate at a rate slower than the chip's core logic, although all current chips operate the secondary cache at full core speed.

A major factor that determines how successful the cache will be is how much information it stores. The larger the cache, the more data that is in it and the more likely any required byte will be available there when your system needs it. Obviously, the best cache is one that's as large as, and duplicates, the entirety of system memory. Chip-makers try to design caches as large as possible within the constraints of fabricating microprocessors affordably.

These days primary caches (L1) are of size 64 or 128 KB. Secondary caches (L2) range from 128 to 512 KB for chips, for desktop and mobile applications and upto 2 MB for server-oriented microprocessors.

In addition, most current computer operating systems allow for the use of *virtual memory—that is, some free hard-disk space is used to extend the capacity of RAM.*

Virtual memory reduces the overall cost of the system because it's cheaper to store data on a hard disk drive than it is to add additional memory chips to the computer.

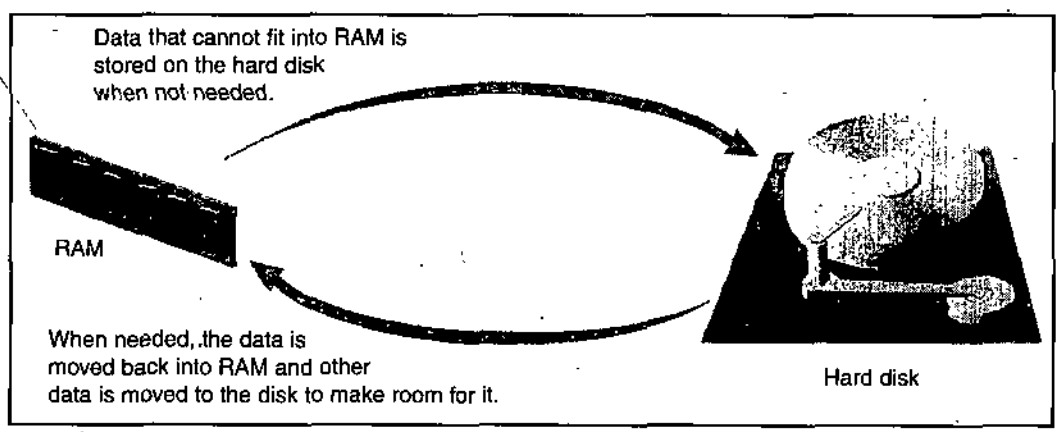


Fig. 40. *Virtual memory.*

NOTES

The processor searches for data or program instructions in the following order : first L1, then L2, then RAM, then hard disk (or CD-ROM). In this order, each kind of memory or storage is slower than its predecessor.

Units of Memory

The *binary number system* has only two digits : 0 and 1. Thus, in the computer, the 0 can be represented by the electrical current being off and the 1 by the current being on. All data and program instructions that go into the computer are stored in terms of these binary numbers.

Memory Capacity is denoted by *bits* and *bytes* and multiples thereof :

- **Bit (Binary Digit).** In the binary number system, *each 0 or 1 is called a bit, which is short for "binary digit."*
- **Byte.** To represent letters, numbers, or special characters (such as ? or *), bits are combined into groups. *A group of 8 bits is called a byte, and a byte represents one character, digit, or other value.* The capacity of a computer's memory or of a floppy disk is denoted in numbers of bytes or multiples such as kilobytes and megabytes.
- **Kilobyte.** A *kilobyte (K, KB) is about 1000 bytes.* (Actually, it's precisely 1024 bytes, but the figure is commonly rounded.) The kilobyte was a common unit of measure for memory or secondary-storage capacity on older computers. 1 KB equals about 1/2 page of text.
- **Megabyte.** *A megabyte (M, MB) is about 1 million bytes* (1,048,576 bytes). Measures of microcomputer primary-storage capacity today are expressed in megabytes. 1 MB equals about 500 pages of text.
- **Gigabyte.** A *gigabyte (G, GB) is about 1 billion bytes* (1,073,741,824 bytes). This measure was formerly used mainly with mainframe computers, but is typical of the secondary storage (hard disk) capacity of today's microcomputers (PCs). 1 GB equals about 500,000 pages of text.
- **Terabyte.** A *terabyte (T, TB) represents about 1 trillion bytes* (1,009,511,627,776 bytes). 1 TB equals about 500,000,000 pages of text.
- **Petabyte.** A *petabyte (P, PB) represents about 1 quadrillion bytes* (1,048,576 gigabytes).

2.7 SUMMARY

- A computer is a fast electronic device that processes the input data and provides the information as output.
- A computer is more accurate, faster, diligent and has much more memory than human beings.
- The Input/Output devices are also known as peripheral devices because they surround the CPU.

NOTES

- Keyboard and mouse are the main input devices of computer.
- Monitor and printer are the main output devices of computer.
- Printers are broadly classified into two types—Impact and Non-impact printers.
- Memory is an integral and an important component of a digital computer.
- **RAM** (Random Access Memory) is used as read/write memory of computer. It is volatile in nature.
- **ROM** (Read Only Memory) is one in which information is permanently stored, that is it is non-volatile memory.
- A **floppy disk,** often called a diskette or simply a disk, is a removable flat piece of mylar plastic packaged in a 3.5 inch plastic case.
- A hard disk can't be inserted or removed from the hard disk drive.
- An optical disk is a removable disk, usually 4.75 inches in diameter and less than one-twentieth of an inch thick, on which data is written and read through the use of laser-beams.
- Magnetic tape is a secondary storage device which can hold large volume of data on it. It is a sequential acces media.
- Cache temporarily stores instructions and data that the processor is likely to use frequently. Thus, cache speeds up processing.

2.8 TEST YOURSELF

1. Why are Input/output devices necessary for a computer system ?
2. What is an input device ? Name some of the commonly used input devices.
3. Write short notes on the following :
   (*a*) Printers
   (*b*) Video Standards
   (*c*) LED
4. What is the difference between impact and nonimpact printers ?
5. What are the differences between a hard disk and a CD-ROM ?
6. Briefly explain the features of hard disk with a neat diagram.
7. Write a short note on magnetic tape.

NOTES

SECTION C

CHAPTER 3 PROGRAMMING FUNDAMENTALS

★ LEARNING OBJECTIVES ★

3.1 INTRODUCTION

A **program** is a sequence of instructions written in a programming language. There are various programming languages, each having its own advantages for program development. Generally every program takes an input, manipulates it and provides an output as shown below :

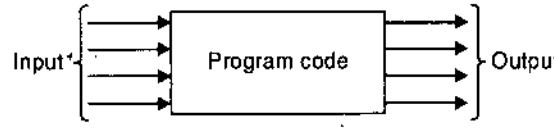


Fig. 1. *A conceptual view of a program.*

John Von Neumann proposed that if a program was stored in memory, program instructions could be easily changed just by loading a new program. Also as the program executed, it could easily change the instructions in memory. This is called the **stored program concept.**

NOTES

For better designing of a program, a systematic planning must be done. Planning makes a program more **efficient** and more **effective.** A programmer should use planning tools before coding a program. By doing so, all the instructions are properly interrelated in the program code and the logical errors are minimized.

There are various planning tools for mapping the program logic, such as **flowcharts, pseudocode, decision tables** and **hierarchy charts** etc. A program that does the desired work and achieves the goal is called an effective program whereas the program that does the work at a faster rate is called an efficient program.

The software designing includes mainly two things—*program structure* and *program representation.* The program structure means how a program should be. The program structure is finalised using top-down approach or any other popular approach. The program structure is obtained by joining the subprograms. Each subprogram represents a logical subtask.

The program representation means its presentation style so that it is easily readable and presentable. A user friendly program (which is easy to understand) can be easily debugged and modified, if need arises. So the programming style should be easily understood by everyone to minimize the wastage of time, efforts and cost.

Change is a way of life, so is the case with software. The modification should be easily possible with minimum efforts to suit the current needs of the organization. This modification process is known as **program maintenance.**

Flowcharting technique is quite useful in describing program structure and explaining it. The other useful techniques for actually designing the programs are :

(*i*) Modular programming

(*ii*) Top-down design (Stepwise refinement)

(*iii*) Structured programming.

3.2 TECHNIQUES OF PROBLEM SOLVING

Computer problem-solving can be summed up in one word—*it is demanding !* It is a combination of many small parts put together in a complex way, and therefore difficult to understand. It requires much thought, careful planning, logical accuracy, continuous efforts, and attention to detail. Simultaneously it can be a challenging, exciting, and satisfying experience with a lot of room for personal creativity and expression. If computer problem-solving is approached in this spirit then the chances of success are very bright.

For solving a problem on a computer a set of explicit and unambiguous instructions is written in a programming language. This set of instructions is called a *program.* An algorithm (step by step procedure to solve a problem in unambiguous finite number of steps) written in a programming language is a program. So, an algorithm corresponds to a solution to a problem which is *independent* of any programming language.

NOTES

Problem solving is a creative process which largely defies systematization and mechanization. Everyone acquires some problem-solving skills during his/her student life which he/she may or may not be aware of.

Some steps for problem solving improve the performance of the problem solver. No universal methods are available for it. Different people use different strategies. In simple words we can say logically that computer problem solving is about understanding.

3.2.1 Understanding of the Problem

When lot of efforts are made in understanding the problem we are dealing with, chances of success are also bright. We cannot hope to make useful progress in solving a problem until it is clear, what it is we are trying to solve. The preliminary investigation may be thought of as the *problem definition phase.* The problem definition defines what the problem is without any reference to the possible solutions. It is a simple statement, may be one to two pages and should sound like a problem. The problem definition should be in user language and it should be described from the user's point of view. It usually should not be defined in technical computer terms. As the analyst assigns the programs to different programmers module-wise, the programmers understand the problem given to them. The programmers define the problem of each program on a document and proceed for the next step. In simple words, a lot of care should be taken in working out precisely what must be done.

The problem solver should obtain information on the following three aspects of the problem after the analyses :

1. Input specification
2. Output specification
3. Special processing, if any.

1. Input Specifications

The input specifications should give the following information :

(*i*) Specific data values to be used as input in the program.

(*ii*) Input data format *i.e.*, order, spacing, accuracy and units.

(*iii*) The valid range of input data.

(*iv*) Restrictions, if any, on use of these data values and what to do if an input data is not accepted by the computer, should it be ignored or modified.

(*v*) The indication of end of input data (if specified by a special symbol).

2. Output Specifications

The output is obtained on executing a program. The output specifications must clearly define the values required and their formats etc. The output specifications must include the following information :

(*i*) The output data values required.

(*ii*) Output data format *i.e.*, precision (number of significant digits), accuracy, units, the position on the output sheet and suitable headings for making the output readable.

(*iii*) Amount of output required because the program has to be coded according to the number of output data values required.

3. Special Processing, if any

It means processing of input data under some conditions. If conditions are violated, certainly results are going to be incorrect. The processing under special condition(s) and the recovery action should be handled carefully. If the special processing conditions are ignored and left in the problem definition phase, it may be a costly affair later on.

So, in the problem definition phase, detailed information about input, output and special processing is gathered. These conditions are taken into consideration while solving the problem. The method of solution is not specified in this phase.

Step by Step Solution for the Problem

There are many ways to solve most of the problems and also many solutions to most of the problems. This situation makes the job of problem-solving a difficult task. When we have many ways to solve a problem it is usually difficult to recognize quickly which paths are likely to be fruitless and which paths may be productive.

A block often occurs after the problem definition phase, because people become concerned with details of the implementation *before* they have completely understood or worked out an implementation-independent solution. The problem solver should not be too concerned about detail. That can be taken into account when the complexity of the problem as a whole has been brought under control. The old computer proverb states, **"the sooner you start coding your program the longer it is going to take".**

An approach that often allows us to make a start on a problem is to take a specific example of the general problem we wish to solve and try to work out the mechanism that will allow us to solve this particular problem (*e.g.*, if you want to find the top scorer in an examination, choose a particular set of marks and work out the mechanism for finding the highest marks in this set).

This approach of focusing on a particular problem can often give us a platform we need for making a start on the solution to the general problem. It is not always possible that the solution to a specific problem or a specific class of problems is also a solution to the general problem. We should specify our problem very carefully and try to establish whether or not the proposed algorithm (step by step procedure in a finite number of steps to solve a problem) can meet those requirements. If there are any similarities between the current problem and other problems that we have solved or we have seen solved, we should be aware of it. In trying to get a better solution to a problem, sometimes too much study of the existing solution or a similar problem forces us down the same reasoning path (which may not be the best) and to the same dead end. Therefore, a better and wiser way to get a better solution to a problem is, try to solve the problem *independently*.

Any problem we want to solve should be viewed from a variety of angles. When all aspects of the problem have been seen, one should start solving it. Sometimes, in some cases it is assumed that we have already solved the problem and then

NOTES

try to work backwards to the starting conditions. The most crucial thing of all in developing problem-solving skills is practice.

Probably the most widely known and most often used principle for problem-solving is the *divide-and-conquer* strategy. The given problem is divided into two or more subproblems which can hopefully be solved more efficiently by the same technique. If it is possible to continue in this way we will finally reach the stage where the subproblems are small enough to be solved without further splitting.

This way of breaking down the solution to a problem has been widely used with searching, selection and sorting algorithms.

3.3 FLOWCHARTING

The technique of drawing *flowcharts* is known as flowcharting. A flowchart is a pictorial representation of the sequence of operations necessary to solve a problem with a computer. The first formal flowchart is attributed to *John Von Neumann* in 1945. The flowcharts are read from left to right and top to bottom. Program flowcharts show the sequence of instructions in a program or a subroutine. The symbols used in constructing a flowchart are simple and easy to learn. These are very important planning and working tools in programming. The purposes of the flowcharts are given below :

(*i*) **Provide Better Communication.** These are an excellent means of communication. The programmers, teachers, students, computer operators and users can quickly and clearly get ideas and descriptions of algorithms.

(*ii*) **Provide an Overview.** A clear overview of the complete problem and its algorithm is provided by the flowchart. Main elements and their relationships can be easily seen without leaving important details.

(*iii*) **Help in Algorithm Design.** The program flow can be shown easily with the help of a flowchart. A flowchart can be easily drawn in comparison to write a program and test it. Different algorithms (for the same problem) can be easily experimented with flowcharts.

(*iv*) **Check the Program Logic.** All the major portions of a program are shown by the flowchart, precisely. So the accuracy in logic flow is maintained.

(*v*) **Help in Coding.** A program can be easily coded in a programming language with the help of a flowchart. All the steps are coded without leaving any part so that no error lies in the code.

(*vi*) **Modification Becomes Easy.** A flowchart helps in modification of an already existing program without disrupting the program flow.

(*vii*) **Better Documentation Provided.** A flowchart gives a permanent storage of program logic pictorially. It documents all the steps carried out in an algorithm. A comprehensive, carefully drawn flowchart is always an indispensable part for the program's documentation.

NOTES

Flowchart Symbols

Flowcharts have only a few symbols of different sizes and shapes for showing necessary operations. Each symbol has specific meaning and function in a flowchart. These symbols have been standardized by the *American National Standards Institute (ANSI)*. The basic rules that a user must keep in mind while using the symbols are :

1. Use the symbols for their specific purposes.
2. Be consistent in the use of symbols.
3. Be clear in drawing the flowchart and the entries in the symbols.
4. Use the annotation symbol when beginning a procedure.
5. Enter and exit the symbols in the same way.

The flowchart symbols alongwith their purposes are given below :

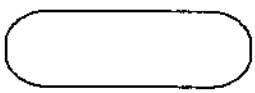
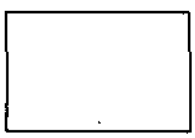
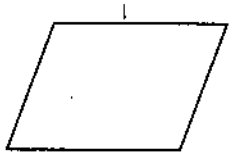
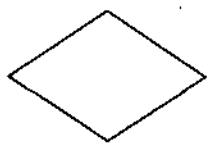
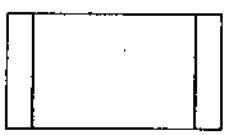
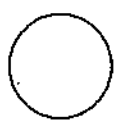
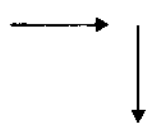
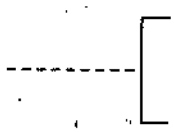
Symbol	Name	Purpose
	Terminal	Indicates the beginning and end of a program.
	Process	For calculation or assigning of a value to a variable.
	Input/Output (I/O)	Any statement that causes data to be input to a program (INPUT, READ) or output from the program, such as printing on the display screen or printer.
	Decision	Program decisions. Allows alternate courses of action based on a condition. A decision indicates a question that can be answered *yes* or *no* (or *true* or *false*).
	Predefined Process	A group of statements that together accomplish one task. Used extensively when programs are broken into modules.
	Connector	Can be used to eliminate lengthy flowlines. Its use indicates that one symbol is connected to another.
	Flowlines and Arrowheads	Used to connect symbols and indicate the sequence of operations. The flow is assumed to go from top to bottom and from left to right. Arrowheads are only required when the flow violates the standard direction.
	Annotation	Can be used to give explanatory comments.

Fig. 2. *Explanation of flowchart symbols.*

NOTES

3.3.1 Control Structures

A *control structure*, or logic structure, is a structure that controls the logical sequence in which computer program instructions are executed. In structured program design, three control structures are used to form the logic of a program : sequence, selection and iteration (or loop). These are also used for drawing flowcharts.

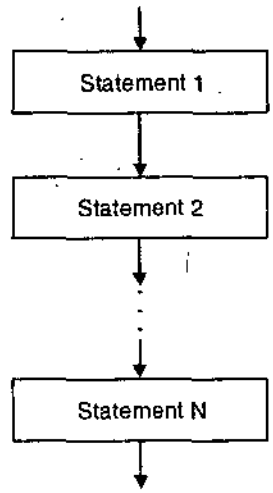


Fig. 3. *Sequence control structure.*

Let us consider the three control structures :

1. In the ***sequence control structure*, one program statement follows another in logical order.** There are no decisions to make, no choices between "yes" or "no". The bones logically follow one another in sequential order. For example, figure 3 illustrates a sequence of N statements :
2. The ***selection control structure*, also known as an IF-THEN-ELSE structure—represents a choice. It offers two paths to follow when a decision must be made by a program.** An example of a selection structure is as follows :

IF a student's marks in a subject is ⩾ 60 **THEN**

Student has secured first division

ELSE

Student has not secured first division

NOTES

Figure 4 illustrates the selection control structure.

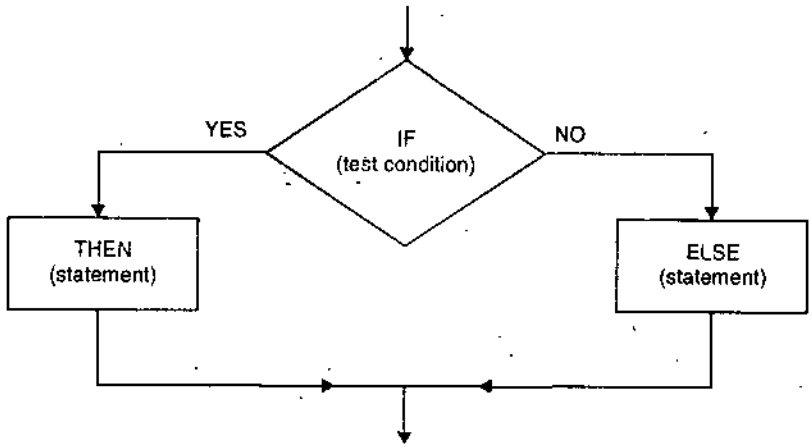


Fig. 4. *Selection control structure (IF-THEN-ELSE).*

A variation on the usual selection control structure is the *case control structure.* This offers more than a single yes-or-no decision. The case structure allows several alternatives, or "cases", to be presented. "IF Case 1 occurs, THEN do thus-and-so. IF Case 2 occurs, THEN follow an alternative course...." And so on). The case control structure saves the programmer the trouble of having to indicate a lot of separate IF-THEN-ELSE conditions. Figure 5 illustrates this :

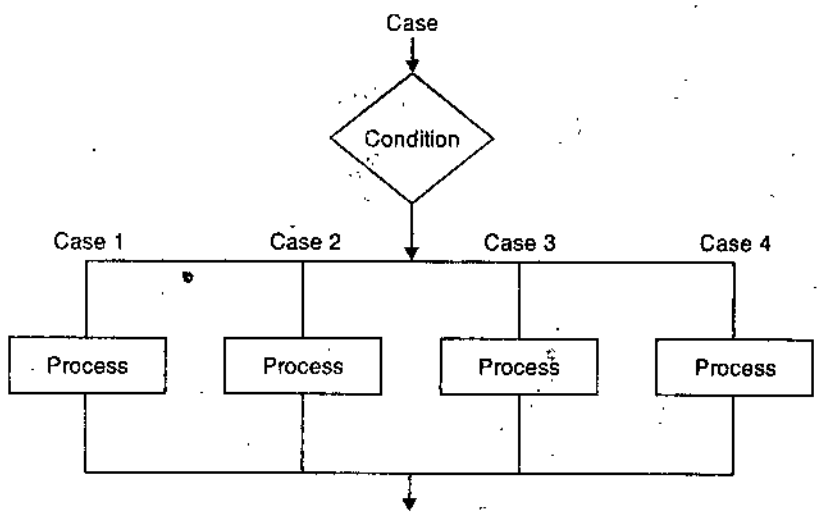


Fig. 5. *Variation on selection : the case control structure.*

3. In the ***iteration,*** or ***loop, control structure,*** **a process may be repeated as long as a certain condition remains true.** There are two types of iteration structures—*REPEAT-UNTIL* and *WHILE-DO.* Of these, REPEAT-UNTIL is more often encountered.

   An example of a REPEAT-UNTIL structure is as follows :

   REPEAT read in student records UNTIL there are no more student records.

NOTES

An example of a WHILE-DO structure is as follows :

WHILE read in student records DO—that is, as long as—there continue to be student records.

The difference between the two iteration structures is : If several statements are to be repeated, we must decide when to *stop* repeating them. **WHILE-DO** structure can be used to stop them at the *beginning* of the loop. Or we can decide to stop them at the *end* of the loop using **REPEAT-UNTIL** structure. REPEAT-UNTIL iteration means that the loop statements will be executed at least once, because the condition is tested in the end of loop.

One thing that all three control structures have in common is *one entry* and *one exit.* The control structure is entered at a single point and excited at another single point. This helps simplify the logic so that it is easier for others following in a programmer's footsteps to make sense of the program.

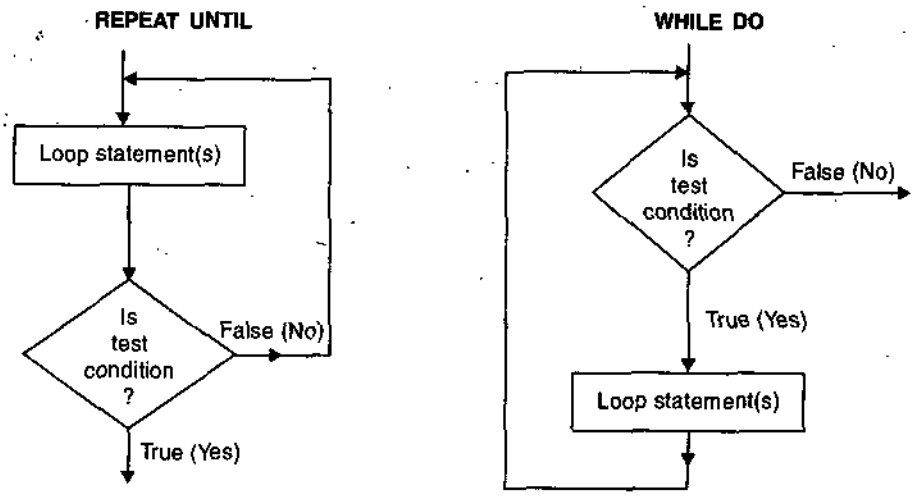


Fig. 6. *Iteration control structures (Loops).*

3.3.2 Types of Flowcharts

The systems designer and programmer use the following types of flowcharts in developing algorithms :

1. System flowcharts
2. Modular program flowcharts
3. Detail program flowcharts or application flowcharts

1. System Flowcharts

It plays a vital role in the system analysis. A system is a group of interrelated components tied together according to a plan to achieve a predefined objective. The elements and characteristics of a system are graphically shown and its structure and relationship are also represented by flowchart symbols. The system analysts use the system flowcharts for analysing or designing various systems. The different stages of a system are :

(*i*) Problem recognition

(*ii*) Feasibility

(*iii*) System analysis

(*iv*) System design

(*v*) Implementation

(*vi*) Evaluation

All the above stages use the system flowchart for convenience. Any alternative solution for the existing system or the entirely new system can be systematically represented. The working system is well documented by a precisely drawn system flowchart.

2. *Modular Program Flowcharts*

A system flowchart indicates the hardware, identifies the various files and represents the general data flow. A modular program flowchart on the other hand defines the logical steps for the input, output and processing of the information of a specific program. In structured or modular programs, the independent modules or units are written for different procedures. This module is useful in performing the specified operation in other programs too. The exact operation in detail is not performed but only the relationship and order in which processes are to be performed are included.

It is also called as block diagram. Its main advantage lies in the fact that the programmer can concentrate more on flow of logic and temporarily computer level details are ignored. Alternate algorithms without much time consumption or effort can also be tried using it. These help a lot in communicating the main logic of the program.

3. *Detail Program Flowcharts*

These are the most comprehensive and elemental charts in developing the programs. The symbols represented by it are quite useful for coding the program in any computer language. Each computer language has its own syntax, so there may be some difference in performing the operations to be followed for coding a program. A detail program flowchart represents each minute operation in its proper sequence, reduced to its simplest parts.

3.3.3 Rules for Drawing Flowcharts

The following rules and guidelines are recommended by ANSI for flow-charting :

1. First consider the main logic, then incorporate the details.
2. Maintain a consistent level of detail for a flowchart.
3. Do not include all details in a flowchart.
4. Use meaningful descriptions in the flowchart symbols. These should be easy to understand.

NOTES

5. Be consistent in using variables and names in the flowchart.
6. The flow of the flowchart should be from top to bottom and from left to right.
7. For a complex flowchart, use connectors to reduce the number of flow lines. The crossing of lines should be avoided as far as possible.
8. If a flowchart is not drawn on a single page, it is recommended to break it at an input or output point and properly labelled connectors should be used for linking the portions of the flowchart on separate pages.
9. Avoid duplication so far as possible.

Levels of Flowcharts

There are two levels of flowcharts :

(*i*) Macro flowchart

(*ii*) Micro flowchart

(*i*) **Macro Flowchart.** It shows the main segment of a program and shows lesser details.

(*ii*) **Micro Flowchart.** It shows more details of any part of the flowchart.

Note : *A flowchart is independent of all computer languages.*

3.3.4 Limitations of Flowcharting

Flowcharts have some limitations also. These are given below :

(*i*) **Time Consuming.** These take a lot of time and are laborious to draw with proper symbols and spacing, especially for large complex programs.

(*ii*) **Difficult to Modify.** Any change or modification in the logic of the program generally requires a completely new flowchart for it. Redrawing a flowchart is tedious and many organizations either do not modify it or draw the flowchart using a computer program.

(*iii*) **No Standard Available.** There are no standards provided all over the world for the details to be included in drawing a flowchart. So different people draw flowchart with different views.

3.4 STRUCTURED PROGRAMMING CONCEPTS

The main objectives of structured programming are :

- Readability
- Clarity of programs
- Easy modification
- Reduced testing problems.

The **goto** statement should be avoided so far as possible. The three basic building blocks for writing structured programs are given below :

1. Sequence Structure
2. Loop or Iteration
3. Binary Decision Structure

1. Sequence Structure :

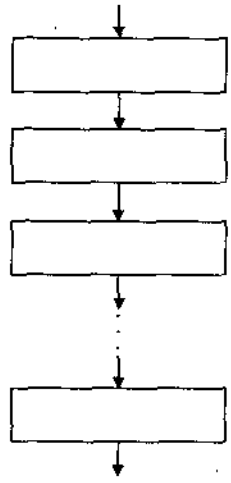


Fig. 7. *Sequence structure.*

It consists of a single statement or a sequence of statements with a single entry and single exit as show above.

2. Loop or Iteration :

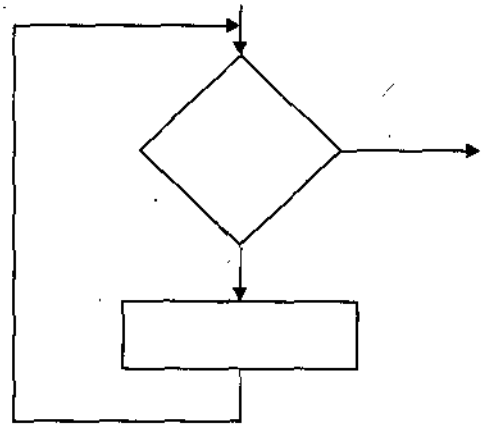


Fig. 8. *Loop or iteration.*

It consists of a condition (simple or compound) and a sequence structure which is executed condition based as shown above.

NOTES

3. Binary Decision Structure :

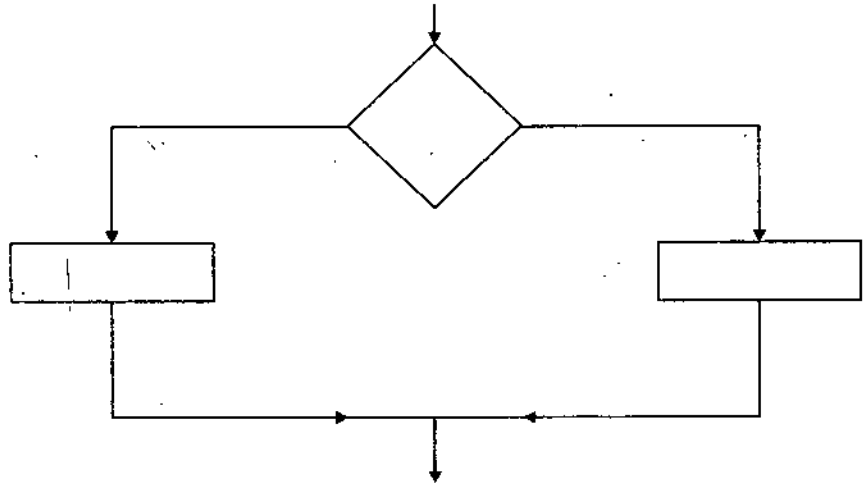


Fig. 9. *Binary decision structure.*

It consists of a condition (simple or compound) and two branches out of which one is to be followed depending on the condition being true or false as shown above.

3.5 MODULAR PROGRAMMING

Breaking down of a problem into smaller independent pieces (modules) helps us to focus on a particular module of the problem more easily without worrying about the entire problem. No processing outside the module should affect the

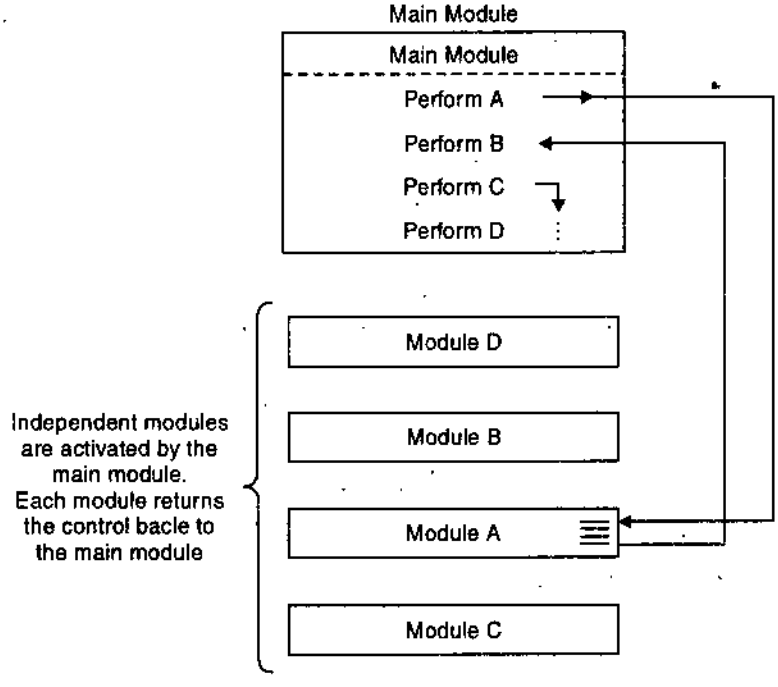


Fig. 10.

processing inside the module. It should have only one entry point and one exit point. We can easily modify a module without affecting the other modules. Using this approach the writing, debugging and testing of programs becomes easier than a monolithic program. A *modular* program is readable and easily modifiable. Once we have checked that all the modules are working properly, these are linked together by writing the main module. The main module activates the various modules in a predetermined order. For example, Figure 10 illustrates this concept :

It must be noted that each module can be further broken into other submodules.

3.5.1 Characteristics of Modular Approach

(*i*) The problem to be solved is broken down into major components, each of which is again broken down if required. So the process involves working from the most general, down to the most specific.

(*ii*) There is one entry and one exit point for each module.

(*iii*) In general each module should not be more than half a page long. If not so, it should be split into two or more submodules.

(*iv*) Two-way decision statement are based on IF..THEN, IF..THEN..ELSE, and nested IF structures.

(*v*) The loops are based on the consistent use of WHILE..DO and REPEAT..UNTIL loop structures.

3.5.2 Advantages of Modular Approach

(*i*) Some modules can be used in many different problems.

(*ii*) Modules being small units can be easily tested and debugged.

(*iii*) Program maintenance is easy as the malfunctioning module can be quickly identified and corrected.

(*iv*) The large project can be easily finished by dividing the modules to different programmers.

(*v*) The complex modules can be handled by experienced programmers and the simple modules by junior ones.

(*vi*) Each module can be tested independently.

(*vii*) The unfinished work of a programmer (due to some unavoidable circumstances) can be easily taken over by someone else.

(*viii*) A large problem can be easily monitored and controlled.

(*ix*) This approach is more reliable.

(*x*) Modules are quite helpful in clarification of the interfaces between major parts of the problem.

NOTES

3.6 ALGORITHM DESIGNING

Computers are basically used to solve complex problems in a systematic and easy manner. In order to solve a problem systematically, the solution should be written as a set of sequential steps. Each of these steps specify some simple actions that need to be performed. Thus *an algorithm may be defined as a finite and ordered sequence of steps which when performed lead to the solution of the problem in a definite time.* Ordered sequence implies that the execution takes place in the same manner or order in which the statements are written *i.e.*, each step of the algorithm is written in such a way that the next instruction follows automatically. The ordering is provided by assigning positive integers to the steps. The words BEGIN and END normally refer to the beginning and end of the algorithm. An algorithm must possess following characteristics :

1. **Finiteness.** Finiteness implies that the algorithm must have finite number of steps. Also the time taken to execute all the steps of the algorithm should be finite and within a reasonable limit.
2. **Definiteness.** By definiteness it is implied that each step of the algorithm must specify a definite action *i.e.,* the steps should not be vague. Moreover, the steps should be such that it is possible to execute these manually in a finite length of time.
3. **Input.** The term input means supplying initial data for an algorithm. This data must be present before any operations can be performed on it. Sometimes no data is needed because initial data may be generated within the algorithm. Thus the algorithm may have no or more inputs. Generally, the initial data, is supplied by a READ instruction or a variable can be given initial value using SET instruction.
4. **Output.** The term output refers to the results obtained when all the steps of the algorithm have been executed. An algorithm must have at least one output.
5. **Effectiveness.** Effectiveness implies that all the operations involved in an algorithm must be sufficiently basic in nature so that they can be carried out manually in finite interval of time.

3.6.1 Expressing Algorithms

The procedure for expressing algorithm is quite simple. The language used to write algorithms is similar to our day-to-day life language. In addition, some special symbols are also used which are described below :

(*i*) **Assignment Symbol (←).** The assignment symbol (←) is used to assign values to various variables. For example, let A be any variable and B be another variable or constant or an expression. Then the statement

$$A \leftarrow B$$

is called assignment statement. The statement implies that A is assigned the value stored in B. If A contains any previous value then that value is destroyed and the new value is assigned.

(*ii*) **Relational Symbols.** The commonly used relational symbols for algorithms are :

Symbol	*Meaning*	*Example*
<	Less than	A < B
<=	Less than or equal to	A <= B
=	Equal to	A = B
≠	Not equal to	A ≠ B
>	Greater than	A > B
>=	Greater than or equal to	A >= B

(*iii*) **Brackets ({}).** The pair of braces is used to write comments for the purpose of documentation. *For example,*

(*i*) BEGIN {Start of the algorithm}

(*ii*) Set N ← N + 1 {Increase the value of N by 1}

(*iii*) END {End of the algorithm}.

Basic Control Structures

The basic control structures needed for writing good and efficient algorithms are :

(*i*) Selection

(*ii*) Branching

(*iii*) Looping.

(*i*) **Selection.** The selection structure is used when we have to perform a given set of instructions if the given condition is TRUE and an alternative set of instructions if the condition is FALSE. The basic statement available for selection is IF-THEN-ELSE.

The syntax is

```
IF (condition is true) THEN
   Begin
      s1
      s2
      .
      .
      .
      sn
   End
```

NOTES

```
ELSE
    Begin
        f1
        f2
        .
        .
        .
        fn
    End
```

For example, consider the following algorithm which finds greater among 2 numbers.

```
        BEGIN
STEP 1  Read NUM1, NUM2
STEP 2  If NUM1 > NUM2
STEP 3  Then
        Write (NUM1, " is greater")
STEP 4  Else
        Write (NUM2, " is greater")
        END
```

(*ii*) **Branching.** The branching statement is required when we want to transfer the control of execution from one part or step of the algorithm to another part or step. The statement available for branching is GOTO and its syntax is

GOTO n

where n is a positive integer and specifies the step number where the control of execution is to be transferred.

(*iii*) **Looping.** The looping structure is used when a statement or a set of statements is to be executed a number of times. The following two loop control structures are commonly used in algorithms :

(*a*) WHILE-DO

(*b*) REPEAT-UNTIL

(*a*) **WHILE-DO :** The syntax is

```
STEP 1    WHILE (Condition) DO
STEP 2    S1
STEP 3    S2
          .
          .
          .
```

```
STEP N+1    SN
STEP N+2    END-WHILE {End of While-Do loop}
```

This control loop structure implies that as long as the condition remains true, all the steps listed between WHILE-DO and END-WHILE are executed again and again. As soon as the condition becomes false, the execution of the loop stops and control is transferred to next statement following END-WHILE.

For example, consider the following algorithm

```
            BEGIN
STEP 1      Set N ← 1
STEP 2      WHILE (N <= 10) DO
STEP 3      Write N
STEP 4      Set N ← N + 1
STEP 5      END-WHILE
            END
```

This algorithm initially sets the value of N to 1. The while statement then checks if the value of N <= 10. If the condition is true it executes steps 3 and 4. When the value of N exceeds 10 the condition becomes false and the control goes to the statement following END-WHILE which is END statement marking the END of algorithm.

(*b*) **REPEAT-UNTIL.** This is similar to WHILE-DO except the fact that the loop is executed till the condition remains false or condition becomes true. The syntax is :

```
STEP 1      REPEAT
STEP 2      S1
STEP 3      S2
            .
            .
            .
STEP N + 1  SN
STEP N + 2  UNTIL (Condition)
```

This control loop structure implies that as long as the condition remains false, all the steps listed between REPEAT and UNTIL are executed again and again. As soon as the condition becomes true, the execution of the loop stops and control is transferred to next statement following UNTIL (condition). *For example,* consider the following algorithm :

```
            BEGIN
STEP 1      Set N ← 1
STEP 2      REPEAT
STEP 3      Write N
```

NOTES

STEP 4 Set N ← N + 1 {Increment N by 1}
STEP 5 UNTIL (N > 10)
END

Initially, the value of N is set to 1. The loop executes till the value of N exceeds 10. After this, the control goes to the next statement following UNTIL (N > 10).

3.6.2 Advantages of Algorithms

(*i*) It is simple to understand step by step solution of the problem.

(*ii*) It is easy to debug *i.e.,* errors can be easily pointed out.

(*iii*) It is independent of programming languages.

(*iv*) It is compatible to computers in the sense that each step of an algorithm can be easily coded into its equivalent in high level language.

3.6.3 Algorithm : Linear Search

Given an array A of N elements. This algorithm searches for an element DATA in the array. I denotes the array index. This algorithm gives the first location where DATA is found.

1. I ← 1
2. While (I ⩽ N) Do upto step 3
3. **IF** (A[I] = DATA) **THEN**
   Begin
   Write "Successful search"
   Write DATA, " found at position ", I
   goto step 5
   End
   ELSE
   Begin
   I ← I + 1
   End
4. Write "Unsuccessful search"
5. END

3.6.4 Algorithm : Binary Search (Always Applicable on Sorted Data)

Given an array A of N elements in ascending order. This algorithm searches for an element DATA. LOW, HIGH, MID denote the lowest, highest and middle position of a search interval respectively.

1. LOW ← 1
   HIGH ← N

2. While (LOW ⩽ HIGH) Do upto step 4
3. MID ← Integral part of ((LOW + HIGH)/2)
4. **IF** (DATA = A[MID]) **THEN**

   Begin

   Write "Successful search"

   Write DATA, " found at position ", MID

   goto step 6

   End

   ELSE

   Begin

   IF (DATA > A[MID]) **THEN**

   LOW ← MID + 1

   ELSE

   HIGH ← MID – 1

   End
5. Write " Unsuccessful search "
6. END

3.6.5 Algorithm : Organize Numbers in Ascending Order (Sorting)

Given an array A of N elements. This algorithm arranges the elements in ascending order. I and J denote array indices. Variable TEMP is used for swapping.

1. Repeat for I = 1, 2, ..., N–1

   Begin

   Repeat for J = I + 1, I + 2, ..., N

   Begin

   IF (A[J] < A[I]) THEN

   Begin

   TEMP ← A[I]

   A[I] ← A[J]

   A[J] ← TEMP

   End

   End

   End
2. End

NOTES

3.6.6 Algorithm : Insertion in a Sorted Array

Given an array A of M elements (M < N) in ascending order. This algorithm inserts an element DATA. The array remains sorted after insertion. I and POS denot array indices. There are M + 1 elements after insertion of DATA.

1. **IF** (DATA ≥ A[M]) **THEN**
   Begin
   A[M + 1] ← DATA
   goto step 6
   End
2. I ← 1
3. Repeat While (*DATA ≤ A[I]*)
   I ← I + 1
4. Repeat for POS = M, M – 1, ..., I
   Begain
   A[POS + 1] ← A[POS]
   End
5. A[I] ← DATA
6. M ← M + 1
7. End

3.6.7 Algorithm : Merging of Ordered Lists

Give two array A and B of size M and N respectively having elements in ascending order. We want to merge these two arrays into array C of size M+N, also in ascending order. I, J and K denote array indices. Assuming the array indices begin at 1.

1. I = 1
   J = 1
   K = 1
2. Repeat While ((I ≤ M) AND (J ≤ N))
   Begin
   IF (A[I] ≤ B[J]) **THEN**
   Begin
   C[K] = A[I]
   I = I + 1
   End

NOTES

```
        ELSE
                Begin
                        C[K] = B[J]
                        J = J + 1
                End
        K = K + 1
    End
3. IF (I > M) THEN
    Begin
        Repeat while (J ≤ N)
            Begin
                C[K] = B[J]
                K = K + 1
                J = J + 1
            End
    End
  ELSE
    Begin
        Repeat While (I ≤ M)
            Begin
                C[K] = A[I]
                K = K + 1
                I = I + 1
            End
    End
4. End
```

3.7 TOP-DOWN PROGRAMMING (STEP WISE REFINEMENT)

Program development includes designing, coding, testing and verification of a program in any computer language. For writing a good program, the top-down design approach can be used. It is also called **systematic programming** or **hierarchical program design** or **stepwise refinement**. A complex problem is broken into smaller subproblems, further each subproblem is broken into a number of smaller subproblems and so on till the subproblems at the lowest

NOTES

level are easy to solve. Similarly a large program is broken into a number of subprograms and in turn each subprogram is further decomposed into subprograms ans so on. Suppose we want to solve a problem S, which can be decomposed into subproblems S1, S2 and S3 and so on. Let the program for S, S1, S2, S3 be denoted by P, P1, P2, P3 respectively. Further suppose that S2 is solved by decomposing it into subproblems S21 and S22 and program P21 and P22 are written for these. This operation of coding a subprogram in terms of lower level subprograms is known as the **process of stepwise refinement.** The following figure shows the hierarchical decomposition of P into its subprograms and sub-subprograms.

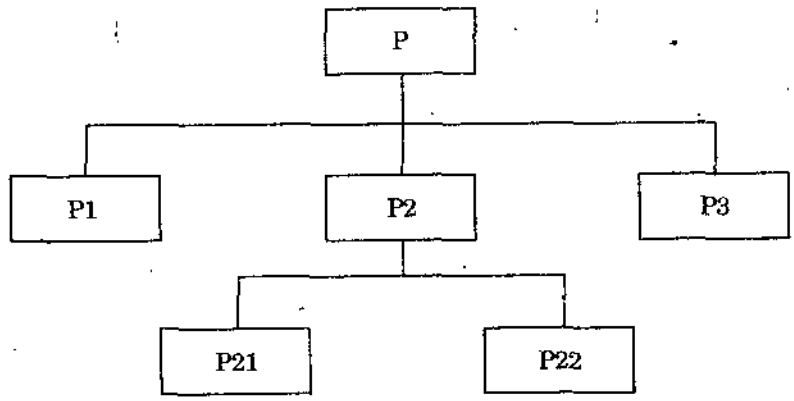


Fig. 11.

The advantages of the top-down design approach are :

1. A large problem is divided into a number of smaller problems using this approach. The decomposition is continued till the subproblems at the lowest level become easy to solve. So the overall problem solving becomes easy.
2. If we use the top-down approach for a problem then top-down programming method can be used for coding modules at various stages. So, the top level modules can be coded without coding the lower level modules earlier. This approach, is better than the bottom-up approach where programming starts first at the lowest level modules.
3. It helps in top-down testing and debugging of programs.
4. The programs become user friendly (that is easy to read and understand) and easy to maintain and modify.
5. Different programmers can write the modules for different levels.

NOTES

3.8 BOTTOM-UP PROGRAMMING

The bottom-up programming approach is the reverse of the top-down programming. The process starts with identification of a set of modules which are either available or to be constructed. An attempt is made to combine the lower level modules to form modules of a high level. This process of combining modules is continued until the program is realised. The main drawback of the bottom-up programming approach is the assumption that the lowest level modules can be completely specified beforehand, which in reality is seldom possible. Thus, in the bottom-up approach, quite often it is found that the final program obtained by combining the predetermined lowest level modules does not meet all the requirements of the desired program.

Here no attempt is made to compare the advantages and disadvantages of the top-down and bottom-up programming. However, program development through top-down approach is widely accepted to be better than the bottom-up approach.

3.9 SUMMARY

- A program is a sequence of instructions written in a programming language.
- John vohn Neumann proposed that the programs be stored in memory. This is called the stored program concept.
- Computers work because they are fed a series of step-by-step instructions called programs.
- A flowchart is a graphical way of illustrating the steps in a process. It uses symbols connected by flowlines to represent processes and the direction of flow within the program. A flowchart is independent of any programming language.
- Some useful techniques of problem solving other than flowcharting are modular programming, top-down design and structured programming.
- Modular programming is breaking down of a problem into smaller independent pieces (modules).
- An algorithm is a step-by-step procedure to solve a problem in unambiguous finite number of steps. An algorithm is independent of any programming language.
- Probably the most widely known and most often used principle for problem solving is the *divide-and-conquer* strategy. It is widely used with searching, selection and sorting algorithms.
- Complex decision logic associated with a problem represented in a tabular form is known as a decision table.

NOTES

- The main objectives of structured programming are readability, clarity of programs, easy modification and reduced testing problems.
- Top-down programming is also known as the *process of stepwise refinement.*
- Bottom-up programming approach is the reverse of the top-down programming.

3.10 TEST YOURSELF

1. Write a short note on the following :
   (*a*) Structured programming concepts
   (*b*) Modular programming
2. What is an algorithm ? Explain its need.
3. Draw a flowchart and write an algorithm for merging two sorted arrays given in descending order into ascending order.
4. Draw a flowchart for searching an element from a sorted array (ascending order) having N elements using binary search method.
5. Explain the concept of top-down and bottom-up programming.

SECTION D

CHAPTER 4 FUNDAMENTALS OF C

NOTES

★ LEARNING OBJECTIVES ★

NOTES

4.1 INTRODUCTION

"C" is the language's entire name, and it does not "stand" for anything. Developed at Bell Laboratories in the year 1972 by *Dennis Ritchie,* **C is a general-purpose, compiled language that works well for microcomputers and is portable among many computers.** It was originally developed for writing system software. (Most of the Unix operating system was written using C.) Today it is widely used for writing applications, including word processing, spreadsheets, games, robotics, and graphics programs. It is now considered a necessary language for programmers to know.

Here are the advantages and disadvantages of C :

Advantages

1. C is flexible, high-level, structured programming language.
2. C includes many low-level features that are normally available only in assembly or machine language.
3. C programs are very concise, due to the large number of operators within the language.
4. C is a weakly typed language and the programs written in it compile into small object programs that run or execute efficiently.

5. C works well with microcomputers.
6. C has a high degree of portability—it can be run without change or little change on a variety of computers.

The advantages written above are also the important characteristics of C language.

NOTES

Disadvantages

1. C is considered difficult to learn.
2. Because of its conciseness, the code can be difficult to follow.
3. It is not suited to applications that require a lot of report formatting and data file manipulation.

For learning how to write programs in C, we must first know what alphabets, numbers and symbols are used, then how using them constants, variables and keywords are formed, and finally how are these combined to form an instruction. A program is written using a group of instructions. Figure 1 illustrates this :

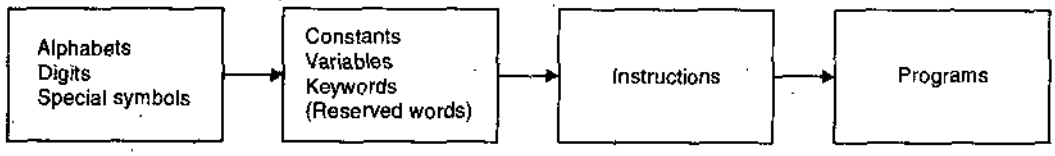


Fig. 1. *Systematic way of learning C language.*

4.2. C CHARACTER SET

A programming language processes some kind of data and provides some meaningful result known as information. The data and information are represented by the character set of the language. A sequence of finite instructions is written following the syntax rules (or grammar) called the program, for getting the desired result. Every program instruction must be coded precisely and user friendly way to help the programmer and others. C language has its own character set for coding compact and efficient programs.

A character represents any alphabet, digit or special character used to form words, numbers and expressions. The C character set can be divided into the following groups :

Alphabets — A...Z and a ... z

Digits — 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Special Characters — ~ ! # % ^ & * ( ) – + = ¦ ' " [ ]
{ } ; : , / \ ? > < . @ _ (blank)

White Space Characters — Space, Tab, Carriage return, form feed and newline.

NOTES

The white space characters are ignored by the C compiler in cases where they are not the part of a string constant. We can use white spaces for separation of words but these are not allowed between the characters of reserved words (keywords) and identifiers (name of some program element).

4.3. CONSTANTS

In C, all the data processed by a program is stored either as a variable or a constant. Constants are the data items that never change their value during the program execution. C constants can be divided into two major categories :

1. Primary constants
2. Secondary constants.

These constants are further categorised as shown in Figure 2.

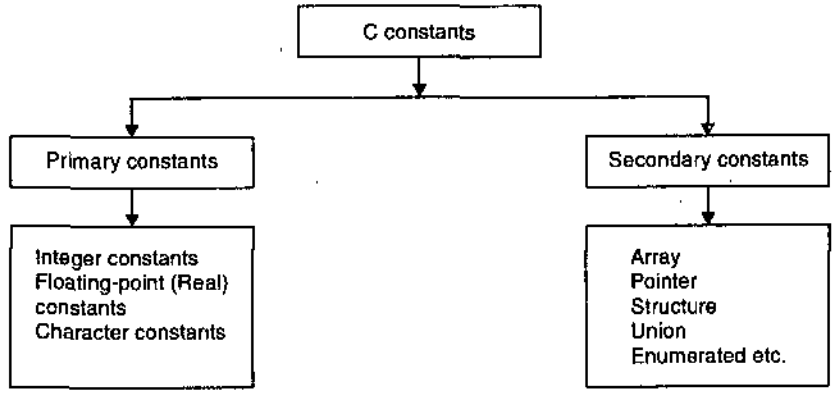


Fig. 2. *Categories of C constants.*

Let us discuss some C constants which are easy to follow at this stage :

- Integer constants
- Floating-point constants
- Character constants
- Back slash constants
- String constants

4.3.1 Integer Constants

These are whole numbers without any fractional part. The following rules are followed for constructing integer constants :

(*a*) It must have at least one digit.

(*b*) It must not contain a decimal point.

(*c*) It may either have + or – sign.

(*d*) When no sign is present it is assumed to be positive.

(*e*) Commas and blanks are not permitted in it.

NOTES

In C, there can be three types of integer constants :

Decimal (base or radix 10)

Octal (base or radix 8)

Hexadecimal (base or radix 16).

Decimal Integer Constants. These consist of a sequence of digits, 0 through 9.

The following are valid integer constants :

1024 3313 275 –12 32767 –5000

The following are invalid integer constants :

.135 (Decimal point not permitted)

2,450 (Comma not permitted)

51237 (Value larger than permitted range)

Range of Integer Constants. The computer memory consists of a number of cells called words. The number of bits in a word is called wordlength. Generally the PC's have wordlength of 16 bits (some may have 32 or more). Let us take n as the wordlength. Then the integer range is given by -2^{n-1} to $2^{n-1}-1$.

If $n = 16$, the integer range is -2^{16-1} to $2^{16-1}-1$.

i.e., – 32768 to 32767.

If an integer less than – 32768 is given, an underflow error will occur and if the integer is more than 32767 an overflow error will occur.

Table 1 illustrates the types of integers with their range and format for 16-bit wordlength :

Table 1. Types of integers with their range

Type	*No. of bytes*	*Allowed range*		*Format*
		From	*To*	
int/short	2	– 32,768	32,767	%d
unsigned int/short	2	0	65,535	%u
long int	4	– 2,147, 483, 648	2,147, 483, 647	%ld
unsigned long int	4	0	4,294, 967, 295	%lu

NOTES

Octal Integer Constants. These are preceded by 0 (digit zero) and consist of sequence of digits 0 through 7. For example, decimal integer 14 will be written as 016 as octal integer (as $14_{10} = 16_8$). The following are some valid octal integer constants :

0235
047
0

Hexadecimal Integer Constants. These are preceded by 0x or 0X and consist of digits 0 through 9 and may also include alphabets. A through *F* or a through *f*. The letters A through *F* denote the numbers 10 through 15. For example, decimal integer 14 will be written as 0*XE* as hexadecimal integer (as $14_{10} = E_{16}$).

The following are some valid hexadecimal integer constants :

0*X*7
0*x*5*F*
0*X*a*ef*

Note : *Octal and hexadecimal numbers are rarely used in programming.*

The suffix *l* or *L*, *u* or *U* and *ul* or *UL* allow any constant to be represented as long, unsigned or unsigned long respectively.

4.3.2 Floating-point Constants (Real Constants)

These have fractional parts to represent quantities like average, height, area etc. which cannot be represented by integer numbers precisely.

These may be written in either **fractional form** or **exponent form.**

A real constant could be written in the following form :

[sign] [integer] · [fraction] [exponent]

where the integer part or the fractional part may be omitted but not both.

The following rules are followed for constructing real constants in fractional form :

(*a*) A floating-point constant in fractional form must have at least one digit before and after the decimal point.

(*b*) It may either have + or – sign.

(*c*) When no sign is present it is assumed to be positive.

(*d*) Commas and blanks are not permitted in it.

The following are valid real constants in fractional form :

12.5 – 15.8 – 0.0055 336.0

The following are invalid real constants in fractional form :

125 (Decimal point missing)

5,415.6 (Comma not allowed)

NOTES

The exponent form consists of two parts : **mantissa** and **exponent.** These are usually used when the constant is either too small or too big. But there is no restriction for us to use exponential form for other floating constants.

In exponent form the part before 'e' is called mantissa and the part after 'e' is called exponent. We can write **e** or **E** for separating the mantissa and exponent. Since the decimal point can "float" due to use of exponent, the number represented in this form gives the floating-point representation. The following rules are followed for constructing real constants in exponent form :

(*a*) The mantissa and exponent are separated by **e.**

(*b*) The mantissa must be either an integer or a proper real constant.

(*c*) The mantissa may have either + or – sign.

(*d*) When no sign is present it is assumed to be positive.

(*e*) The exponent must be at least one digit integer (either positive or negative). Default sign is +.

The following are valid real constants in exponent form :

+ 15.8E5 .05E – 3

The following are invalid real constants in exponent form :

–125.8 E (No digit specified for exponent)

15.7 E 2.5 (Exponent cannot be fraction)

Range of Floating Constants

The range of floating constants in exponent form on a 16-bit PC is – 3.4e38 to 3.4e38. It occupies 4 bytes of memory. This range is for single precision real numbers. The precision of this data type is seven decimal digits. Whereas the double precision value occupies 8 bytes of memory, range is from – 1.7e308 to 1.7e308 and the precision is fourteen decimal digits.

4.3.3 Character Constants

These consist of a single character enclosed by a pair of single quotation marks. A character value occupies 1-byte of memory. A character constant cannot be of length more than 1. *For example,*

'A' '9' ';' ' '

Here the last constant represents a blank space. The character constant '9' is different from the number 9. Each character constant has an ASCII value

NOTES

associated with it. For example, the following statements will print 65 and A respectively :

```
printf("%d", 'A');
printf("%c", '65').
```

Arithmetic operations on characters are possible due to the fact that each character constant associates an integer value with it.

4.3.4 Back Slash Constants or Escape Sequences

These character constants represent one character, although they consist of two characters. These are also known as escape sequences. These are interpreted at execution time. The values of these characters are implementation-defined.

C uses some characters such as line feed, form feed, tab, newline etc. through execution characters *i.e.*, which cannot be printed or displayed directly. Table 2 shows some of the escape sequence characters or back slash character constants :

Table 2. Escape sequence characters

Escape sequence	*Meaning*	*Execution time result*
'\0'	End of string	NULL
'\n'	End of line	Takes the control to next line
'\r'	Carriage return	Takes the control to next paragraph
'\f'	Form feed	Takes the control to next logical page
'\t'	Horizontal tab	Takes the control to next horizontal tabulation position
'\v'	Vertical tab	Takes the control to next vertical tabulation position
'\b'	Back space	Takes the control to the previous position in the current line
'\\'	Back slash	Presents with a back slash \
'\a'	Alert	Provides an audible alert
'\" '	Double quote	Presents with a double quote

4.3.5 String Constants

These consist of a sequence of characters enclosed in double quotes. The characters enclosed in double quotes can be alphabets, digits, blank space and special characters. *For example,*

```
"Hello ! World"
"Year 2010"
```

"Sum"

"A"

Remember that a character constant (*e.g.*, 'A') is not equivalent to the single character string constant (*e.g.*, "A") in C language. As mentioned earlier the equivalent integer value for 'A' is 65 but "A" does not have any such value. We use strings in C programs for providing suitable messages in output statements. Character strings are quite useful in certain situations and are discussed later on.

NOTES

4.4 VARIABLES

In C, *a variable is an entity that may be used to store a data value and has a name.* Variable names are names given to different memory locations that store different kind of data. All C variables must be declared in the program before their use. The value associated with a variable may vary during program execution. For example,

Let 2 be stored in a memory location and a name **sum** is given to it. On assigning a new value 7 to the same memory location **sum,** its earlier contents are overwritten, since a memory location can store only one value at a time. Figure 3 illustrates this :

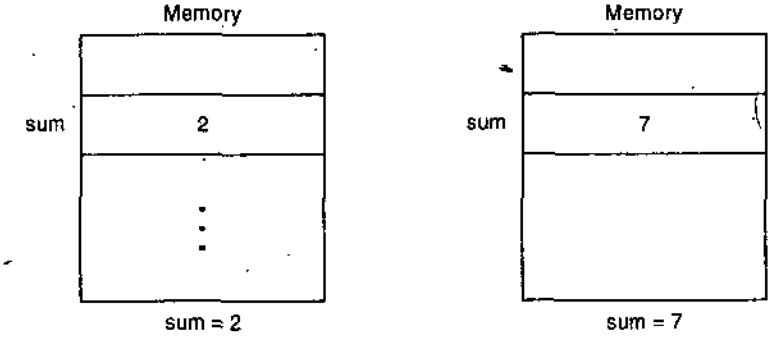


Fig. 3. *Illustrating change in value of a variable.*

Variable names should be user friendly. *For example,*

sum

num

count

average

principal

rate

time

NOTES

If we want to calculate the sum of two numbers, the variable can be named as 'sum' rather than 'tangura' or some other cryptic (difficult to understand) name.

4.4.1 Rules for Defining Variables

As seen earlier, the rules for constructing different types of constants are different but for constructing variable names in C the rules are same for all types. The following rules must be followed while naming variables :

(*a*) A variable name consists of alphabets, digits and the underscore (–) character. The length of variable should be kept upto 8 characters though your system may allow upto 40 characters.

(*b*) They must begin with an alphabet. Some systems also recognize an underscore as the first character.

(*c*) White space and commas are not allowed.

(*d*) Any reserved word (keyword) cannot be used as a variable name.

For example,

simple_interest, avg_marks, total_salary are valid but these should be written as si_int, av_marks, tot_sal for sure recognization by the compiler.

4.4.2 Declaration of Variables

As stated earlier, all variables in a C program **must** be declared before their use. The type of the variables must be specified at the beginning of the program. The declaration of a variable in C informs about :

1. The name of the variable
2. The type of data to be stored by the variable.

Following are the examples of type declaration statements :

```
int    i, j;
float  x, y;
char   choice;
```

A meaningful name given to a variable always helps in better understanding of the program. A variable can be used to store a value of any valid data type. The general syntax for declaration of a variable is given below :

```
type var1, var2, ....., varn;
```

Here, type specifies the data type such as int, float, char etc. and var1, var2, ..., varn are variable names separated by commas. Note that the declaration statement must end with a semicolon (as shown in all the three declaration statements above).

4.5 DATA TYPES IN C

NOTES

A data type is a finite set of values along with a set of rules for allowed operations.

C supports several different types of data, each of which is stored differently in the computer's memory. Data types in C are shown with the help of Figure 4.

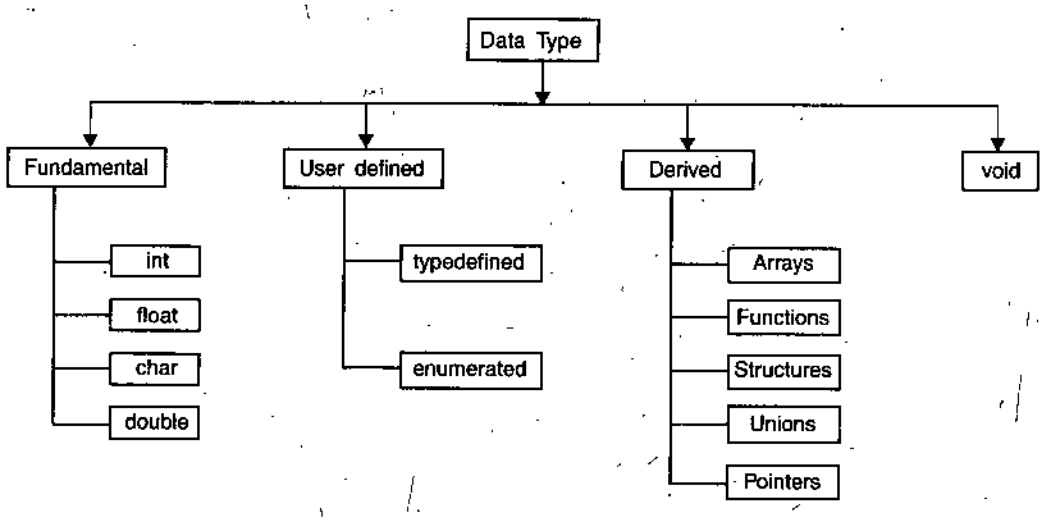


Fig. 4. *Various data types in C.*

The fundamental and user defined data types have been discussed below. The remaining data types will be discussed in the due course.

4.5.1 The Four Fundamental Data Types

Fundamental data types include the data types at the lowest level *i.e.*, those which are used for actual data representation in the memory of the computer. All other data types are based on the fundamental data types.

The fundamental data types in C are :

char — for characters and strings
int — for integers
float — for numbers with decimals
double — for double-precision floating numbers.

Since, the above data types are fundamental *i.e.*, at the machine level, the storage requirement is hardware dependent. Table 3 shows the storage requirement of the above data types on a 16-bit machine :

Table 3. Storage requirements of fundamental data types in C

Data type	*Size (bits)*	*Range of values*
char	8 (1 byte)	– 128 to 127
int	16 (2 bytes)	– 32768 to 32767
float	32 (4 bytes)	– 3·4e38 to 3·4e38
double	64 (8 bytes)	– 1·7e308 to 1·7e308

NOTES

Table 4 represents the various data types in C-basic data types and qualifier. A qualifier changes the characteristics of the data type, such as its size or sign.

Table 4. Various data types in C-basic data types and qualifier

Keyword	*Representation of data type*
char	a single character
int	an integer
float	a single precision floating point number
double	a double precision floating point number
unsigned char	an unsigned single character
signed char	a signed single character
signed int	a signed integer
unsigned int	an unsigned integer
long int	a long integer
short int	a short integer
long double	an extended precision floating point number
signed short int	a signed short integer
unsigned short int	an unsigned short integer
signed long int	a signed long integer
unsigned long int	an unsigned long integer

For altering the size we use — **short** and **long**

For altering the sign we use — **signed** and **unsigned**

Generally size qualifiers cannot be used with data types **float** and **char,** and sign qualifiers are not applicable with **float, double** and **long double.** Also note that the precision means the number of significant digits after the decimal point.

Here **signed** means that the variable can take both + and – values and **unsigned** means only + values are allowed. The keywords **short** means the variable takes lesser number of bits and **long** means greater number of bits.

Remember that **char** or **signed char** mean the same, **int** or **short** or **short int** or **signed short int** mean the same, **long** or **long int** or **signed long int** mean the same. If we use **short, long** or **unsigned** without a basic data type specifier, the data type is taken as an **int** type by the C compilers.

The following program segment illustrates the declaration of different types of variables :

```
main()
{
    int  i, j;
```

```
float a, b;
char  grade;
short int n;
long int  population;
double    value ;
- - - - -      /* executable statements */
}
```


Note. *All floating arithmetic computations in C are carried out in double mode. Whenever, a* ***float*** *appears in an expression, it is changed to double mode. When a double has to be converted to float, double is rounded before truncation to float length.*

4.6 USER-DEFINED TYPE DECLARATION

In C, we can define an identifier for representing an existing data type. It is known as "type definition" and can be used for declaration of variables. The syntax for type definition is given below :

typedef type identifier;

Here, type represents an existing data type and identifier represents the new name in place of type. Remember that **typedef** is not capable of creating any new data type but provides only an alternative name to an existing data type. *For example,*

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

The above statements tell the compiler to recognize **number** as an alternative to **int** and **amount** to **float.**

Now, the variables can be created using **number** and **amount** as given below :

```
number num1, num2;
amount fund, loan, instalment;
```

The significance of **typedef** is that we can provide suitable names to existing data types and make the code easier to read and understand.

Using **typedef** does not replace the existing standard C data type name with the new name, rather we can now use both for creating variables.

4.7 ENUMERATED DATA TYPES

C provides another user-defined data type known as "enumerated data type". It attaches names to numbers, thereby increases the readability of the program.

NOTES

It is generally used when we know in advance the finite set of values that a data type can take on. The syntax for enumerated data type is given below :

enum identifier {val1, val2, ..., valn};

Here, identifier represents the user defined enumerated data type and val1, val2, ..., valn are called members or enumerators.

Now, we can declare variables using the above declared type :

enum identifier var1, var2, ..., varn;

The variables var1, var2, ..., varn can take only one value out of val1, val2, ..., valn.

For example,

enum months {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};

enum months month1, month2; /* variables declared of type months */

Enumerated means that all values are listed. The enumerators are automatically assigned values starting from 0 to n–1. Thus the first value Jan will have 0, the second value Feb will have value 1, and so on, lastly the value Dec will have value 11.

The assignment, arithmetic and comparison operations are allowed on enumerated type variables. *For example,*

```
month1 = Jan;
month2 = Jul;
printf ("%d", month2-month1);
```

The above output statement will print 6 as Jul has value 6 and Jan a 0 with it.

The following statement will work as well,

```
if (month1 < month2)
        .....
else
        .....
```

We can change the default value assignment by assigning the integer values to the enumerators. *For example,*

enum months {Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};

Now the enumerators will have values from 1 to 12 respectively.

The default ordinal values can be changed for more than one enumerator also.

We can combine the definition and declaration of enumerated variables.

For example,

enum months {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec} month1, month2;

The values assigned to the enumerators need not be distinct, increasing or positive. *For example,*

```
enum values {mid = 16383, low = 0, high = 32767};
```

NOTES

4.8 ASSIGNMENT OPERATOR

An operator is a symbol that operates on a certain data type.

In C, the '=' symbol is known as the assignment operator. It sets the value of the variable on the left hand side of it to that of the right hand side of it.

4.8.1 Assignment Expressions and Assignment Statements

In C, an expression is a combination of variables, constants and operators written according to the syntax. An expression always results in some value of a certain type that can be assigned to a variable using = operator.

For example,

```
interest = (prin * rate * time)/100.0;
```

Here, the variables prin, rate, time used in the expression on the right hand side of = must be declared with data types and assigned values before the execution of the above statement. The above statement is an example of an assignment statement.

C supports a variety of assignment statements. These are given below :

(*a*) Simple assignment statement

(*b*) Multiple assignment statement

(*c*) Arithmetic assignment statement.

(a) Simple Assignment Statement

The syntax of simple assignment statement is

```
var_name = expression;
```

C treats = operator like other operators. *For example,*

```
int i, j, s;
    i = 20;
    j = 30;
    printf ("\nValue of s = %d\n", s = i + j);
```

The above statement is valid and prints the result as :

```
Value of s = 50
```

First the values of i and j are added, assigned to s and then the value of s is printed.

Remember that the type of var_name and expression must be compatible.

NOTES

Let us consider the following variable declarations :

```
int i, j,
float x, y, avg;
```

Now the following assignment statements are valid :

```
i = 55;
j = 250;
x = 14.0;
y = 30.0;
avg = (x + y)/2.0;
```

The following are invalid assignment statements :

```
x + y = avg; /* x + y is not a valid variable name */
avg = 'A'; /* avg is not a character variable */
```

In C, a variable can be initialized when declared. The syntax of this is :

```
type      var_name = constant;
```

For example,

```
int sum = 0;
float avg = 0.0;
char ans = 'y';
```

(b) Multiple Assignment Statement

In C, the same value can be assigned to more than one variables of same type using multiple assignment statement. The syntax of a multiple assignment statement is given below :

```
var_name1 = var_name2 = var_name3 = constant;
```

Here var_name1, var_name2, var_name3 are variable names and constant is the value to be assigned. It can be extended for more than three variables also. The value is assigned from right to left.

For example,

```
int i, j, k, n;
i = j = k = n = 0;
```

(c) Arithmetic Assignment Statement (Self Replacement Statements)

When the variable itself takes part in the assignment statement and stores the result in itself, the arithmetic assignment statement comes into picture. *For example,*

```
int num1, num2;
    num1 = 100;
    num2 = 250;
    num1 = num1/20; /* arithmetic assignment statement */
    num2 = num2 * 4; /* arithmetic assignment statement */
```

NOTES

4.9 OPERATORS AND EXPRESSIONS

Operators are the verbs of a language that help the user perform computations on values. C language supports a rich set of operators. Different types of operators discussed in this UNIT are :

1. Arithmetic operators
2. Unary operators
3. Relational operators
4. Logical operators
5. Assignment operators
6. The conditional operator
7. Bitwise operators

An expression is a formula consisting of one or more operands and zero or more operators connected together to compute the result. An operand in C may be a variable, a constant or a function reference. For example, in the following expression,

x – y

minus character '–' is an operator and x, y are operands.

4.10 ARITHMETIC OPERATORS

These are used to perform arithmetic operations. All of these can be used as binary operators. These are :

+	add	
–	subtract	
*	multiply	
/	divide	(the divisor must be non zero)
%	modulo	(gives remainder after division on integers)

The parenthesis( ) are used to clarify complex operations. The operators + and – can be used as unary plus and unary minus arithmetic operators also. The unary – negates the sign of its operand.

Note : **C language has no operator for exponentiation.**

The function **pow** (x, y) which exists in **math.h** returns x^y.

Following are some examples of arithmetic operators :

x + y, x – y, x * y, x/y, x % y, – x * y

Here x and y are operands. The % operator cannot be used on floating point data type.

NOTES

4.10.1 Arithmetic Expressions

An expression consisting of numerical values (either any number, variable or even some function call) joined together by arithmetic operators is known as an arithmetic expression. For example, consider the following expression :

(x – y) * (x + y)/5

Here x, y and 5 are **operands** and the symbols –, *, +, / are operators. The precedence of operators for the expression evaluation has been given by using parenthesis which will over rule the operators precedence. If x = 25 and y = 15, then the value of this expression will be 80.

Consider the following arithmetic expression :

3 * ((i%4) * (5 + (j – 2)/(k + 3)))

where i, j and k are integer variables. If i, j and k have values 9, 14 and 6 respectively, then the above expression would be evaluated as

3 * ((9%4) * (5 + (14 – 2)/(6 + 3)))
= 3 * (1 * (5 + (12/9)))
= 3 * (1 * (5 + 1))
= 3 * (1 * 6)
= 3 * 6
= 18

4.11 RELATIONAL OPERATORS

These are used to compare two variables or constants. C has the following relational operators :

Operator	Meaning
= =	Equals
! =	Not equals
<	Less than
>	Greater than
< =	Less than or equals
> =	Greater than or equals

4.12 LOGICAL OPERATORS

In C, we can have simple conditions (single) or compound conditions (two or more). The logical operators are used to combine conditions. The notations for these operators is given below :

Operator	Notation in C
NOT	!
AND	&&
OR	||

NOTES

Note : The notation for the operator OR is given by two broken lines. These follow the same precedence as in other languages - NOT (!) is evaluated before AND (&&) which is evaluated before OR (\|\|). Parentheses( ) can be used to change this order.

4.12.1 Relational and Logical Expressions

An expression involving a relational operator is known as a **relational expression.** The resulting expression will be of integer type, since in C, true is represented by 1 and false by 0.

For example,

```
int a = 2, b = 3, c = 4;
```

Relational expression	Value
$a < b$	1 (true)
$c <= a$	0 (false)
$b == c$	0 (false)
$c >= b$	1 (true)
$a != c$	1 (true)
$a > 1$	1 (true)

When a relational operation is carried out, different types of operands (if present) will be converted appropriately.

Note : Don't use = for testing equality. The operator for testing equality is = = (two = signs/together).

In C, the arithmetic operators have higher priority over relational operators.

Relational operators are used with **if, while, do while** statements to form relational expressions which help in making useful decisions. These statements are discussed later on.

An expression formed with two or more relational expressions is known as a **logical expression** or **compound relational expression.**

For example,

```
int day;
day >= 1 && day <= 31
```

The logical expression given above is true, when both the relational expressions are true. If either of these or both are false, it evaluates as false.

NOTES

4.13 UNARY OPERATORS

C has a class of operators that act upon a single operand to give a new value. These type of operators are known as unary operators. Unary operators generally precede their single operands, though some unary operands are written after their operands.

4.13.1 Unary Minus

Perhaps the most common unary operation is **unary minus,** where a minus sign precedes a numerical constant, a variable or an expression. (Some programming languages permit a minus sign to be included as a part of a numeric constant). In C, however, all numeric constants are positive. Thus, a negative number is actually an expression, having the unary minus operator, followed by a positive numeric constant. Note that the unary minus operation is totally different from the arithmetic operator which means subtraction (–). The subtraction operator requires two separate operands.

For example,

```
– (a + b)
– 3E – 7
```

4.13.2 Increment and Decrement Operators

Two other commonly used unary operators are the **increment operator, + +,** and the **decrement operator, – –,** that operate on integer data only. The increment (++) operator increments the operand by 1, while the decrement operator (– –) decrements the operand by 1. *For example,*

```
int i, j;
i = 10;
j = i + +;
printf ("%8d%8d", i, j);
```

Here, the output would be 11 10. First **i** is assigned to **j** and then **i** is incremented by 1 *i.e.,* post-increment takes place.

If we have

```
int i, j;
i = 20;
j = + + i;
printf ("%8d%8d", i, j);
```

The output would be 21 21. First **i** is incremented by 1 and then assignment takes place *i.e.,* pre-increment of **i.**

All the three statements given below are identical :

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

All of these increment the value of i by 1.

Now, consider the example for (– –) operator :

```
int i, j
i = 10;
j = i – –;
printf ("%8d%8d", i, j);
```

Here the output would be 9 10. First **i** is assigned to **j** and then **i** is decremented by 1 *i.e.,* post-decrement takes place.

If we have

```
int i, j;
i = 20;
j = – – i;
printf ("%8d%8d", i, j);
```

The output would be 19 19. First **i** is decremented by 1 and then assignment takes place *i.e.,* pre-decrement of **i.**

> ***Note :*** *On some compilers a space is required on both sides of + + i or i + +, i– – or – – i.*

4.13.3 The sizeof Operator

It is an unary operator which provides the size, in bytes, of the given operand. The syntax of **sizeof** operator is :

sizeof(operand)

Here the operand is a built in or user defined data type or variable.

The **sizeof** operator always precedes its operand.

For example,

```
sizeof (float)
```

returns the value 4.

This information is quite useful when we execute our program on a different computer or a new version of C is used. The **sizeof** operator helps in case of dynamic memory allocation for calculating the number of bytes used by some user defined data type.

NOTES

A *cast* is also considered to be a unary operator. It has been discussed earlier. In general terms, a reference to the cast operator is written as given below :

(type)

Thus, the unary operators we have discussed so far in this book are –, + +, – –, sizeof and *(type).*

Unary operators have a higher precedence than arithmetic operators. Hence, if a unary minus operator acts upon an arithmetic expression that contains one or more arithmetic operators, the unary minus operation will be carried out first (if parentheses do not enclose the arithmetic expression). Also the associativity of the unary operators is right-to-left, though consecutive unary operators rarely appear in simple programs. *For example,*

```
int x, y;
x = 5;
y = 25;
```

The value of the expression – x + y will be – 5 + 25 = 20. Here the unary minus is carried out before the addition operation. If the expression is – (x + y), then it becomes – (5 + 25). The value of this expression is – (5 + 25) = – 30. In this case, the addition is performed first and then the unary minus operation. C has several other unary operators. These will be discussed, as the need arises.

4.14 ASSIGNMENT OPERATORS

There are several different assignment operators in C. All of these are used to form *assignment expressions,* which assign the value of an expression to an identifier.

The most commonly used assignment operator is = . Assignment expressions which use this operator are of the following form :

identifier = expression

where *identifier* generally represents a variable and *expression* represents a constant, a variable or a more complex expression.

For example,

```
int i = 10;
float a = 5.0;
```

The ***lvalue*** is an entity that appears on the left side of an assignment statement, whereas the value of something that appears on the right side is called an *rvalue.* The ***lvalue*** must have storage space associated with it, that is, has memory address. If the storage space is not assigned to the entity appearing on the left, the assignment statement will not be compiled.

The following are not *lvalues* and hence **cannot** appear on the left side of an assignment statement :

type names

constants—(numeric, character and literal)

function, array names

enumerated data types

NOTES

The function names have memory addresses but are protected by the C compiler.

C has the following five additional assignment operators :

+ =, − =, * =, /= and % =

These are known as **shorthand assignment operators** or **arithmetic assignment operators.**

4.15 FUNCTIONS

A function groups a number of program statements into a single unit and gives it a name. Every C program is a collection of functions. The function **main()** is executed first and it calls the other functions directly or indirectly. It is necessary that every C program must have the function main(). We may have user defined function(s) which can be called (invoked) from other parts of the program. Functions are the building blocks of C programs where all the program activity occurs.

Monolithic program (a large single list of instructions) becomes difficult to understand, debug, test and maintain. For this reason functions are used. A function has a clearly-defined objective (purpose) and a clearly-defined interface with other functions in the program. A function is also called a **subprogram** and it is easy to understand, debug and test. Reduction in program size is the another reason for using functions. Any sequence of statements that is repeated in a program can be combined together to form a function. The function code is stored in only one place in memory, even though it may be executed as many times as a user needs thus saving both time and space.

We can use a function (already tested) in many programs and thus only the additional coding is required.

4.15.1 Advantages of Using Functions in C

Some advantages of using functions in C are listed below :

(*i*) A complex program can be divided into small subtasks and function sub-programs can be written for each.

(*ii*) These are easy to write, understand and debug.

NOTES

(iii) A function can be utilised in many programs by separately compiling it and loading them together.

(iv) C has the facility of defining a function in terms of itself *i.e.,* recursion. Recursion suits to some processes but not all.

(v) Many functions such as **scanf(), printf(), sqrt()** etc. are kept in C library and the compiler of C is written for calling any such function.

4.16 DEFINING AND USING FUNCTIONS

In C, functions return an **int** value by default *i.e.,* when no type has been specified before the function name it will always return an **int** value.

4.16.1 Function Prototype

Like any variable in a C program it is necessary to prototype or declare a function before its use, if it returns a value other than an **int.** It informs the compiler that the function would be referenced at a later stage in the program. The general form of function prototype is :

```
type function_name(argument list);
```

When we place the function prototype above all the functions (including **main()**), it is known as a **global prototype.** A prototype declared in global environment is available for all the functions in the program.

When we place the function prototype inside the definition of another function (*i.e.,* in the local declaration section), the prototype is known as a **local prototype.** Such declarations are primarily used by the functions containing them.

It is a good programming style to have *global prototypes* for adding flexibility and enhance documentation. Function prototypes are *not mandatory* (compulsory) in C. These are desirable, however, because these help in error checking between the calls to a function and the corresponding function definition.

> ***Note :*** *The function prototype is always terminated with a semicolon.*

4.16.2 Function Definition

In C, a function must be defined prior to its use in the program. The function definition contains the code for the function.

4.16.3 Eliminating the Prototyping

If the called function definition appears before the calling function's definition, then the called function's prototype may be avoided.

NOTES

4.16.4 Calling or Invoking or Accessing Functions

A function can be called (*i.e.,* invoked) by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. If no arguments, an empty pair of parentheses must follow the function's name. The function call may appear by itself, or it may be one of the operands within an expression (in case it returns a value). The arguments or parameters appearing in the function call are known as *actual arguments,* in contrast to the *formal arguments* that appear in the first line of the function definition. In a normal function call, there will be one actual argument for each formal argument. The actual arguments may be expressed as constants, single variables or more complex expressions. However, *actual arguments must match in number, type and order with their corresponding formal arguments.*

Calling Convention

It specifies the order in which arguments (parameters) are passed to a function when a function is called. There are two possibilities :

(*i*) Parameters might be passed from left to right.

(*ii*) Parameters might be passed from right to left.

C language obeys the second order.

For example, consider the following function call :

```
max(a, b, c);
```

In this call it doesn't matter whether the parameters are passed from left to right or from right to left. However, in some cases the order of passing parameters becomes an important thing. *For example,*

```
int x = 10;
printf("%d%d%d", x, --x, x--);
```

It looks that this **printf()** would output 10 9 8.

This however is not, the case. Actually, it outputs 8 8 10.

This is because C's calling convention is from right to left. That is, firstly 10 is passed through the expression **x--** and then **x** is decremented to 9. Then result of **--x** is passed. That is, **x** is decremented to 8 and then passed. Finally latest value of **x,** *i.e.,* 8, is passed. Thus in right to left order 10, 8, 8 get passed. Once **printf()** collects them it prints them in the order in which we have asked it to get them printed (and not the order of passing). Thus, 8 8 10 gets printed as the result.

NOTES

Note : *We must include the appropriate header files when the standard library functions are used.* For example, ***<stdio.h>*** *for all input/output functions and* ***<math.h>*** *for all mathematical functions. A successfully compiled C program might have a mismatch in the format specifiers and the variables in the list, in such cases either garbage values get printed or no values are printed by* ***printf()*** *function.*

In C programs, functions that have parameters are called in one of the two ways :

- Call by value
- Call by reference

Call by Value

The following program illustrates the concept of call by value method :

```
/* find largest of three numbers using function */

#include<stdio.h>
main()
{
   void max(int x, int y, int z); /* function prototype */
   int a,b,c;
   clrscr();
   printf("Enter the three numbers\n");
   scanf("%d%d%d",&a,&b,&c);
   /* echo the data */
   printf("\na=%8d b=%8d c=%8d",a,b,c);
   /* function call - call by value method */
   max(a,b,c); /* a,b,c are actual parameters or arguments */
   getch(); /* freeze the monitor */
}
/* function definition max() */
void max(int x, int y, int z)  /* x,y,z are formal parameters */
{
   int big; /* local variable declaration */
   printf("\n\nx=%8d y=%8d z=%8d\n",x,y,z);
   big=x;
   if(y>big)
      big=y;
   if(z>big)
      big=z;
   printf("\n\nLargest of three numbers is %d\n",big);
}
```

PROGRAM 1

The output of Program 1 will be :

```
Enter the three numbers
60  75  40
a=      60  b=      75  c=      40
x=      60  y=      75  z=      40
Largest of three numbers is 75.
```

In the above program, values entered for the variables *a, b* and *c* in the *main()* function are passed to the function *max()*. These values get copied into the memory locations of the arguments *x, y* and *z* respectively of the function *max()* when it is called. This is shown in Figure 5.

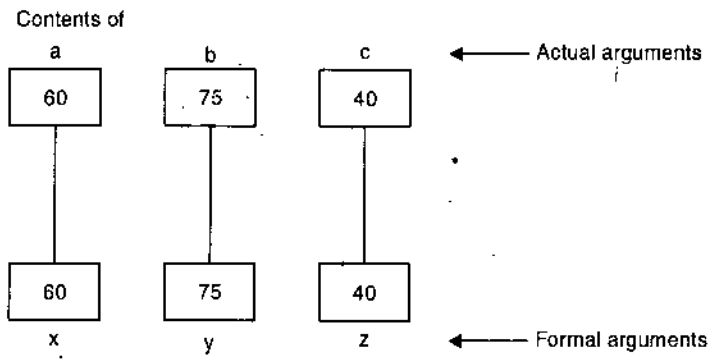


Fig. 5. *Example of* ***call by value.***

> ***Note :*** *If any function is called with different actual argument(s) more than once in a program, the same set of statements within the function are executed. Without using function, the same set of statements would have to be coded the required number of times in the function main().*

The scope of local variables is local to the function in which these are defined and not outside it. *The most important point to remember while using arguments is that the actual and formal arguments should match in number, type and order.* The actual and formal parameters use different memory locations and any change in formal arguments is not reflected back in the calling function using *call by value* method.

Call by Reference

We know that variables are stored somewhere in memory. In case we are able to pass the location number (also known as address) of the variable to a function, we obtain the *call by reference* method. So, we must be familiar with the knowledge of how to make a 'call by reference'. For this purpose we require at least an elementary knowledge of a concept called 'pointers'. So, let us know about the basics of pointer, which will help a lot in understanding *call by reference* method of calling C functions.

NOTES

NOTES

4.17 CATEGORY OF FUNCTIONS

In C, the functions can be divided into the following categories :

(*i*) Functions with no arguments and no return values.

(*ii*) Functions having arguments but no return values.

(*iii*) Functions having arguments and return values also.

(*i*) Functions with no Arguments and no Return Values

When a function is without arguments (parameters), it does not receive any data from the calling function nor does the calling function get any data from the called function. So, we can say that no data exchange takes place in any direction. Such type of function cannot be used in any expression and it is always coded as a standalone statement. *For example,* the declaration

void display **(void);**

implies that the function **display()** takes no arguments or returns no value. Now the function definition is

```
void display (void)
{
   printf("Always enjoy your life\n");
}
```

For calling this function, we use

display();

(*ii*) Functions having Arguments but no Return Values

There are situations when the calling function sends the data to the called function after validating it (if necessary). Program 1 falls under this category.

(*iii*) Functions having Arguments and Return Values also

Sometimes we may require that the result from the called function is needed by the calling function for further use. Such type of function assures a high degree of portability between programs as it provides a two way communication between the calling and the called functions.

There is no restriction on the number of **return** statements in a function. Also, note that the **return** statement need not always be present at the end of the called function in a C program.

So far in the programs having functions with arguments, different variable names have been used for *actual* and *formal* arguments. Make it clear that the 'actual argument(s)' and 'formal argument(s)' can have the same name but the

compiler would treat them as different due to their presence in different function.

In all the programs having formal parameters or arguments, the arguments have been used taking into account the **ANSI** method which is more common these days. There is another method also known as **Kernighan** and **Ritchie** (or just K and R) method, which can be used for declaration of formal arguments.

For example,

```
/* function definition f() */
float f(x)
float x; /* formal argument declaration */
{
   if(x < -3)
      return(-3);
   if(x>=-3 && x<=3)
      return(x);
   else
      return(3);
}
```

4.18 RECURSION

In C, a function can call itself, this is called recursion. A function is said to be recursive if there exists a statement in its body for the function call itself. Recursion is sometimes called 'circular definition'.

The main advantage of recursion is that it is useful in writing clear, short and simple programs.

Recursion is implemented using C language, by **defining a function in terms of itself,** *i.e.,* function invokes itself.

A commonly used example of a recursive procedure is finding the factorial of a number. The factorial of a number n (denoted as n!) is defined as :

$$n! = n \times (n - 1) \times (n - 2) \times ..... \times 1$$

which can be expressed as :

$$n! = n \times (n - 1) !$$

This is an example of a recursive definition wherein the factorial of a number is defined **in terms of itself,** *i.e.,* the factorial **n!** is defined in terms of the factorial (n – 1) !.

NOTES

Obviously, this recursive definition should have a **stopping condition,** otherwise an infinite loop will result. In this case it is given below :

$$0 ! = 1$$

Thus, the complete definition of the factorial function is :

$$n ! = n \times (n - 1) ! \quad \text{with} \quad 0 ! = 1$$

Before writing a recursive function for finding factorial, let us code it **non-recursively :**

```
/* find factorial of a number using a non-recursive function */
#include<stdio.h>
main()
{
   int n;
   long int factorial(int n);  /* function prototype */
   clrscr();
   printf("Enter the number for finding factorial : ");
   scanf("%d",&n);
   if(n<0)
      printf("\nFactorial of %d not defined\n",n);
   else
      printf("\nFactorial of %d = %ld",n,factorial(n)); /*
                                                  factorial call */
}
/* function definition factorial() */
long int factorial(int n)
{
   int i; /* local variable declaration */
   long int prod=1;
   for(i=1;i<=n;i++)
      prod=prod*i;
   return(prod);
}
```

PROGRAM 2

The output of Program 2 will be :

Enter the number for finding factorial : 10

Factorial of 10 = 3628800

Enter the number for finding factorial : – 7

Factorial of – 7 not defined

NOTES

4.19 ARRAYS

Many applications require the processing of multiple data items that share common properties (*e.g.,* a set of numerical data, represented by a_1, a_2, ... a_n). The individual data items can be characters, integers, floating-point numbers, and so on.

Earlier we have used C basic data types. C provides the *derived* data types also, which are built from the basic integer and floating data types. An array is a C *derived* type that can store several values of one type. *An array is a collection of homogeneous (same type) elements that are referred by a common name.* It is also called a **subscripted variable** as the array elements are used by the name of an array and an index or subscript. Arrays are of two types :

(*i*) *One dimensional array*

(*ii*) *Multi dimensional array (2 or more).*

Here we will study one dimensional and two dimensional arrays.

4.20 ONE DIMENSIONAL ARRAYS

The syntax of declaring a one dimensional array in C is as given below :

```
type  array_name[size];
```

Here *type* declares the **base type** of the array, which is the type of each element of the array. The **array_name** specifies the array name by which the array will be referenced and **size** defines the number of elements the array will store. *For example,*

```
int  a[5];
```

In C the array index always begins with 0. So, a [2] would refer to the third element in the array **a** where 2 is the array index or subscript. The entire array having elements 55, 90, 17, 88 and 36 can be shown as in Figure 6 :

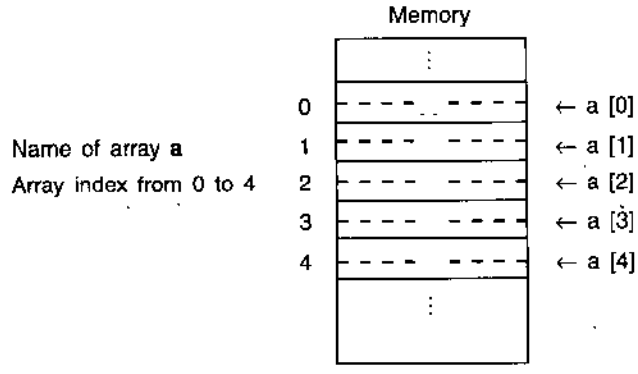


Fig. 6. *Schematic representation of an array* ***a[5].***

NOTES

Since each element in **a** is an integer, it occupies two bytes. Notice that the first element has the index 0. Thus, since there are five elements, the last one is number 4.

Name of an array, without subscripts, also refers to the address of the first element. Thus, **a** or a[0] imply same address. Or, in other words, name of the array is pointer to the first element of the array. The & operator may be used to obtain the address of an array element. Thus, &a[0] is address of the element a[0]. Array may be declared either **static** or **auto.**

We can initialize array (*i.e.*, give value to each array element) when the array is first defined.

The general syntax of initialization of arrays is given below :

type array_name = {list of values comma separated};

Initialization at the time of declaration is known as **Compile Time Initialization.**

For example,

```
int a[5]={55,90,17,88,36};
```

The values to which **a** is initialized are enclosed in braces and separated by commas. They are connected to the array expression by = sign. The array cannot be initialized with only selected elements.

In case we provide all the array elements during initialization the specification of array size is not essential. The compiler will automatically count the number of elements for reserving the space in memory for the array. Thus we can write

```
int a[ ]={55,90,17,88,36};
```

In case we use an explicit array size, but it is not having the number of elements equal to size of array, the missing elements will be set to 0. If the number of elements is greater than the specified size, an error message will be displayed. Consider few more examples of array initialization :

```
float marks[ ] = {80.5,90.6,78.3,59.7,100.0,62.0};
long int salary[4] = {50000,18000,25000};
```

Note that the element salary[3] will be initialized to 0 in the last example.

4.20.1 Inputting Array Elements

For reading the array elements (input operation) we must declare the array first along with the index to be used on the array. Then the array elements can be inputted as given ahead :

```
int a[5],i;
for(i=0;i<5;i++)
        scanf("%d",&a[i]);
```

Such an explicit initialization at run time is known as **Run Time Initialization.**

It is the duty of the programmer or user to check that the number of elements entered must not exceed the size of the array.

No shortcut method is available for initialization of an array having a large number of elements.

NOTES

4.20.2 Accessing Array Elements

By accessing array elements we are either inputting these or outputting or performing some other operation on these. The array elements have been accessed above by using the statement

```
scanf("%d",&a[i]);
```

Here, the expression for array element is

```
a[i]
```

Since **i** is the loop variable in the **for** loop, it starts at 0 and is incremented until it reaches 4, thereby accessing each element of array **a.**

For writing the elements (output operation) from the array, use the following method :

```
for(i=0;i<5;i++)
    printf("%d",a[i]);
```

The following program illustrates the input, output operations on an array and finds the sum and average of n numbers.

```
/* find sum and average of n numbers */

#include<stdio.h>
#define S 10
main()
{
   float a[S],avg,total=0.0;
   int i,n;
   clrscr();
   printf("Enter the number of elements in the array <=%d\n",S);
   scanf("%d",&n);
   printf("\nEnter %d elements\n\n",n);
   for(i=0;i<n;i++)
   scanf("%f",&a[i]);
   /* echo the data */
   printf("\nGiven array is\n\n");
   for(i=0;i<n;i++)
      printf("%8.2f",a[i]);
      /* find the sum */
```

```
    for(i=0;i<n;i++)
       total+=a[i];
    avg=total/n;
    printf("\n\nSum= %.2f\n",total);
    printf("\nAverage = %.2f\n",avg);
    getch(); /* freeze the monitor */
}
```

PROGRAM 3

The output of Program 3 will be :

Enter the number of elements in the array <=10

8

Enter 8 elements

10 24 30 22 50 42 66 71

Given array is

10.00 24.00 30.00 22.00 50.00 42.00 66.00 71.00

Sum = 315.00

Average = 39.38

4.20.3 Passing Array Elements to Function

In C, array elements can be passed to a function in two ways. These are :

(*i*) Calling the function by value

(*ii*) Calling the function by reference

(i) Calling the Function by Value

In this method we pass values of array elements to the function. The following program illustrates this concept :

```
/* passing array elements to a function - call by value method */

#include<stdio.h>
#define SIZE 10
main()
{
   void show(int); /* function prototype */
   int a[SIZE],i,n;
   clrscr();
   printf("Enter number of elements in the array <= %d\n\n",SIZE);
   scanf("%d",&n);
   printf("\nEnter %d elements\n\n",n);
   for(i=0;i<n;i++)
```

```
      scanf("%d",&a[i]);
   printf("\nEntered elements of array are\n\n");
   for(i=0;i<n;i++)
      show(a[i]); /* function call */
   getch(); /* freeze the monitor */
}

/* function definition show ()*/

void show(int value)
{
   printf("%8d",value);
}
```

PROGRAM 4

The output of Program 4 will be :

Enter number of elements in the array <= 10

5

Enter 5 elements

18 27 15 38 91

Entered elements of array are

18 27 15 38 91

In the above program, we are passing an individual array element at a time to the function **show( )** and getting it printed in the function **show( ).** Note that since at a time only one array element is being passed, this element is assigned to formal parameter **value,** in the function **show( )** and printed there.

(ii) Calling the Function by Reference

In this method, we pass addresses of individual array elements to the function. The following program illustrates this concept :

```
/* passing array elements to a function-call by reference method */

#include<stdio.h>
#define SIZE 10
main()
{
   void show(int *); /* function prototype */
   int a[SIZE],i,n;
   clrscr();
```

NOTES

```
    printf("Enter number of elements in the array <= %d\n\n",SIZE);
    scanf("%d",&n);
    printf("\nEnter %d elements\n\n",n);
    for(i=0;i<n;i++)
      scanf("%d",&a[i]);
    printf("\nEntered elements of array are\n\n");
    for(i=0;i<n;i++)
      show(&a[i]); /* function call */
    getch(); /* freeze the monitor */
}

/* function definition show() */
void show(int *value)
{
    printf("%8d",*value);
}
```

PROGRAM 5

The output of Program 5 will be :

```
Enter number of elements in the array <= 10
5
Enter 5 elements
18  27  15  38  91
Entered elements of array are
        18      27      15      38      91
```

In the above program, since at a time the address of one array element is being passed, this address is assigned to formal parameter **value** in the function **show( )**. For printing, the array element in **show( )** we have used *'value at address'* operator **(*)**.

4.21 TWO DIMENSIONAL ARRAYS

So far we have manipulated arrays with only one dimension. In C we can have arrays of two or more dimensions. The two dimensional array is also known as a matrix.

A two dimensional array is a grid having rows and columns in which each element is specified by two subscripts. It is the simplest of multidimensional arrays. *For example,*

NOTES

An array a[m][n] is an m by n table having m rows and n columns containing m × n elements. The size of the array (total number of elements) is obtained by calculating m × n.

Here, a[i][j] denotes the element in the ith row and jth column. Size of the array is m × n.

The syntax of declaring a two dimensional array in C is as follows :

```
type variable_name[number of rows][number of columns];
```

For example, `int a[5][5];`

Here 'a' is the name of the array of type **int** of size 5 by 5.

The array elements are a[0][0], a[0][1], ..., a[4][4].

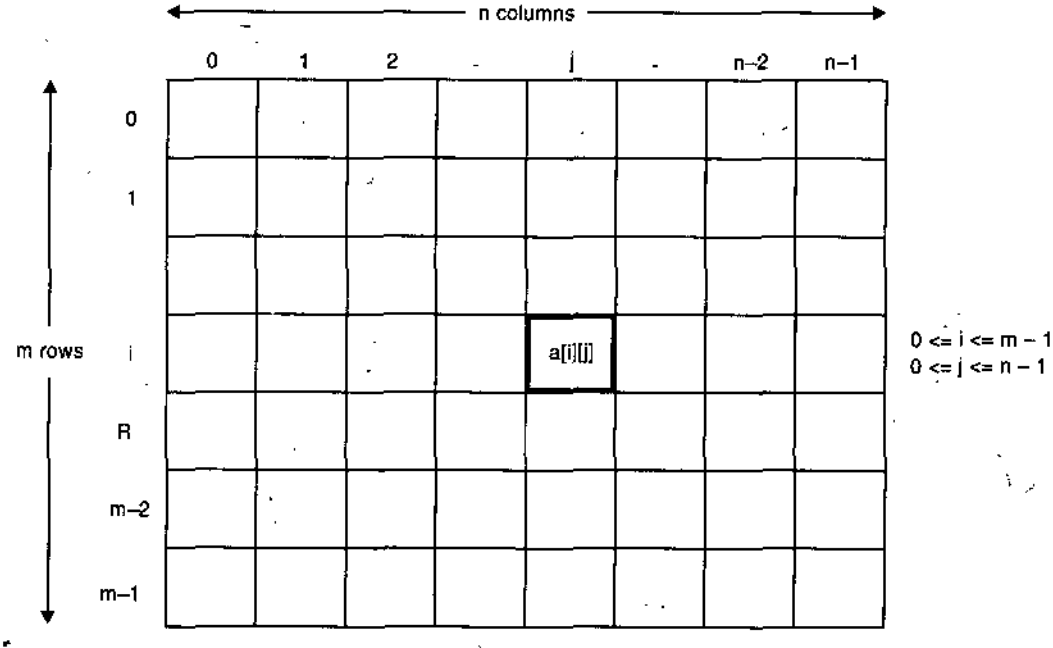


Fig. 7. *Illustration of logical concept of a two dimensional array.*

In C the size of each dimension is closed in a separate bracket.

We can initialize a two dimensional array when the array is first defined. *For example,*

```
float amount[][4] = {
                        {2155.40, 159.65, 937.37, 10918.66},
                        {517.00, 17936.35, 5009.39, 88.75},
                        {7500.60, 7039.55, 4085.25, 837.00}
                    };
```

NOTES

As mentioned earlier such an initialization is known as **Compile Time Initialization.**

Remember that a two dimensional array is really an array of arrays. The format for initializing such an array is based on this fact. The initializing values for each sub-array are enclosed in braces and separated by commas :

```
{2155.40,  159.65,  937.37,  10918.66}
```

and similarly others are enclosed by braces and comma separated.

The second dimension size **must** *be specified when we initialise the two-dimensional arrays in unsized manner but the first dimension size is optional.*

Figure 8 shows how the array **amount [ ][4]** declared above looks :

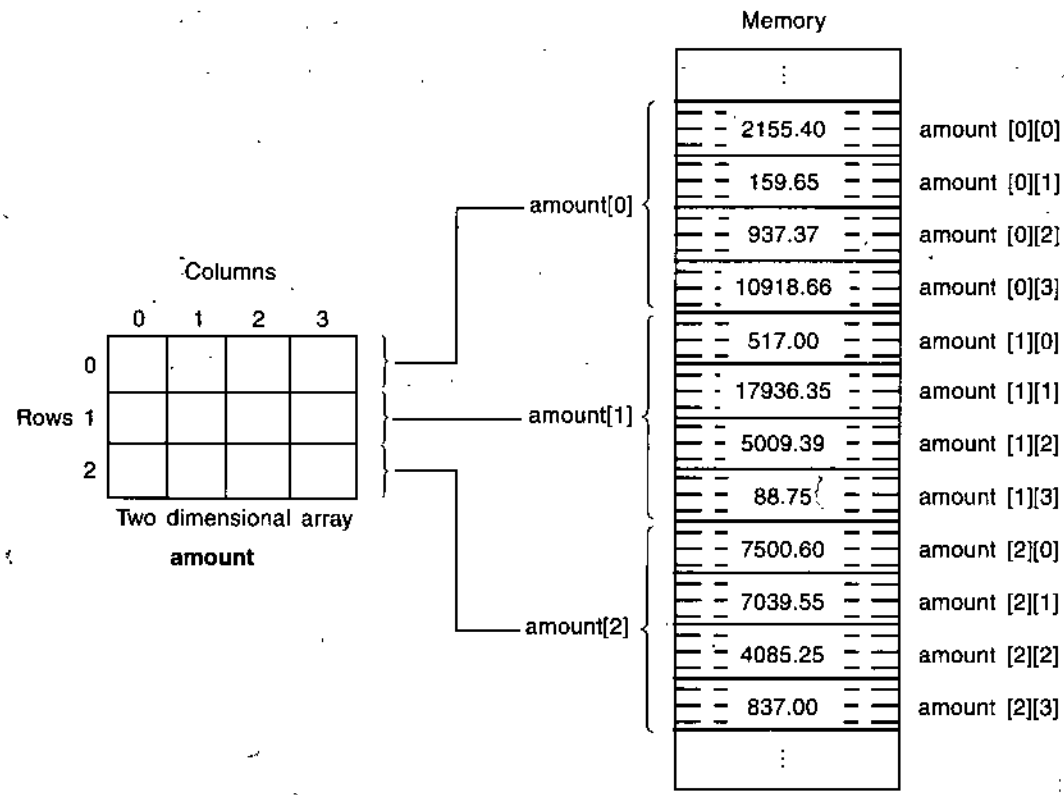


Fig. 8. *Illustration of storage of array amount [ ][4].*

If the number of rows in the above array amount[ ][4] are increased or decreased in the initialization, the array will also grow or shrink accordingly.

Now consider the declaration when partial values are given as :

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

This is equivalent to

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

We should note that the braces mark the end of initialization for a "cluster" of elements within an array.

When all the elements are to be initialized to zero, the following short-cut method may be used.

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

The following statement will also achieve the same result :

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

NOTES

4.21.1 Inputting Array Elements

For reading the two dimensional array elements (input operation) we must declare the array first alongwith the indices to be used on the array. Then the array elements can be inputted as given below :

```
#define S = 5
int a[S][S],i,j,m,n;
printf("Enter the size of two dimensional array <= %d * %d\n",S,S);
scanf("%d%d",&m,&n);
printf("\nEnter the array of size %d * %d\n",m,n);
for(i=0;i<m;i++)
{
   for(j=0;j<n;j++)
      scanf("%d",&a[i][j]);
}
```

As mentioned earlier such an explicit initialization is known as **Run Time *Initialization.***

4.21.2 Accessing Array Elements

By accessing array elements we are either inputting these or outputting or performing some other operation on these. The array elements have been accessed above by using the statements

```
scanf("%d",&a[i][j]);
```

Here, the expression for array element is

```
a[i][j]
```

Since **i** and **j** are the loop variables, **i** is set to **0** and **j** is incremented until it reaches **n,** then **i** is incremented to **1** and **j** again varies from 0 to n–1, *i.e.,* the inner **for** loop is again executed **n** times.

This process continues until **i** reaches **m** and the loop terminates. It means the array elements have been accessed.

NOTES

For writing the elements (output operation) from the array, use the following method :

```
for(i=0; i<m; i++)
{
   for(j=0; j<n; j++)
      printf("%d",a[i][j]);
   printf("\n");
}
```

The statement **printf("\n");** after displaying the elements of a particular row, brings the control to the next line so that the elements can be displayed in matrix form.

The following program illustrates the concept of two dimensional array for finding the transpose of a square matrix and stores the result in the matrix itself.

```
/* transpose of a square matrix
  transpose is stored in the original matrix */
#include<stdio.h>
#define SIZE 5
main()
{
   int a[SIZE][SIZE],i,j,order,temp;
do
{
   clrscr();
   printf("Enter the order of square matrix <= %d\n",SIZE);
   scanf("%d",&order);
}
while(order<=0 || order>SIZE); /* get ranged value only */
printf("\nEnter the matrix of order %d * %d\n",order,order);
/* row wise reading */
for(i=0;i<order;i++)
{
   for(j=0;j<order;j++)
      scanf("%d",&a[i][j]);
}
/* echo the data */
printf("\nGiven matrix is\n\n");
for(i=0;i<order;i++)
{
   for(j=0;j<order;j++)
```

```
            printf("%6d",a[i][j]);
         printf("\n");
      }
      /* transpose */
      for(i=0;i<order-1;i++)
      {
         for(j=i+1;j<order;j++)
            {
               temp=a[i][j];
               a[i][j]=a[j][i];
               a[j][i]=temp;
            }
      }
         printf("\nTranspose of matrix\n\n");
         for(i=0;i<order;i++)
            {
               for(j=0;j<order;j++)
                  printf("%6d",a[i][j]);
               printf("\n");
            }
         getch(); /* freeze the screen until some key is pressed */
      }
```

PROGRAM 6

The output of Program 6 will be :

Enter the order of square matrix <= 5

3

Enter the matrix of order 3 * 3

1 2 3

4 5 6

7 8 9

Given matrix is

1	2	3
4	5	6
7	8	9

Transpose of matrix

1	4	7
2	5	8
3	6	9

NOTES

4.22 LIMITATIONS OF ARRAYS

The limitations or problems of arrays are given below :

1. Size of an array is fixed. If we do not know the size requirement, maximum size array is declared which may result in wastage of space. If we need more space at run time, it is not possible to extend array.
2. The elements must be homogeneous (*i.e.,* all elements are of same type).
3. The insertion and deletion operations in an array require shifting of elements which takes time.

4.23 STRING PROCESSING

A string is a collection of characters enclosed within quotes. This type of data is very important and almost all programming languages have provision to handle it. In C, a string is defined as a character array being terminated by a NULL character *i.e.,* '\0'. Each element of string is stored as one element in the array housing it. So the character arrays must be declared one character longer than the size of the string we wish to store. The last byte stores the string terminator '\0'.

4.24 STRING VARIABLE

It is actually any valid C variable name and is declared as an array of characters. *For example,* if an array **name** is used to store a 20 character string, the declaration must be :

```
char name[21];
```

The individual characters of the string are accessed using a subscript. The end of the string can be checked by comparing the character by NULL character. The NULL character is not a part of the string, it is merely used to mark the end of the string. In fact, a string not terminated by NULL character is not really a string, but merely a collection of characters. The general syntax of string declaration is :

```
char name_of_string[size];
```

where size gives number of characters.

4.24.1 Declaring and Initializing String Variables

A string can be declared as given above. In C, a string can be initialized while declaring it by specifying value of some or all of its elements. This is possible in two ways :

```
char name[8]   = {'A', 'p', 'o', 'o', 'r', 'v', 'a', '\0'};
```

or

```
char name[8]   = "Apoorva";
```

When individual array elements are moved then the terminator must be explicitly specified. But in case of string assignment, the terminator is automatically attached.

The size of the array is optional when we initialize it and in such cases it is calculated auto-matically.

The following program illustrates this concept :

```
/* count number of characters in a string and print it */
#include<stdio.h>
main()
{
   char book_name[]="MASTERING C PROGRAMS";
   int i=0;
   clrscr();
   while(book_name[i] != '\0')
      {
         printf("%c",book_name[i]);
         i++;
      }
   printf("\n\nNumber of characters in the string are %d\n",i);
   getch(); /* freeze the monitor */
}
```

PROGRAM 7

The output of Program 7 will be :

MASTERING C PROGRAMS

Number of characters in the string are 20

The **while** loop in the above program is terminated when a '\0' character is encountered in the array **book_name.**

4.24.2 Reading and Writing Strings

In C, there are two ways to read and write string data. These are :

(*i*) Using character I/O functions

(*ii*) Using string I/O functions

NOTES

(*i*) **Using Character I/O Functions**

The Input/Output functions for **char** type data are :

getch(), getche(), getchar(), putch(), putchar(), scanf() and **printf().**

All of these above mentioned functions have been discussed earlier.

(*ii*) **Using String I/O Functions**

The Input/Output functions for string type data are :

gets(), puts(), scanf() and **printf()**

All of these above mentioned functions have been discussed earlier. Let us use some of these functions through programming :

The following program reverses the entered string and prints it.

```
/* reverse a given string */

#include<stdio.h>
#define SIZE 80
main()
{
   char string[SIZE],ch;
   int i,mid,len=0;
   clrscr();
   printf("Enter a string of length <= %d\n\n",SIZE-1);
   ch=getchar();
   while(ch!='\n')
   {
      string[len]=ch;
      len++;
      ch=getchar();
   }
   string[len]='\0';   /* store string terminator */
   mid=len/2;
   /* reversal of string */
   for(i=0;i<mid;i++)
   {
      ch=string[len-1-i];
      string[len-1-i]=string[i];
```

```
        string[i]=ch;
    }
    printf("\nReversed string is : %s\n",string);
    getch(); /* freeze the monitor */
}
```

NOTES

PROGRAM 8

The output of Program 8 will be :

Enter a string of length <= 79

RAMA O RAMA

Reversed string is : AMAR O AMAR

4.25 STANDARD STRING-HANDLING FUNCTIONS

Every C compiler provides a large number of useful string functions. These functions are used for string manipulation. Some of the most useful functions are :

strcat() — Concatenates two strings

strcmp() — Compares two strings

strcpy() — Copies one string over the other

strlen() — Finds length of a string

strrev() — Reverses the string

4.25.1 String Concatenation

The process of joining two strings together is called concatenation. In C, the function for concatenation of two strings is **strcat()**. It takes two arguments, the first one is a string variable and the second can be a string variable or a string constant. The character(s) of the second string are appended at the end of the first one. The first variable must have enough space for accommodating the second string, otherwise an overflow occurs. The syntax of **strcat()** function is :

strcat(str1, str2);

Here str1 and str2 are character arrays (str2 can be a string constant also). The NULL character is removed from the end of str1 and str2 is stored from this position.

NOTES

The following program illustrates the use of **strcat()** function :

```
/* Concatenation of strings using library function */
#include<stdio.h>
#include<string.h>  /* for strcat() function */
#define S 40
#define T 80
main()
{
   char string1[T],string2[S];
   clrscr();
   printf("Enter the first string\n\n");
   gets(string1);
   printf("\nEnter the second string\n\n");
   gets(string2);
   /* concatenation using string function */
   strcat(string1,string2);
   printf("\nUsing string function\n\n");
   printf("\nResultant string is : %s\n\n",string1);
   getch(); /* freeze the monitor */
}
```

PROGRAM 9

The output of Program 9 will be :

Enter the first string

Programming in C

Enter the second string

is a fun

Using string function

Resultant string is : Programming in C is a fun

4.25.2 String Comparison

In C, **the comparison of two strings is not allowed directly.** *For example,* the following statement is not valid

```
if(str1==str2)
   printf("Strings are equal\n");
```

```
else
   printf("Strings are not equal\n");
```

The comparison can be done either using the standard library function **strcmp()** or the strings can be compared character by character. The comparison terminates on mismatch of characters or termination of any one of the strings, whichever occurs first. The syntax of **strcmp()** function is :

strcmp(str1,str2);

Here str1 and str2 are string variables or string constants. It returns a 0 when strings are identical and returns the numeric difference between the ASCII values of the first mismatch characters otherwise.

For example,

```
char str1[ ]="Vansh Dixit";
strcmp(str1,"Vansh Dixit");
strcmp(str1,"Baby");
strcmp("Enjoy",str1);
```

The values returned by the above three **strcmp()** functions are 0, – 1 and 4 respectively.

When the value returned is negative str1 comes before str2 in dictionary-order.

The following program illustrates the use of **strcmp()** function :

```
/* compare strings using library function strcmp() */
#include<stdio.h>
#include<string.h>  /* for strcmp() function */
#define S 41
main()
{
  char string1[S],string2[S];
  int diff;
  clrscr();
  printf("Enter the first string of length <=%d\n\n",S-1);
  gets(string1);
  printf("\nEnter the second string of length <=%d\n\n",S-1);
  gets(string2);
  diff=strcmp(string1,string2);
  if(diff==0)
    printf("\nStrings are identical\n");
  else if(diff<0)
    printf("\nString1 comes before string2 in dictionary order\n");
```

NOTES

```
    else
      printf("\nString1 comes after string2 in dictionary order\n");
    getch(); /* freeze the monitor */
}
```

NOTES

PROGRAM 10

The output of Program 10 will be

Enter the first string of length <= 40
Vansh Dixit
Enter the second string of length <= 40
Vansh Dixit
Strings are identical
Enter the first string of length <= 40
Vansh Dixit
Enter the second string of length <= 40
Baby
String1 comes after string2 in dictionary order
Enter the first string of length <= 40
Baby
Enter the second string of length <= 40
Vansh Dixit
String1 comes before string2 in dictionary order

4.25.3 String Copying

The process of copying one string into the other is called string copying. In C, the function for copying of one string over the other is **strcpy( ).** The syntax of strcpy() function is :

strcpy(str2,str1);

Here **str2** is the target string which stores the contents of the source string **str1.** The string **str1** may be a character array variable or a string constant. The string **str2** must have a size greater than or equal to that of string str1. So it is the responsibility of the programmer to check out for the size of the target string. The string str1 is copied into string str2 character by character and the process terminates on getting '\0' (NULL character) in the source string. Note that the contents of target string will be lost and source string remains unchanged.

The following program illustrates the use of **strcpy()** function :


```
/* copy a string using library function strcpy() */
#include<stdio.h>
#include<string.h>    /* for strcpy() function */
#define S 41
main()
{
   char string1[S],string2[S];
   clrscr();
   printf("Enter the string of length <=%d\n\n",S-1);
   gets(string1);
   /* copy the string string1 to string2 */
   strcpy(string2,string1);
   printf("\nEntered string is : %s\n\n",string1);
   printf("\nCopied string is : %s\n\n",string2);
   getch(); /* freeze the monitor */
}
```

PROGRAM 11

The output of Program 11 will be :

Enter the string of length <= 40

Enjoy your life with nature

Entered string is : Enjoy your life with nature

Copied string is : Enjoy your life with nature

4.25.4 Reversing the String

The process of exchanging the characters in the string *i.e.,* the first character exchanged with last one, second character exchanged with second last and so on, till we reach the mid position is called reversing the string. In C, the function for reversing the string is **strrev()**. The syntax of the function **strrev()** is :

strrev(string);

A string is said to be a palindrome, if it reads same from both ends. *For example,* MADAM, ARORA, NAYAN, MALAYALAM, NITIN etc.

The following program checks a string for palindrome using standard library functions.

NOTES

```
/* check a string for palindrome using library functions */

#include<stdio.h>
#include<string.h>
#define SIZE 80
main()
{
  char string[SIZE],stringdup[SIZE];
  int diff;
  clrscr();
  printf("Enter a string of length <= %d\n\n",SIZE-1);
  gets(string);
  strcpy(stringdup,string); /* copy string in duplicate string */
  strrev(stringdup);   /* reverse the duplicate string */
  diff=strcmp(string,stringdup);  /* compare the two strings */
  if(diff==0)  /* if strings are identical */
    printf("\nString is a palindrome\n");
  else
    printf("\nString is not a palindrome\n");
  getch();
}
```

PROGRAM 12

The output of Program 12 will be :

Enter a string of length <= 79

malayalam

String is a palindrome

Enter a string of length <= 79

jaggu

String is not a palindrome

4.26 DATA FILES

A file is a bunch of bytes stored on some storage device like magnetic disk or tape etc. Most of the application programs process large volume of data which is permanently stored in files. We can write programs that can read data from file(s) and write data to file(s). Compilers read source code files and provide executable files. Database programs and word processors also work with files.

Data transfer is generally one or both of the two types given below :

(*i*) Transfer between console unit and the program.

(*ii*) Transfer between the program and a file on disk or tape.

NOTES

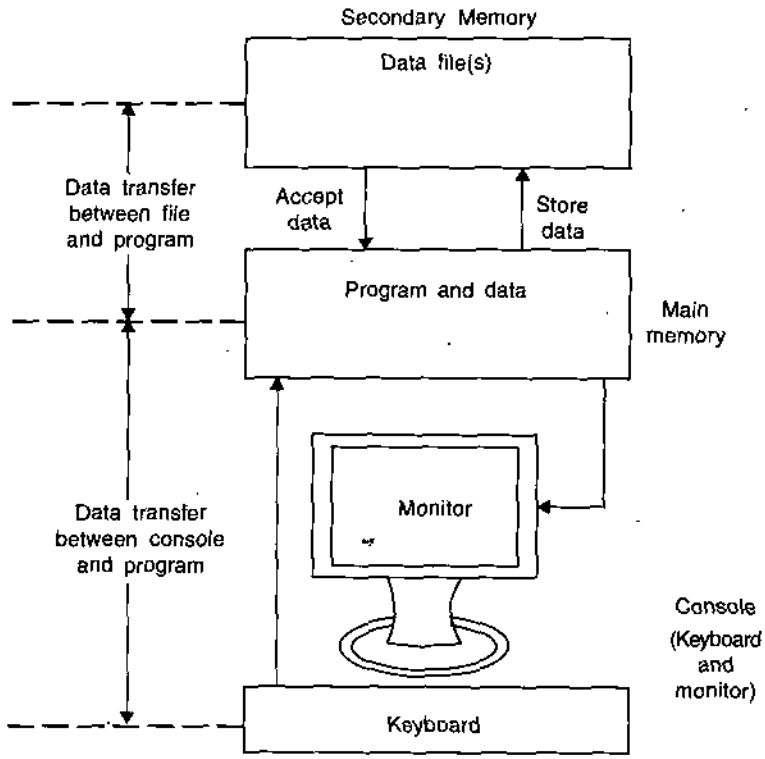


Fig. 9. *Data communication between various units.*

So far we have used the technique of data communication between the console unit and the program. Now we are going to describe the storage and retrieval of data from the files which overcomes the handling of large volume of data from the keyboard at run time. Some of the problems which may occur when data is entered from the keyboard are :

(*i*) A lot of time is taken for data entry.

(*ii*) Entire data would be re-entered if a mistake is made in data.

(*iii*) If the same data is to be processed later on, it would be entered afresh *i.e.,* for every run, we require the same data entry.

So for fast data processing, enter the data in computer's main memory, correct it (if required) and store it in a disk file. Now the data from the file can be read directly at a very high speed.

In some situations the output of a program may become the input of another program(s). So in such cases it is better to store the output of the program in a disk file which can be read by other program(s) later on.

NOTES

4.27 FILE HANDLING IN C

The treatment of files in C is very simple, unlike that of other programming languages which have special built-in and often rigid file structures and file-handling routines. C treats file input-output in almost the same way as input-output from/to the console, and provides file input-output functions, very similar to those for input-output from/to the console.

Unlike other programming languages C does not distinguish between sequential and random access (direct access) data files.

The **Turbo C** provides two different ways of file processing. These are

(*i*) Standard Input/Output (stream I/O or high-level I/O)

(*ii*) System-oriented Input/Output (low-level I/O).

We can perform most of the tasks by either of these, still there are many important differences between these two. The major differences are given below :

The **Standard I/O** is very simple and most commonly used way of performing file I/O in C language. It provides a vide variety of commands. The I/O operations, such as buffering, data conversions etc. take place automatically. It will be the only system for I/O if the C version has only one system for I/O.

The data is written as individual **characters** or as **strings** or as **formatted data.** Library functions are available for transfer of information. **Unformatted** data files, organize data into blocks having contiguous bytes of information for more complex data structures such as structures and arrays. Library functions are available that can transfer entire structures or arrays to or from data files.

The **System-oriented I/O** reads and writes the data to/from files the same way as MS-DOS. The data **cannot** be written as individual **characters,** or as **strings,** or as **formatted data.** Using this approach the data can be written as a buffer full of bytes. The programmer must set up the buffer for the data, place the appropriate data into it before writing, and take it out from the buffer after reading. It is harder to program than standard I/O but efficient both in terms of operation and the amount of memory used by the program.

4.28 OPENING AND CLOSING A DATA FILE

When working with a standard data file (stream oriented data file), first of all a **buffer area** (the portion of main memory which can be directly accessed

by the I/O devices) is established, where the information is kept temporarily during transfer between the computer's memory and the data file. The buffer area helps in fast read/write operation from/to the data file. The buffer area is associated by writing.

NOTES

```
FILE  *fptr;
```

Here, FILE (use uppercase letters only) is a special structure type defined within a system **include** file, namely **stdio.h** in DOS. It is not a good practice to use the members of the FILE structure shown below :

```
typedef struct {
   short          level;          /* fill/empty level of buffer */
   unsigned       flags;          /* File status flags */
   char           fd;             /* File descriptor (handle) */
   unsigned char  hold;           /* Ungetc char if no buffer */
   short          bsize;          /* Buffer size */
   unsigned char  *buffer, *curp; /* Data transfer buffer,
                                          Currentactive pointer */
   unsigned       istemp;         /* Temporary file indicator */
   short          token;          /* Used for validity checking */
} FILE; /* FILE object */
```

fptr is a pointer variable that indicates the beginning of the buffer area. It is also known as a **stream pointer** or **stream.**

A data file must be **opened** before it can be created or processed. The filename is associated with the buffer area or the stream. The mode of file utilization *i.e.*, read only, write only or read/write both is also specified while opening the data file.

The syntax of library function **fopen( )** used to open a file is given below :

```
fptr = fopen ("filename", "mode");
```

Here, **filename** represents the name of the data file and **mode** specifies the purpose of opening the file.

The **filename** must be in accordance with the rules for naming files, as per the operating system in use. The valid modes are given in Table 5.

NOTES

Table 5. File opening modes and their purpose

Mode	*Purpose*
r	Open for reading only. The file must already exist
w	Open for writing only. If the file already exists, its contents will be destroyed. If it does not exist, it will be created
a	Open for appending (*i.e.*, for adding data to the end of the existing file). If it does not exist, it will be created
r+	Open for both reading and writing. The file must already exist
w+	Open for both reading and writing. If the file exists, its contents are overwritten
a+	Open for both reading and appending. If the file does not exist, it will be created

The **fopen( )** function returns a pointer to the beginning of the buffer area associated with the file (if possible); otherwise a NULL value is returned which is defined in **stdio.h.** The pointer to the structure type FILE is assigned to **fptr.** While opening the file in text mode we can use either **"r"** or **"rt",** but since text mode is the default mode we generally drop the **'t'** from it.

Note : For opening a file in binary mode each of the above shown modes can be ***suffixed*** *with letter 'b'.*

When the processing is over (reading/writing) in a file, it must be closed. The library function **fclose( )** performs this task. The syntax of **fclose( )** function is given below :

```
fclose(fptr);
```

Here **fptr** is the file pointer associated with the file to be closed.

If we do not close file(s) explicitly using the **fclose( )** function, most of C compilers will automatically close the data file(s) at the end of program execution.

Closing a file explicitly performs the following operations :

(*i*) Data from buffer is transferred to the file. It may be noted that the buffer used in the standard I/O is invisible to the programmer.

(*ii*) The area occupied by the file (area consisting of FILE structure and the buffer itself) is made free so that it may be used by other files.

4.29 TROUBLE IN FILE OPENING

The fopen( ) function for opening file for read/write operations may fail due to anyone of the following reasons :

(*i*) A file for reading may not be present on the disk.

(*ii*) Insufficient disk space for opening a file for writing.

(*iii*) Write protected disk does not allow storage of data on it.

(*iv*) Dealing with a corrupt file.

The following program statements will provide you a clear idea for checking the successful opening when we open a file for reading.

NOTES

```
#include<stdio.h>
main()
{
    FILE *fptr;
    fptr = fopen("TEXT.DAT", "r"); /* open file for reading*/
    if(!fptr)
    {
        printf("\nCan't open file for reading\n");
        exit();
    }
        ....
        ....
}
```

4.30 POINTERS

Pointers are very useful and important feature of C language. A beginner may find it a little confusing to start with. But once the concept of pointers is clear the user can write complex code with great ease, using this powerful tool, making C an excellent language.

A pointer is a variable which holds a memory address which is the location of some other variable in memory. As a pointer is a variable, its value is also stored in another memory location. Any variable declared in a C program has two components :

(*i*) Address of the variable

(*ii*) Value stored in the variable.

For example,

```
int x = 547;
```

The above declaration tells the C compiler for :

(*a*) Reservation of space in memory for storing the value.

(*b*) Associating the name **x** with this memory location.

(*c*) Storing the value 547 at this location.

NOTES

It can be represented with Figure 10 :

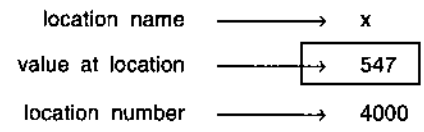


Fig. 10. *Representation of a variable.*

Here, the address 4000 is assumed one, it may be some other address also. Remember that *the address of a variable is the address of the first byte occupied by that variable in memory.* Also the values are stored in binary form inside the memory.

Let the address of **x** be assigned to a variable **ptr** having address 4036. Since the value of **ptr** is the address of the variable **x**, the value of x can be accessed using the value of **ptr** or in other words we can say that the variable **ptr** 'points to' the variable x so it is called a 'pointer'. The above concept can be represented as given shown in Figure 11 :

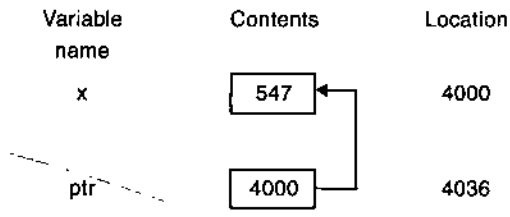


Fig. 11. *Illustration of a pointer as a variable.*

Pointers are frequently used in C language, as they offer a number of benefits to the users. They include :

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function parameters.
3. Pointers permit references to functions and thereby allowing passing of functions as parameters to other functions.
4. For saving the storage space by using the pointer arrays for character strings.
5. Pointers allow C to support dynamic memory management (*i.e.,* allocation/ deallocation of memory at run time).
6. Dynamic data structures such as *structures, linked lists, stacks, queues* and *trees* can be easily manipulated using pointers.
7. For reducing the size and complexity of programs.
8. For fast execution of programs.

4.31 DECLARING AND INITIALIZING A POINTER

NOTES

For storing the address of a variable, we must declare the appropriate pointer variable for it. The syntax for a pointer declaration is given below :

type *ptr_name;

Here, type specifies the type of the variable that is to be pointed to by the pointer **ptr_name.**

* represents the variable **ptr_name** as a pointer variable and it needs a memory location too.

For example,

```
int *ptr;    /* declaration of an integer pointer */
int x = 547;
ptr = &x;    /* ptr stores the address of x */
```

The actual address of a variable in memory is not known to us. So the **&** (address operator) is needed for returning the address of the variable following it *i.e.,* a variable name is followed after **&**. Similarly, the following statements

```
float *fptr, fvalue;
char *cptr, ch;
fvalue = 40.5;
ch = 'A';
fptr = &fvalue;
cptr = &ch;
```

show the pointer initialization, by first declaring the pointer variables and then making the pointer variables to point to their respective data type variables. *A pointer variable contains garbage until it is initialized.* We should not use a pointer before initializing it.

Remember that the definition for a pointer variable allocates memory only for the pointer variable, not for the variable to which it is pointing.

> ***Note :*** *The data type of the pointer must be same as the data type of the variable to which it points.*

In C, the assignment of an absolute address is not allowed to a pointer variable. *For example,*

```
int *iptr;
iptr = 258; /* invalid assignment */
```

NOTES

We can initialize a pointer variable while declaring it, as given below :

```
int num = 85;
int *iptr = &num; /* initialization while declaration */
```

Note that variable **num** is first declared and then its address stored in pointer variable **iptr.**

The following program prints the different types of variables and their addresses. As the memory addresses are unsigned integers so we can use %u or %lu format for printing the address values in integer form or %x format for printing the address values in hexadecimal form.

```
/* illustration of address of (&) operator for getting address */

#include<stdio.h>
main()
{
  char ch;
  int x;
  float y;
  x=336;
  y=12.5;
  ch='J';   /* ASCII value of 'J' gets stored in ch */
  clrscr();
  printf("The addresses are shown in decimal form\n\n");
  printf("You may get some other addresses on your system\n\n");
  printf("\nValue of ch = %c",ch);
  printf("\nAddress of ch is %u", &ch);
  printf("\n\nValue of x = %d",x);
  printf("\nAddress of x is %u",&x);
  printf("\n\nValue of y = %.2f",y);
  printf("\nAddress of y is %u",&y);
  getch();/* freeze the monitor */
}
```

PROGRAM 13

The output of Program 13 will be :

The addresses are shown in decimal form

```
You may get some other addresses on your system
Value of ch = J
Address of ch is 65489
Value of x = 336
Address of x is 65490
Value of y = 12.50
Address of y is 65492
```

4.32 ACCESSING A VARIABLE USING POINTER

In C, the value of a variable (once its address has been assigned to a pointer variable) can be accessed using the unary operator * (asterisk) known as the indirection operator.

The operator * is followed by an address and it can be kept in mind as 'value at address'. *For example,*

```
int value, num, *iptr;
value = 2007;
iptr = &value;
num = *iptr;
```

after the execution of the above statements num and value both have 2007.

In C, the pointers and addresses are utilized by means of symbolic names. A statement like *336 will not work at all. The following program prints the value of variables using the indirection operator ' * ' alongwith the addresses.

```
/* illustration of indirection operator (*) for printing values */

#include<stdio.h>
main()
{
  char ch, *cptr;
  int x, *iptr;
  float y, *fptr;
  x=336;
  y=12.5;
  ch='J';    /* ASCII value of 'J' gets stored in ch */
  cptr=&ch;
  iptr=&x;
  fptr=&y;
  clrscr();
  printf("The addresses are shown in Hexadecimal form\n\n");
  printf("You may get some other addresses on your system\n\n");
  printf("\nValue of ch = %c",*cptr);
  printf("\nAddress of ch is %x",cptr);
  printf("\n\nValue of x = %d",*iptr);
```

NOTES

```
    printf("\nAddress of x is %x",iptr);
    printf("\n\nValue of y = %.2f",*fptr);
    printf("\nAddress of y is %x",fptr);
    getch(); /* freeze the monitor */
}
```

PROGRAM 14

The output of Program 14 will be :
The addresses are shown in Hexadecimal form
You may get some other addresses on your system
Value of ch = J
Address of ch is ffcb
Value of x = 336
Address of ch is ffcc
Value of y = 12.50
Address of y is ffce

The operation of writing the value or manipulating it by using * as a prefix with a pointer variable or pointer expression is called **dereferencing** pointers. In C, a pointer stores the address of another variable which in turn can store address of another variable and so on.

4.33 void POINTERS

We know that pointers are used for pointing to different data types. A **float** pointer points to float variables, **int** type pointer points to integer variables, a **char** type pointer points to character variables. In C, there is a general purpose pointer that can point to any data type and is known as **void** pointer. The **void** pointer is a **generic pointer** that can represent any pointer type. The syntax of its declaration is given below :

```
void  *vptr;   /* pointer to void */
```

In C, pointers to void cannot be directly dereferenced like other pointer variables by using *, the indirection operator. A suitable typecast is must prior to dereferencing a pointer to void as given below :

```
        ┌──────────── dereferencing operator
        ↓
*((type *)vptr)
        ↑
        └──────────── type cast
```

Here, type refers to any valid C data type.

4.34 POINTER EXPRESSIONS

In C, the pointer variables can be used in expression like ordinary variables. Never use /* together in an expression as it is treated as the beginning of a

C comment. *We can add, subtract integers from pointers, subtract one pointer from another. For example,*

NOTES

```
int *ptr1,*ptr2,x,y;
x = ptr1 + 5;
y = ptr2 - 3;
```

The above statements are valid in C. Also the statements given below will work :

```
--ptr1;
ptr2++;
x = *ptr2 / *ptr1;
y += *ptr1;
```

We can compare pointers using the relational operators of same types. *For example,* the statements given below are valid.

```
if(ptr1 < ptr2)
  statement;
if(ptr1 == ptr2)
  statement;
if(ptr1 != ptr2)
  statement;
```

Above type of comparisons are quite useful while dealing with arrays and strings.

But, remember that *a comparison of pointers that belong to separate and unrelated variables is meaningless.*

> ***Note :*** *Pointers cannot be used in addition, multiplication or division individually.*

4.35 POINTERS AND FUNCTIONS

As stated earlier, a function groups a number of program statements into a single unit and this unit can be called from other parts of the program. We know that a function can be called in one of the following ways :

(*i*) Call by value.

(*ii*) Call by reference.

NOTES

These methods have been discussed in Chapter 8 on Functions. Concept of pointers is used in the call by reference method :

In call by reference method, the addresses of the actual arguments in the calling function are copied into the formal arguments of the called function. So the called function refers to the original values by the address it accepts. Using pointers we can return more than one value to the calling function, which is not possible ordinarily by using a **return** statement. A return statement can return only a single value.

For example, the following program finds the area and perimeter of a circle using pointers :

```
/* area and perimeter of circle - using pointers */

#include<stdio.h>
#define PI 3.14159
main()
{
   void areaperi(float,float *,float *); /* function prototype */
   float radius,area,perimeter;
   clrscr();
   printf("Enter radius of circle\n\n");
   scanf("%f",&radius);
   /* echo the data */
   printf("\nRadius = %.2f\n",radius);
   areaperi(radius,&area,&perimeter); /* function call */
   printf("\nArea = %.2f square units\n",area);
   printf("\nPerimeter = %.2f units\n",perimeter);
}

/* function definition areaperi() */

void areaperi(float r,float *a,float *p)
{
   *a=PI*r*r;
   *p=2*PI*r;
}
```

PROGRAM 15

The output of Program 15 will be :

```
Enter radius of circle
5
Radius = 5.00
Area = 78.54 square units
Perimeter = 31.42 units
```

4.36 POINTERS AND ONE DIMENSIONAL ARRAYS

NOTES

We know that the name of the array holds the address of the first element (index 0) in that array in C. *For example,*

```
char name[21];
```

Here, **name** holds the address of name[0] and it is a constant pointer to the first element. So we can make the conclusion that the name of an array is actually a pointer.

The following program illustrates the manipulation of an array using a pointer :

```
/* manipulation of an array using a pointer */

#include<stdio.h>
main()
{
  char arr[]="Welcome to the world of C programming";
  char *cptr;
  clrscr();
  printf("\nMessage for all is : %s\n",arr);
  cptr=arr;     /* cptr and arr point to same location */
  printf("\nMessage for all is :%s\n",cptr);
  getch();  /* freeze the monitor */
}
```

PROGRAM 16

The output of Program 16 will be :

Message for all is : Welcome to the world of C programming

Message for all is : Welcome to the world of C programming

Here, **arr** as a pointer is constant, that is, it points to the first element of the array of characters and we cannot change its value but we can make **cptr** point to any location in memory as it is dynamic.

The following program illustrates the traversing of an array using a pointer :

```
/* traversing an array using a pointer */

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

```
{
  char arr[]="Welcome to the world of C programming";
  char *cptr;
  clrscr();
  cptr=arr;     /* cptr initialized with array address */
  printf("\nMessage for all is\n\n")
  for(;*cptr!= '\0';cptr++)
    printf("%c",*cptr);
  getch(); /* freeze the monitor */
}
```

PROGRAM 17

The output of Program 17 will be :

Message for all is

Welcome to the world of C programming

4.37 ARRAY OF POINTERS

In C, we may have an array of pointers also. For declaring an array storing 3 integer pointers, we may declare :

```
int *num[3];      /* array of 3 integer pointers */
```

This can be represented as given below :

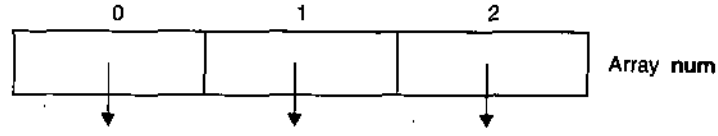


Fig. 12. *Array of pointers.*

The following program illustrates this concept :

```
/* print array elements being pointed to by an array of pointers */
#include<stdio.h>
main()
{
  int *num[3];/* array of 3 integer pointers */
  int i,a=50,b=60,c=70;
  clrscr();
  /* array initialization */
  num[0]=&a;
  num[1]=&b;
```

```
num[2]=&c;
/* echo the data */
for(i=0;i<3;i++)
  printf("\nThe pointer num[%d] points to the value %d\n",i,*num[i]);
printf("\n\nAddresses printed below are in Hexadecimal form\n");
printf("\nYou may have different addresses on your system\n");
/* addresses are in Hexadecimal form */
printf("\n\nBase address of array of pointers \"num\" is %x\n", num);
/* name of an array denotes its base address */
for(i=0;i<3;i++)
  printf("\nThe address contained in num[%d] is %x",i,num[i]);
getch(); /* freeze the monitor */
}
```

PROGRAM 18

The output of Program 18 will be :

```
The pointer num[0] points to value 50
The pointer num[1] points to value 60
The pointer num[2] points to value 70
Addresses printed below are in Hexadecimal form
You may have different addresses on your system
Base address of array of pointers "num" is ffcc
The address contained in num[0] is ffd2
The address contained in num[1] is ffd4
The address contained in num[2] is ffd6
```

4.38 POINTERS AND STRINGS

As mentioned earlier, a string is an array of characters terminated by a NULL ('\0'). An array of **char** pointers is generally better than two dimensional array of characters due to the following reasons :

(*i*) Better utilization of memory (less number of bytes taken by an array of pointers).

(*ii*) Manipulation of strings is easy using an array of pointers.

One important use of pointers is in handling of a table of strings. Generally the individual strings are of varying lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. *For example,*

```
char *name[3] = {"Sri Lanka", "Pakistan", "India"};
```

NOTES

declares **name** to be an *array of three pointers* to characters, each pointer pointing to a particular name as shown below :

name[0] ⟶ Sri Lanka

name[1] ⟶ Pakistan

name[2] ⟶ India

This declaration allocates only 25 bytes, sufficient to hold all the three strings as shown in Figure 13 :

S	r	i		L	a	n	k	a	\0
P	a	k	i	s	t	a	n	\0	
I	n	d	i	a	\0				

Fig. 13. *Array of pointers to strings.*

These strings can be printed as

```
for(i=0;i<3;i++)
  printf("%s\n",name[i]);
```

The character arrays with the rows of varying length are called *'ragged arrays'* and are better handled by pointers.

You can easily manipulate strings using an array of pointers.

4.39 PROBLEMS WITH POINTERS

Common bugs related to pointers and memory management are given below :

(*i*) **Problem of dangling pointers.** Such a problem occurs when the programmer fails to initialize a pointer with a valid address. Such an un-initialized pointer, is known as *dangling pointer.* Such a pointer can cause programs to crash unceremoniously. Therefore, care must be taken to initialize pointers with valid address.

(*ii*) **Problem of null pointer assignment.** It may happen that the pointer points to address 0, which is called NULL. For example, if the pointer is a global variable or local static variable. In such situations if the pointer is not assigned some valid address then the computer system will display a message *"Null pointer assignment"* on termination of program.

NOTES

(*iii*) **Problem of memory leaks.** Another common problem with pointers is that of *memory leak*. Memory leak is a situation where the programmer forgets to release (deallocate) the memory allocated at execution (run) time in a module. A pointer allocated memory at some stage when goes out of scope and there is no way to reach that memory block. Even use of **goto** statement might cause unconditional jump and we may forget to deallocate the memory allocated earlier. Therefore, care must be taken to release (deallocate) the allocated memory so that it may be reused.

(*iv*) **Problem of allocation failures.** We may face problem of *allocation failures* using pointers. An *allocation failure* is a situation when the program through *malloc( ), calloc( ),* or *realloc( )* function request for a block of memory, and the operating system could not fulfill the request of desired memory which may not be available in the free storage pool. So, proper measures should be applied in such cases by the programmer.

4.40 SUMMARY

- C is a general-purpose, structured programming language.
- C language is *case sensitive i.e.,* uppercase and lowercase characters are not equivalent.
- C is a free form language.
- Identifiers are names given to various program elements, such as variables, functions and arrays.
- C supports several different types of data, each of which may be represented differently within the computer's memory. The basic data types are *int, char, float* and *double.*
- C has four basic types of constants, these are *integer constants, floating-point constants, character constants* and *string constants.*
- An escape sequence always begins with a backward slash (\) and is followed by one or more special characters. For example, **\n** (newline).
- Operators are used to form expressions.
- The data items that operators act upon are called *operands.*
- There are five arithmetic operators in C (+, –, *, / and %).
- The operators that act upon a single operand to produce a new value are known as *unary operators.* Commonly used unary operators are – (*unary minus*), ++ (*increment operator*), – – (*decrement operator*) and *sizeof.*

NOTES

- The *sizeof* operator returns the size of its operand in bytes and always precedes its operand.
- A *cast* is considered to be a unary operator and a reference to the cast operator is written as (*type*). It is also called a *type cast.*
- A *function* is a self-contained program segment that performs some specific, well-defined task.
- Every C program has one or more functions, one of these must be called **main( )** in which the program execution begins. Additional functions (if any) are subordinate to *main( )*, and perhaps to one another.
- Function declaration specifies what is the return type of the function and the types of arguments it accepts.
- Function definition defines the body of the function.
- Pointers can be used to make a function return more than one value simultaneously.
- Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.
- The use of recursion is not necessarily the best way to solve a problem, even though the problem definition may be recursive in nature. A non-recursive implementation may be more efficient in terms of memory utilization and execution speed.
- An array is like an ordinary variable except that it can store multiple elements of same type.
- While declaring an array, we need to specify three things, namely, *name, type* and *size.*
- Always remember that array subscripts begin at 0 (not 1) and end at size –1.
- Array elements can be passed to a function either by value or by reference.
- An array is always passed by reference to a function.
- C permits arrays of three or more dimensions.
- During initialization of multidimensional arrays, it is an error to omit the array size of any dimension other than first.
- A string is nothing but an array of characters terminated by null character ('/0').
- Being an array, all the characters of a string are stored in contiguous memory locations.
- Some of the useful standard library functions for string manipulation are, **strlen( ), strcpy( ), strcat( ), strcmp( )** and **strrev( ).**
- The comparison of two strings is not allowed directly.

- When using string functions for copying and concatenating strings be sure that the target string is capable of storing the resulting string. Otherwise memory overwriting may occur.
- A *pointer* is a variable that represents the location (rather than the value) of a data item, such as a variable or an array element.
- Do not store the address of a variable of one type into a pointer variable of another type.
- The value of a variable cannot be assigned to a pointer variable.
- Before initialization a pointer variable contains garbage. Therefore, we must not use a pointer variable before it is assigned, the address of a variable.
- Pointers are closely associated with arrays and therefore provide an alternative way to access individual array elements.
- An array name is actually a pointer to the array, *i.e.,* the array name represents the address of the first element in the array.
- Ordinary variables *cannot* be assigned arbitrary addresses (*i.e.,* an expression such as &num cannot appear on the lefthand side of an assignment statement).

4.41 TEST YOURSELF

1. When was 'C' developed ?
2. Who has developed 'C' language ?
3. What does the 'C' character set consists of ?
4. What is data type ? Describe the fundamental data types in 'C' language.
5. Write a short note on the following :
   (*i*) User defined type declaration
   (*ii*) Enumerated data type
6. What is an operator ? Describe various types of operators available in C language. Also summarize the precedence of the arithmetic operators.
7. Explain various unary operators in C.
8. Explain various assignment operators in C.
9. What are the different categories of functions in C ? Give examples.
10. Clearly differentiate between function prototype, function definition and function call in C.
11. What is recursion ? While writing any recursive function what thing(s) must be taken care of ?
12. What conditions must be satisfied by the entire elements of any given array ?
13. What are subscripts ? How are they written ? What restrictions apply to the values that can be assigned to subscripts ?

NOTES

14. How are two dimensional arrays defined ? Compare with the manner in which one dimensional arrays are defined. Also write about how initial values can be specified for each type of array.
15. Write a C program to find transpose of a matrix of order m × n.
16. Write a C program to print a matrix alongwith row and column sum.
17. Write a C program to compute the sum of elements on both diagonals of a square matrix. *For example,*

$$\begin{bmatrix} 5 & 7 & 3 \\ 9 & 4 & 1 \\ 8 & 0 & -2 \end{bmatrix}$$

will result into sum as 18 (5 + 4 - 2 + 3 + 8 = 18). Take care that in case of a square matrix of odd order the common diagonal element must be added once only.

18. What are the limitations of arrays ?
19. How is string is stored in C ?
20. Explain the following string handling functions :
    (*i*) **strcat()** (*ii*) **strcpy()** (*iii*) **strcmp()** (*iv*) **strlen()** (*v*) **strrev()**.
21. What are pointers ? Why are they needed ? Explain with an example.
22. What are the benefits of using pointers ?
23. Write in brief about **void** pointers.
24. Write a C program to reverse an array using pointers.
25. Write a C program to implement binary search recursively on a sorted array given in ascending order using pointers.
26. Give an example of array of pointers and pointers to pointers.

❐❐❐

APPENDIX

NOTES

TESTING AND DEBUGGING OF PROGRAM

There are various stages of software development and the methods that can be applied at each stage, once there is awareness of the problem. It is the approach to increasing programmer productivity and ensuring that the programmers are as correct as possible at the end of program development cycle. Coding is mostly confused with software development. Coding is writing programs in a language that is comprehensible to a computer. In fact, coding is usually a small part of software development.

Software development can be divided into several stages. There are :

1. Problem definition
2. Program design
3. Coding
4. Debugging
5. Testing
6. Documentation
7. Maintenance
8. Extension and redesign.

The programmer works on several stages at the same time—coding, debugging, testing and documentation are often concurrent activities. Let us discuss debugging and testing in detail.

Debugging

This stage is the discovery and correction of programming errors. Few programs run correctly the first time, so **debugging** is an important and time-consuming stage of software development. Programming theorists often refer to program debugging and testing as verification and validation, respectively. Verification ensures that the program does what the programmer intends to do. Validation ensures that the program produces the correct results for a set of test data. There is no clear demarcation line between these stages. The debugging of microprocessor programs is generally quite difficult because of the inability to observe register contents directly, the primitive debugging aids, the close interaction between hardware and software, the frequent dependence of programs on precise timing, and the difficulty of obtaining adequate data for real-time applications. The tools that can be used to debug programs with brief descriptions are given below :

NOTES

1. **Simulators.** A simulator is a computer program that simulates the execution of programs on another computer.
2. **Logic analysers.** A logic analyser is a test instrument that is the digital bus-oriented version of the oscilloscope. It detects the states of digital signals during each clock cycle and stores them in the memory. It then displays the information on a CRT, much as an oscilloscope does.
3. **Breakpoints.** A breakpoint is a place in a program at which execution can be halted, in order to examine the current contents of registers, memory locations and I/O ports.
4. **Trace routines.** A trace is a program that prints information concerning the status of the processor at specified intervals. Most simulator programs and some microcomputer development systems have trace facilities.
5. **Memory dumps.** A memory dump is a listing of the current contents of a section of the memory. Most simulator programs, microcomputer development systems and monitors can produce memory dumps.
6. **Software interrupts.** The software interrupt or trap instruction is frequently used for debugging purposes. The instruction usually saves the current value of the program counter and then branches to a specified memory location. That memory location can be the starting point of a debugging program that lists or displays status information—breakpoints may be inserted with trap instructions.

Each program has its own unique errors. Nevertheless, some errors are sufficiently common to deserve mention. They include :

1. Failure to initialise variables, particularly counters and pointers. Registers, flags and memory locations should not be assumed to contain zero at the start of the program.
2. Failure to handle trivial cases, such as an array or table with no elements or only one element.
3. Inverting conditions, such as jumping on zero instead of on not zero.
4. Reversing the order of operands, such as moving A to B when A to B was meant.
5. Jumping on conditions that have been changed, since they were set to the desired values.
6. Failure to handle fall-through conditions, such as an entry that is never found in a table or a condition that is never met. Such a failure can cause an endless loop.
7. Confusing addresses and values. Memory location 1000 does not necessarily contain the number 1000.
8. Confusing numbers and characters. ASCII zero or EBCDIC zero is not the same as number zero.

9. Ignoring the direction of noncumulative operations.
10. Ignoring overflow when doing signed arithmetic.

Testing

NOTES

This stage is the validation of the program. Testing ensures that the program performs correctly the required tasks. Program testing and program debugging are closely related. Testing is essentially a later stage of debugging in which the program is validated by trying it on a suitable set of test cases. Some of the test cases will certainly be the ones used in debugging—the all-zeros case, the various special cases and other obvious cases that must be checked.

Program testing is, however, more than a simple matter of exercising the program a few times. Exhaustive testing of all possible cases is the best alternative, but this process is usually impractical. Formal validation methods exist, but are only applicable to very simple programs. Thus, program testing requires a choice of test cases. The situation is further complicated by the fact that many microcomputer programs depend on realtime inputs that are difficult to control or simulate; the microprocessor must interact in a precise manner with a large and complex system. How can the necessary data be generated and presented to the microcomputer ? Several tools are available to help with this task. Clearly, the debugging tools mentioned earlier will be useful.

Among the rules that can aid in program testing are the following :

1. **Make the test plan part of the program design.** Testing should be one of the factors in the problem definition, program design and coding stages.
2. **Check all trivial and special cases.** Often the simplest can lead to the most annoying and mysterious errors.
3. **Select test data on a random basis.** Doing so will eliminate any inadvertent bias caused by the programmer selecting test data. Random number tables are widely available and most computers have random number generators.
4. **Plan and document software testing just like hardware testing.** Obviously, testing can never prove that no errors exist; so good software design, like good hardware design, is an essential part of the testing process.
5. **Use the maximum and minimum values of all variables as test data.** Extreme values are often the source of special errors.
6. **Use statistical methods in planning and evaluating complex tasks.** Methods are available for selecting data and evaluating the significance of results. Optimisation of techniques may suggest good choices for system parameters and efficient sets of test data.

There are two goals in preparing a test plan. First, a properly detailed test plan demonstrates that the program specifications are understood completely.

NOTES

Second, the test plan is used during program testing to prove the correctness of the program. During this step, a general approach to the testing of the program is prepared and documented, indicating the number of tests needed and the purpose of each test. In addition, all input test data is defined in detail, and all expected results, including reporting, are determined and documented.

Program testing includes physically running the tests specified in the test plan as well as correcting errors found in the code during the testing. This step is complete when the test results have been reviewed and approved by the project leader.

To build a more consistent history of the time expended to complete each step, the life of the coding step can be defined. The time used in making coding changes needed to correct errors that were detected during the test step, is recorded as test time. This eliminates the question of where to record those hours and facilitates consistent recording of time throughout the programming staff. Similarly, it is not allowed to record time in any step that has been previously completed. However, if a flow is detected, that forces redesign of all or part of a program, this time can be reported in the earlier phases.

In order to test a program properly, test plan development has to be undertaken at some point in the program development process. It has been experienced that performing this step before any logic is constructed ensures that a usable and exhaustive plan for proving the correctness of a program is available when the coding of the program has been completed. This will most likely provide a more sensible approach to testing all facets of a program with as few test runs as possible. If the development of a test plan is postponed until after the program is coded, it will tend to test only those parts of the program which the programmer has become more concerned with or interested in.

❑❑❑