PROGRAMMING IN 'C'

MODULE 2.3

Centre for Electronics Design & Technology of India

A Scientific Society under Department of Electronics,
Govt. of India, New Delhi

Published by CFS Documentation Cell
Centre for Electronics Design and Technology of India
An Autonomous Scientific Society under Department of Electronics,
Govt. of India,
New Delhi.

First Edition: 1999

CEDTI/CFS/99/6/2.3/R1

FOREWORD

The information technology and telecom sectors have suddenly opened up avenues, which require a very large specially trained manpower. These sectors are highly dynamic and need training and re-training of manpower at a rapid rate. The growing gap of requirement of the industry and its fulfillment has created a challenging situation before manpower training institutes of the country. To meet this challenge most effectively, Centre for Electronics Design and Technology of India (CEDTI) has launched its nation-wide franchising scheme.

Centre for Electronics Design and Technology of India (CEDTI) is an Autonomous Scientific Society under the Govt. of India, Department of Electronics with its Headquarters at New Delhi. It operates seven centres located at Aurangabad, Calicut, Gorakhpur, Imphal, Mohali, Jammu and Tezpur. The scheme will be implemented and coordinated by these centres.

The scheme endeavours to promote high quality computer and information technology education in the country at an affordable cost while ensuring uniform standards in order to build a national resource of trained manpower. Low course fees will make this education available to people in relatively small, semi urban and rural areas. State-of-the-art training will be provided keeping in view the existing and emerging needs of the industrial and Govt. sectors. The examinations will be conducted by CEDTI and certificates will also be awarded by CEDTI. The scheme will be operated through all the seven centres of CEDTI.

The CEDTI functions under the overall control and guidance of the Governing Council with Secretary, Department of Electronics as its Chairman. The members of the council are drawn from scientific, government and industrial sectors. The Centres have separate executive committees headed by Director General, CEDTI. The members of these committees are from academic/professional institutes, state governments, industry and department of electronics.

CEDTI is a quality conscious organisation and has taken steps to formally get recognition of the quality and standards in various activities. CEDTI, Mohali was granted the prestigious ISO 9002 certificate in 1997. The other centres have taken steps to obtain the certification as early as possible. This quality consciousness will assist CEDTI in globalizing some of its activities. In keeping with its philosophy of ‘Quality in every Activity’, CEDTI will endeavour to impart state of the art – computer and IT training through its franchising scheme.

The thrust of the Software Courses is to train the students at various levels to carry out the Management Information System functions of a medium sized establishment, manufacture Software for domestic and export use, make multimedia presentations for management and effectively produce various manufacturing and architectural designs.

The thrust of the Hardware Courses at Technician and Telecommunication Equipment Maintenance Course levels is to train the students to diagnose the faults and carry out repairs at card level in computers, instruments, EPABX, Fax etc. and other office equipment. At Engineer and Network Engineer levels the thrust is to train them as System Engineers to install and supervise the Window NT, Netware and Unix Networking Systems and repair Microcontrollers / Microprocessor based electronic applications.

An Advisory Committee comprising eminent and expert personalities from the Information Technology field have been constituted to advise CEDTI on introduction of new courses and revising the syllabus of existing courses to meet the changing IT needs of the trade, industry and service sectors. The ultimate objective is to provide industry-specific quality education in modular form to supplement the formal education.

The study material has been prepared by the CEDTI, document centre. It is based on the vast and rich instructional experience of all the CEDTI centres. Any suggestions on the improvement of the study material will be most welcome.

(R. S. Khandpur)
Director General (CEDTI)

TABLE OF CONTENTS

UNIT	CHAPTER NAME	PAGE NO.
	SECTION - A	
1	C Language	11
2	Numeric Constants and Variables	17
3	Data Input and Output	23
4	Operators and Expressions	29
5	The Decision Control & Loop Structure	35
	SECTION - B	
6	Arrays	45
7	Functions	51
8	Standard Library String Functions	57
9	Dynamic Data Structures in C	61
10	Structures and Union	75
11	Disk I/O Functions	83
12	Graphics Features in C	91

PREFACE

In recent years, there has been a major trend toward the use of C among serious programmers. Among the many reasons for C's popularity are the following :

- C is largely machine-independent. Programs written in C are easily ported from one computer to another.
- C is widely available. Commercial C compilers are available for most personal computers, mini-computers, and mainframes.
- C includes certain low-level features that are normally available only in assembly or machine language.
- Programs written in C compile into smaller programs that execute efficiently.
- C is a flexible, high-level, structured programming language.

This course material provides instruction in computer programming with C. It includes complete and understandable explanations of the commonly used features of C. In addition, the material presents a contemporary approach to programming, stressing the importance of clarity, legibility, modularity, and efficiency in program design. Thus, the reader is exposed to the principles of good programming practice as well as the specific rules of C. Examples of C programs are presented throughout the text, beginning with the first chapter. The use of an interactive programming style is emphasized throughout the text.

This volume can be used by the beginners in the programming as well as professionals. It is particularly well suited for an introductory programming course.

Sets of review questions and problems are provided at the end of each chapter. The review questions enable readers to test their assimilation of the material presented within each chapter. They also provide an effective chapter summary. The problems reinforce the principles presented within each chapter.

SECTION - A

COMPETENCY OBJECTIVES

The objective of this Section is to provide instructions in Computer Programming with C language . It includes complete explanations of the commonly used feature of C. At the end of the course, a student should be able to :-

- Appreciate the flexible & structured programming of C language.
- Describe Constants & Variables used in C.
- Carry out procedures of Data input and output.
- Understand various operators and expressions used in C.
- Execute various decision controls and loop structures in C.

CHAPTER - 1

C LANGUAGE

History

C was developed by Dennis Retchie in 1972 at the Bell Telephone Laboratories in U.S.A. C was derived from a Language known as BCPL which was evolved at the Massachusetts Institute of Technology in the late 60's. BCPL was used to develop an operating system known as MULTICS for early multi-user time shared computers. One of the aims of BCPL was to achieve efficiency in compiled code. Thus BCPL was defined such that a translator could produce efficient machine Language code. C being a successor of BCPL has a similar philosophy. C Language has been defined so that it has the advantages of a high level language namely machine independence. At the same time it is concise, providing only the bare essentials required in a language so that a translator can translate it in to an efficient machine language code. Until 1978, C was confined to use within Bell Laboratories. In 1978, when Brian Kernighan and Ritchie published a description of the language, known as "k & rc" computer professionals got impressed with C's many desirable features.

By mid 1980s, popularity of C became wide spread Numerous C Compiler were written for computers of all sizes.

This chapter Covers following Topics.

1. The structure of a C program.
2. How to develop small program in C.
3. How to input data and print the results obtained from a C program.

Structure of a C Program:-

The structure of a C program can by explained by taking an example of a C program to calculate area and perimeter of a rectangle. The program is written as follows.

```
/*      Example program */
/*      This program finds the area and perimeter of a rectangle */
  #     include <stdio.h>
        main( )
```

```
{
        int p,q, area, perimeter;
        p = 4
        q = 6
        area = p*q;
        perimeter = 2 * (p+q);
        printf("area = %d\n", area);
        printf("perimeter = % d\n", perimeter);
}
        /* End of main */
```

Comments

In the above program the first line in the program starts with /* and ends with */. Any thing written between /* and */ is called a comment. In the C Language comments are an aid to the programmer to read and understand a program. It is not a statement of the language. The compiler ignores comments It is a good practice to include comments in a program which will help in understanding a program.

PREPROCESSOR DIRECTIVE

Now let us observe the line
# include <stdio.h>

This is called a preprocessor directive. It is written at the beginning of the program. It commands that the contents of the file stdio.h should be included in the compiled machine code at the place where # include appears. The file stdio.h contains the standard input/output routines. All preprocessor directives begin with pound sign # which must be entered in the first column. The # include line must not end with a semicolon. Only one preprocessor directive can appear in one line.

Main( ) Function

The next line is main( ). It defines what is known as a function in C. A C program is made up of many functions. The function main( ) is required in all C programs. It indicates the start of a C program. We will use main( ) at the beginning of all programs. Observe that main( ) is not followed by a comma or semicolon.

Braces {And}

Braces{and} enclose the computations carried out by main ( ). Each line in the program is a statement. Every statement is terminated by a semicolon; The statement itself can be written anywhere in a line. More than one statement can be on a line as a semicolon separates them. However it is a good practice to write one statement per line.

Declaration

In the above program the first statement **int p, q, area, perimeter**;

This statement is called a declaration. It informs the compiler that p,q, area and perimeter are variable names and that individual boxes must be reserved for them in the memory of the computer. Further it tells that the data to be stored in the memory boxes named p,q, area and perimeters are integers namely, whole numbers without a fractional part(e.g, 0,1,2,...). The statement ends with a semicolon The effect of statement is shown in figure given below;

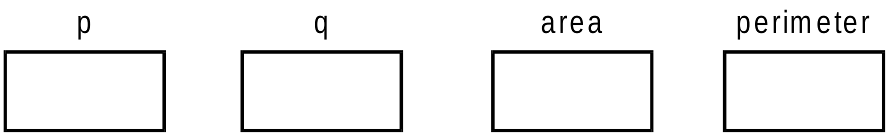


Fig. 1.1 Effect of defining p, q, area and perimeter as integer variable names.

Assignment Statements

The statement p=4; is an assignment statement. It commands that the integer 4 be stored in the memory box named p. when the statement is executed the integer 4 will be stored in the memory box named p as shown in fig. 1.2 . This statement assigns a value 4 to the variable name p.

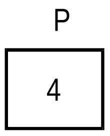
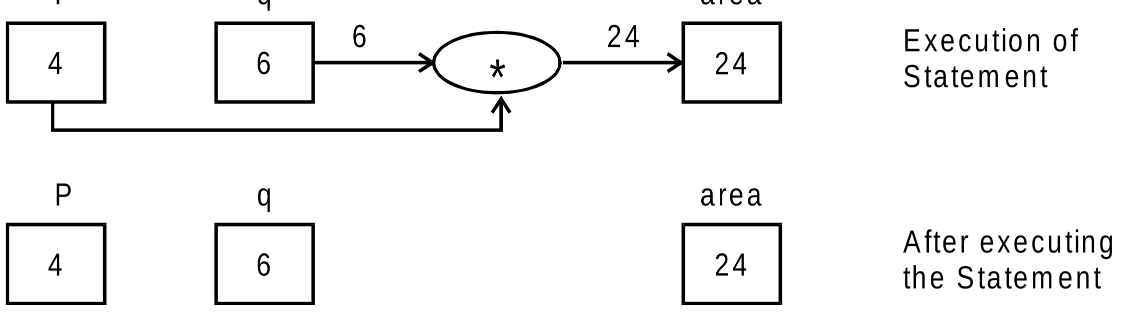


Fig. 1.2

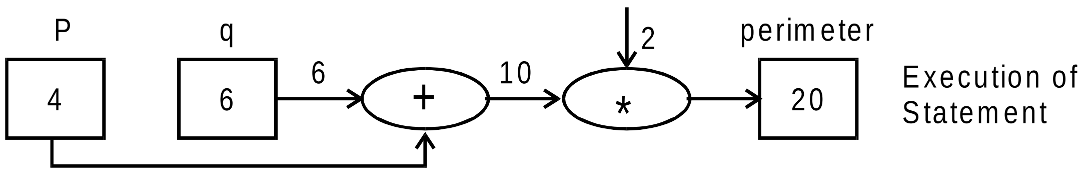
The next statement q=6; assigns 6 to q

Arithmetic Statement

The statement area=p*q; is an arithmetic statement. It commands that the numbers stored in memory boxes p and q should be copied in to the CPU. The original contents of p and q remain in their respective boxes. These numbers are multiplied by the CPU and product is stored in box named area. After executing this statement the box named area will contain 24 as shown in fig. 1.3.



The next statement perimeter = 2 *(p+q) is also an arithmetic statement.



The next two statements in the program are commands to display the contents of memory box named area and perimeter respectively. The library function used for display is printf( ). the general form of this function is

```
printf (format string, variable 1, variable2, ——variable n);
```

The format string is enclosed in quotes. It specifics any message to be printed and the manner in which the contents of variables are to be displayed for each of the variables in the list of variables.

```
printf(“area = % d\n”, area);
```

The format string is % d\n The symbol % d says interpret the variable area occurring after the comma in the printf statement as an integer and display its value”. The symbol \n causes the display to advance to the next line. Thus when this statement is carried out we will see on the screen

```
area = 24
```

After the statement

```
printf(“perimeter = % d\n”, perimeter);
```

we will have on the screen

```
area = 24
perimeter = 20
```

Some C Program Examples

Let us write a program to convert a temperature given in Celsius to Fahrenheit - The formula for conversion is

f = 1.8C + 32

```
/* Example program */
/* This program converts a Celsius temperature to Fahrenheit */
# include <stdio.h>
  main ( )
  {
        float fahrenheit, celsius;
        scanf(“%f”, & celsius);
        fahrenhit = 1.8*celsius + 32.0;
printf (“fahrenheit = % f\n”, fahrenheit);
}         /* End of main */
```

Note in the above program after # include we have defined the main function. If the statement. float fahrenheit, celsius; is observed we see that

This is a declaration which informs the compiler that fahrenheit and celsius are variable names in which floating point number will be stored. By floating point we mean a number with a fractional part.

The statement scanf(“%f”, & celsius) is a command to read a floating point number and store it in variable name celsius. The format string %f indicates that floating point number is to be read. The next statement is an arithmetic statement contents of celsius is multiplied by 1.8 and added to 32.0 and stored in fahrenheit. The number 32 is written with a decimal point to indicate that it is a floating point number. The last statement is to print the answer.

ASSIMILATION EXERCISE

Q.1 Where was C originally developed and by whom ?

Q.2 What are the major components of a C program ? What significance is attached to name “main” ?

Q.3 How can comments be included within a C program ? Where can comments be placed?

CHAPTER - 2

NUMERIC CONSTANTS AND VARIABLES

CONSTANTS

The term constant means that it does not change during the execution of a program constants may be classified as:

(i) Integer Constant
(ii) Floating point Constant

Integer Constant

Integer constants are whole numbers without any fractional parts. There are three types of Integer constant:

(i) Decimal constant (base 10)
(ii) Octal Constant (base 8)
(iii) Hexa decimal constant(base 16)

Allowed digit in decimal constant 0,1,.2,3,4,5,6,7,8,9,. The digits of one octal constant can be 0,1,2,3,4,5,6,7 and that in hexadecimal constants are 0,1,2,3,4,5,6,7,8.9,A,B,C,D,E,F,

Rule for decimal constant

A decimal integer constant must have at least one digit and must be written without a decimal point. The first digit in the constant should not be 0. It may have either sign + or -. If either sign does not precede to the constant it is assumed to be positive.

Following are valid decimal integer constants:

(i) 12345
(ii) 3468
(iii) -9746

The following are invalid decimal integer constants :-

(i) 11. (Decimal point not allowed)
(ii) 45,256(comma not allowed)
(iii) $ 125 (currency symbol not allowed)
(vi) 025 (first digit should not be 0 for a decimal integer)
(v) ƒ 248 (first symbol not a digit)

Octal Constant

An octal constant must have at least one digit and start with the digit 0. It must be written without a decimal point. It may have either sign + or -. A constant which has no sign is taken as positive.

The following are valid octal constants.

(i) 0245
(ii) - 0467
(iii) +04013

The following are invalid octal constants

(i) 25(Does not begin with 0)
(ii) 0387 (8 is not an octal digit)
(iii) 04.32 (Decimal point not allowed)

Hexadecimal constant

A hexadecimal constant must have at least one hexadecimal digit and start with Ox or OX. It may have either sign

The following are valid hexadecimal constants

(i) OX 14 AF
(ii) OX 34680
(iii) – OX 2673E

The following are invalid hexadecimal constants:

(i) Ox14AF
(ii) OX34680
(iii) -Ox2673E

The following are invalid hexadecimal constants:

(i) 0345 (Must start of Ox)
(ii) 0x45H3 (H not a hexa decimal digit)
(iii) Hex 2345 (Ox defines a hexadecimal number, not Hex)

Floating Point Constants

A floating point constant may be written in one or two forms could fractional form or the exponent form, the rules for writing a floating point constant in these two forms is given as follows.

Rule (Fractional Form)

A floating point constant in a fractional form must have at least one digit to the right of the decimal point. It may have either the + or the - sign preceding it. If a sign does not precede it then it is assumed to be positive.

Following are valid floating point constants:

(i) 1.0
(ii) –0.5678
(iii) 5800000.0
(iv) –0.0000156

Following are invalid:

(i)	1	(Decimal point missing)
(ii)	1.	(No digit following decimal point)
(iii)	-1/2,	(Symbol/illegal)
(iv)	.5	(No digit to the left of the decimal)
(v)	58, 678.94	(Comma not allowed)

Rule (Exponential form)

A floating point constant in the exponent form consists of a mantissa and an exponent. The mantissa must have at least one digit. It may have a sign. The mantissa is followed by the letter E or e and the exponent. The exponent must be an integer(without a decimal point) and must have at least one digit. A sign for the exponent is optional.

The following are valid floating point constants in exponent form:

(i)	(a)	152 E08	(b)	152.0 E8	(c)	152 e+8
	(d)	152 E+08	(e)	15.2 e9	(f)	1520 E7

(ii) -0.148E-5

(iii) 152.859 E25

(iv) 0.01540 e05

The following are invalid:

(i) 152.AE8 (Mantissa must have a digit following).

(ii) 152*e9 (* not allowed).

(iii) +145.8E (No digit specified for exponent).

(iv) -152.9E5.5 (Exponent can not be a fraction).

(v) 0.158 E+954 (Exponent too large).

(vi) 125, 458.25 e-5 (comma not allowed in mantissa).

(vii) 02E8 (A digit must precede in mantissa).

C Variables and Their Types

In C, a quantity which may vary during program execution is called a variable. Variable names are names given to locations in the memory of computer where different constants are stored. These locations can contain integer, real or character constants. In any language the types of variables that it can support depends on the type of constants that it can handle. This is because a constant stored in a location with a particular type of variable name can hold only that type of constant. For example, a constant stored in a memory location with an integer variable name must be an integer constant. One stored in location with a real variable name must be a real constant and the one stored in location with a character variable name must be a character constant.

Rules for Variable Names

(i) A variable name is any combination of 1 to 8 alphabets, digits or underscores.

(ii) The first character in the variable name must be an alphabet

(iii) No commas or blanks are allowed within a variable name.

(iv) No special symbol other than an underscore(as in gross-sal) can be used in a variable name.

e.q. si_int
m_hra
pop_e_89

C compiler is able to distinguish between the variable names, by making it compulsory for you to declare the type of any variable name you wish to use in a program. This type of declaration is done at the beginning of the program.

Following are the examples of type declaration statements:

e.q. int si, m-hra;
float bassal;
char code;

C Keywords

Keywords are the words whose meaning has already been explained to the C compiler. Keywords can not be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. The keywords are also called ‘Reserved words’. There are 32 keywords available in C. following is list of keywords in C.

auto	double	if	static
break	else	int	struct
case	enum	long	typedef
const	float	register	union
continue	far	return	unsigned
default	for	short	void
do	goto	signed	white

ASSIMILATION EXERCISE

Q.1 Determine which of the following are valid identifiers. If invalid, explain why ?

(a) record 1 (b) return (c) name_and_address (d) name-and-address

Q.2 What is the range of long double?

Q.3 What are the rules for naming variables?

Q.4 Discuss the various types of constants?

CHAPTER - 3

DATA INPUT AND OUTPUT

SINGLE CHARACTER INPUT - THE getchar( ) FUNCTION

Single characters can be entered in to the computer using the C library function getchar( ). The getchar function is a part of the standard C Language i/o Library. It returns a single character from a standard input device. The function does not require any arguments, though a pair of empty parentheses must follow the word getchar.

In general terms a reference to the getchar function is written as

```
character variable = getchar( );
```

Here character variable refers to some previously declared character variable

SINGLE CHARACTER OUTPUT-THE putchar( ) FUNTION

The putchar( ) function, like getchar( ), is a part of the standard C language i/o library. It transmits a single character to a standard output device. The character being transmitted will normally be represented as a character- type variable. It must be expressed as an argument to the function enclosed in parentheses following the word putchar.

In general a reference to the putchar function is written as .

```
putchar( char var )
```

<u>e.q</u> A C-Program contains the following statement :

```
char C;
__ _
__ _
__ _
__ _
putchar(C);
```

The first statement declares that C is a character type variable. The second statement causes the current value of C to be transmitted to the standard output device.

ENTERING INPUT DATA THE scanf( ) FUNTION

Input data can be entered into the computer from a standard input device by means of the C library function scanf().

In general terms, the scanf function is written as

scanf(Control string, arg1,arg2,.....,argn)

Here control string refers to a string containing certain required formatting information, and arg1, arg2,...arg n are arguments that represent the individual input data items. The arguments represent pointers that indicate the addresses of the data item within the computers memory.

The control string comprises individual groups of characters with one character group for each input data item. Each character group must begin with a a percent sign(%). In its simplest form a single character group will consist of the percent sign, followed by a conversion character which indicates the type of the corresponding data item.

COMMONLY USED CONVERSION CHARACTERS FOR DATA INPUT

Conversion character	Meaning
c	data item is a single character
d	data item is a decimal integer
f	data item is a floating point value
h	data item is a short integer
I	data item is a decimal, hexadecimal or octal integer
o	data item is an octal integer
s	data item is a string followed by white space character
u	data item is an unsigned decimal integer
x	data item is a hexadecimal integer

e.q of scanf function

```
# include <stdio.h>
main( )
{
        char item [20];
        int partno;
        float cost;
        .   .   .
        scanf(“%s% d % f”, item, &partno, & cost);
        .   .   .
    }
```

The first character group % s indicates that the first argument (item) represents a string the second character group, %d indicates that the second argument (& partno) represents a decimal integer value. The third character group % f, indicates that the third argument (& cost) represents a floating point value.

<u>e.q.</u> consider the skeleton of the following program

```
    # include <stdio.h>
    main( )
    {
        char item [20];
        int partno;
        float cost;
        .   .   .
        scant(“%s%d%f”, item &partno, &scost);
        }
```

The following data items could be entered from the standard input device when the program is executed.

```
    fastener 12345 0.05
                or
    fastener
    12345
     0.0 5
```

Now let us again consider the skeleton structure of a C program

```
    # include <stdio.h>
    main ( )
    {
        int a,b,c;
        .  .  .
        scanf (“%3d %3d %3d”, & a, & b, & c);
        .   .   .
    }
```

Suppose the input data items are entered as

```
    1     2     3
```

Then the following assignment is will result

```
    a = 1,  b = 2, c = 3
```

It data is entered as

```
123     456     789
```

The following assignments would be

```
a = 123,        b = 456,        c = 789
```

Now suppose that the data had been entered as

```
123456789
```

The assignments would be

```
a = 123,        b = 456,        c = 789
```

Again consider the following example

```
# include <stdio.h>
main ( )
{ int i ;
float x;
char c ;
.     .     .
scanf (“%3d %5f %c”, & i, & x, & c);
.     .     .
}
```

If the data item are entered as

```
10256.875 T
```

These when the program will be executed

10 will be assigned to i
256.8 will be assigned to x

and the character 7 will be assigned to c. the remaining two input characters(5 and T) will be ignored.

WRITING OUTPUT DATA - THE printf( ) FUNCTION

Output data can be written from the computer on to a standard output device using the library function printf the general form of printf function is

```
printf(control string, arg 1, arg 2, . . ., argn)
```

where control string refers to a string that contains formatting information.

arg1, arg 2, . . ., argn are arguments that represent the individual output data items.

e.g:- Following is a simple program that makes use of the printf function.

```
# include <stadio.h>
# include <math.h>
main ( )  /* Print several floating-point numbers */
{
```

```
            float i = 2.0, j = 3.0 ;
            printf (“%f %f %f %f”, i, j, i+j, sqrt (i + j ));
    }
```

Executing the program produces the following output

```
2.000000      3.000000      5.000000      2.236068
```

The gets( ) and puts( ) FUCTIONS

The gets and puts are the functions which facilitate the transfer of strings between the computer and the standard input/output devices.

Each of these functions accepts a single argument.

The argument must be a data item that represents a string (e.g, a character array). The string may include white space characters. In the case of gets, the string will be entered from the keyboard and will terminate with a newline character (“i’e” the string will end when the user presses the RETURN key).

```
# include <stdio.h>
main ( )
{
        char line [80];
        gets(line);
        puts(line);
        }
```

Now suppose following string is entered from the standard input device

```
I am happy
```

Now the output of the program will be

```
I am happy
```

ASSIMILATION EXERCISE

Q.1 A C program contains the following statements :

```
# include <stdio.h>
char a, b, c;
```

(i) Write appropriate *getchar* statement that will allow values for a, b and c to be entered into the computer

(ii) Write appropriate putchar statements that will allow the current values of a,b and c to be written out of the computer.

Q.2 When entering a string via the scanf function using an s_type conversion factor, how is the string terminated?

Q.3 What is the purpose of the printf function? How is it used within a C program ? compare with the *putchar* function ?

CHAPTER - 4

OPERATORS AND EXPRESSIONS

1. Arithmetic operators

There are five arithmetic operators in C . They are

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

The operator % is known as the modulus operator

e.g. Suppose that a and b are integer variables whose values are 10 and 3 respectively, several arithmetic expressions involving these variables are shown below together with their resulting values

Expression	Value
a+b	13
a - b	7
a * b	30
a / b	3
a % b	1

Now suppose that v1 and v2 are floating - point variables whose values are 12.5 and 2.0 respectively. several arithmetic expressions involving these variables are shown below, together with their resulting values

Expression	Value
v1 + v2	14.5
v1 - v2	10.5
v1 * v2	25.0
v1 / v2	6.25

Now suppose c1 and c2 are character - type variables that represent the characters P and T, respectively. Several arithmetic expression that make use of these variables are shown below together with their resulting values (based upon the ASCII character set)

Expression	Value
C1	80
C1 + C2	164
C1 + C2 +5	169
C1 + C2 +'5'	217

P is encoded as (decimal) 80, T is encoded as 84, and 5 is encoded as 53 in the ASCII character set, as shown above.
Suppose that a and b are integer variables whose values are 11 and - 3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values

Expression	Value
a+b	8
a-b	14
a*b	-33
a/b	-3

Type cast

The value of an expression can be converted to a different data type. If desired to do so the expression must be preceded by the name of the desired data type enclosed in parentheses.

(data type) expression

This type of construction is known as a cast (It is also called a type cast.)

Suppose that i is an integer variable whose value is 7, and f is floating- point variable whose value is 8.5. The expression (i +f) % 4 is invalid because the first operand (i + f) is floating point rather than integer. However the expression ((int) (i + f)) % 4 forces the first operand to be an integer and is therefore valid, resulting in the integer remainder 3.

UNARY OPERATORS

C includes a class of operators that act upon a single operand to produce a new value such operators are known as unary operators. Unary operators usually precede their single operands , though some unary operators are written after their operands.

(i) **UNARY MINUS**

Unary minus is a unary operator where a minus sign precedes a numerical constant, a variable or an expression. In C, all numeric constants are positive. Thus a negative number is actually an expression consisting of the unary minus operator followed by a positive number.

e.g. - 743, -0 X 7FFF, -0.2, -5E -8, -root 1, -(x +y) -3 * (x+y)

(ii) **INCREMENT OPERATOR AND DECREMENT OPERATOR**

The increment operator (++) causes its operand to be increased by one where as the decrement operator causes its operand to be decreased by one. The operand used with each of these operator must be a single variable.

e.g. Suppose i is an integer variable that has been assigned a value of 5. the expression ++i, causes the value of i to be increased by one, Whereas the decrement operator causes the value to be decreased by 1 so, now the new variable of i will be 4

i ++	means	i = i + 1
— i	means	i = i-1

The increment and decrement operators can each be utilized in two different ways, depending an whether the operator is written before or after the operand. If the operator precedes the operand (e.g. ++ i) then the operand will be altered in value before it is utilized for its intended purpose within the program. If however, the operator follows the operand (e.g i ++) then the value of the operand will be altered after it is utilized

e.g.
```
printf (“ i = %d\n”, i);
printf (“ i = % d\n”, ++i);
printf (“ i = %d\n”, i);
```

There printf statements will generate following three lines of out put

```
i = 1
i = 2
i =2
```

Now let us take the second case

```
printf (“ i = %d\n”, i);
printf (“ i = %d\n”, i++);
printf ( i = %d\n”, i )
```

The statement will generate the following three lines of output

```
i = 1
i = 1
i = 2
```

LOGICAL OPERATORS

There are two logical operators also available in C. They are

Operator	Meaning
&&	and
||	or

The result of a logical **and** operation will be true only if both operands are true where as the result of a logical **or** operation will be true if either operand is true or if both operands are true. In other words the result of a logical or operation will be false only if both operands are false. Suppose i is an integer variable whose value is 7, f is a floating point variable whose value is 5.5 and C is a character variable that represents the character 'w', several complex logical expressions that make use of these variables are shown below:

Expression	Interpretation	Value
(i > = 6)&&(c = = 'w')	true	1
(i >= 6)&&(c = = 119)	true	1
(f < 11) && (i > 100)	false	0
(c = 'p') || ((i + f) < = 10)	true	1

RELATIONAL AND LOGICAL OPERATORS

These are four relational operators in C. They are

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

EQUALITY OPERATORS

There are two equality operators in C which are as follows

Operators	Meaning
= =	Equal to
! =	Not equal to

The six operators mentioned above are used to form logical expressions representing conditions that are either true or false.

The resulting expression will be of type integer, since true is represented by the integer value 1 and false is represented by the value 0.

e.g . Let us suppose i, j and k are integer variables whose values are 1,2 and 3 respectively. Several logical expressions involving these variables are shown below.

Expression	Interpretation	Value
i < j	true	1
(i + j) > = k	true	1
(j + k) > (i + 5)	false	0
k ! = 3	false	0
j = = 2	true	1

Expression	Interpretation	Value
(i > = 6)&&(c == 'w')	true	1
(i > = 6) || (c = = 119)	true	1
(f < 11) && (i > 100)	false	0
(c1 == 'p') || ((i + f) < = 10)	true	1

Both operands are true hence expression is true
Both operands are true hence expression is true
Second operand is false hence expression is false
First operand is true hence expression is true

ASSIGNMENT OPERATORS

The commonly used assignment operator is = Assignment expressions that make use of this operator are written in the form

Identifier = expression

Here identifier represents a variable and expression represents a constant, a variable or a more complex expression e.g. Here are some typical assignment expressions that make use of the = operator

```
a = 3
x = y
sum = a+b
area = length  * width.
```

THE CONDITIONAL OPERATOR

Conditional operator is (?:) A expression that makes use of the conditional operator is called a conditional expression

for e.g. expression 1? expression 2 : expression 3

When evaluating a conditional expression, expression 1 is evaluated first. If the expression 1 is true then expression 2 is evaluated and this becomes the value of conditional expression However, if expression 1 is false then expression 3 is evaluated and this becomes the value of the conditional expression e.g. In the conditional expression shown below suppose f and g are floating point variables. The conditional expression is

(f < g)? f: g

It takes the value f if f is less than g otherwise the conditional expression takes on the value of g. In other words conditional expression returns the value of the smaller of the two variables.

ASSIMILATION EXERCISE

Q.1 What is an operator ? Describe several different type of operator included in C.

Q.2 A C program contains the following declarations

```
int i, j;
long ix;
short s;
float x;
double dx;
char c;
```

Determine the data type of each of the following expressions,

(a) x + c	(c) i + x	(e) ix + 4	
(b) dx + x	(d) s + 4	(f) s + c	

Q.3 A C program contains the following declaration and initial assignments

```
int i = 8, j = 5;
double   x =  0.005, y = - 0.01;
char c = ‘c’, d = ‘d’;
```

determine the value of each of the following expressions which involve the use of library functions

(a)	abs (i - 2 * j)	(e)	log (x)
(b)	abs (x + y)	(f)	sqrt (x * x + y * y)
(c)	toupper (d)	(g)	strlen (“ hello\ 0”)
(d)	floor (x)	(h)	pow (x - y)

CHAPTER - 5

THE DECISION CONTROL & LOOP STRUCTURE

The decision control structure in C can be implemented in C using

(a) The if statement
(b) The if - else statement
(c) The conditional operators

The if Statement

The general form of if statement looks like this:

```
if (this condition is true)
        execute this statement;
```

Here the keyword **if** tells the compiler that what follows, is a decision control instruction. The condition following the keyword if is always enclosed within a pair of parentheses. If the condition, whatever it is true, then the statement is executed. It the condition is not true then the statement is not executed instead the program skips past it. The condition in C is evaluated using C's relational operators. The relational operators help us to build expression, which are either true or false

e.q

Expression	Is true if
X = = Y	X is equal to Y
X ! = Y	X is not equal to Y
X <Y	X is less than Y
X>Y	X is greater than Y
X< = Y	X is less than or equal to Y
X> = Y	X is greater than or equal to Y

Demonstration of if statement

```
main( )
{
        int num;
        printf("Enter a number less than 10");
        scanf("%d", & num);
        if(num < = 10)
        printf("The number is less than 10");
        }
```

Multiple Statements within if

If it is desired that more than one statement is to be executed if the condition following if is satisfied, then such statements must be placed within pair of braces.

e.q The following program demonstrate that if year of service greater than 3 then a bonus of Rs. 2500 is given to employee. The program illustrates the multiple statements used within if

```
/* calculation of bonus */
main( )
{
int bonus, CY, Yoj, yr-of-ser;
printf("Enter current year and year of joining");
scanf("%d %d", &cy, &yoj);
yr-of-ser = CY-Yoj;
if(yr-of-ser > 3)
{
bonus = 2500;
printf("Bonus = Rs. %d", bonus);
}
}
```

If - else The if statement by itself will execute a single statement or a group of statements when the condition following if is true. it does nothing when the condition is false. It the condition is false then a group of statements can be executed using else statement. The following program illustrates this

```
/* Calculation of gross salary */
main( )
{
float bs, gs, da, hra;
printf("Enter basic salary");
scanf("%f", & bs);
if(bs <1500)
{ hra = bs * 10/100;
  da  = bs * 90/100;
}
else
{ hra = 500;
```

```
  da  = bs * 98/100;
}
gs = bs+hra+da;
printf("gross salary = Rs. %f", gs);
}
```

Nested if - else If we write an entire if - else construct within the body of the if statement or the body of an else statement. This is called 'nesting' of ifs.

e.g.

```
if(condition)
{
  if (condition)
        do this;
   else
        { do this;
          and this;
           }
}
        else
        do this;
```

The Loop Control structure

These are three methods by way of which we can repeat a part of a program. They are:

(a) Using a for statement
(b) Using a while statement
(c) Using a do-while statement

The while Loop:-

The general from of while is as shown below:

```
initialise loop counter;
while (test loop counter using a condition)
{
        do this;
        and this;
        increment loop counter;
}
```

The parentheses after the while contains a condition so long as this condition remains true all statements within the body of the while loop keep getting executed repeatedly

<u>for e.g.</u>

```
/* calculation of simple interest for 3 sets of p, n and r */
main( )
{    int p,n, count;
        float r, si;
        count = 1;
        while(count < = 3 )
        {
                printf("\n Enter values of p, n and r");
                scanf("%d %d %f", & p, & n, & r);
                si = p*n*r/100;
                printf("simple interest = Rs. %f", si);
                count = count +1;
        }
```

The do-while Loop

The do-while loop takes the following form

```
        do
        {
                this;
                and this;
                and this;
                and this;
                } while (this condition is true);
```

There is a minor difference between the working of <u>while</u> and <u>do-while</u> loops. The difference is the place where the condition is tested. The <u>while</u> tests the condition before executing any of the statements within the <u>while</u> loop. As against this the do-while tests the condition after having executed the statements within the loop.

<u>e.g</u>:-

```
        main( )
        {
        while(5<1)
                printf("Hello \n");
        }
```

In the above e.q. the printf will not get executed at all since the condition fails at the first time itself. Now let's now write the same program using a do-while loop.

```
        main( )
        {
                do
                {
                printf ("Hello \n");
                } while (5<1);
        }
```

In the above program the printf( ) would be executed once, since first the body of the loop is executed and then the condition is tested.

The for Loop

The general form of for statement is as under:

```
        for( initialise counter; test counter; increment counter)
        { do this;
        and this;
        and this;
        }
```

Now let us write the simple interest problem using the for loop

```
    /* calculation of simple interest for 3 sets of p,n and main( )
    {
    int p,n, count;
    float r, si;
    for(count = 1; count <=3; count = count +1)
    {
        printf("Enter values of p,n and r");
        scanf("%d %d %f", & p, & n, & r);
        si = p * n * r / 100;
        printf("Simple interest = Rs %f \n", si);
    }
}
```

The break Statement

The keyword break allows us to jump out of a loop instantly without waiting to get back to the conditional test. When the keyword break is encountered inside any C loop, control automatically passes to the first statement after the loop.

for e.q. The following program is to determine whether a number is prime or not.

Logic:- To test a number is prime or not, divide it successively by all numbers from 2 to one less than itself. If the remainder of any of the divisions is zero, the number is not a prime.

following program implements this logic

```
main( )
{
        int num, i;
        printf("Enter a number");
        scanf("%d", &num);
        i = 2
        while (i < = num -1)
        {
                if (num%i= = 0)
                {
                        printf("Not a prime number");
                        break;
                        }
                        i + +
                }
                if(i = = num)
                        printf("Prime number");
        }
```

The continue Statement

The keyword continue allows us to take the control to the beginning of the loop bypassing the statements inside the loop which have not yet been executed. When the keyword continue is encountered inside any C loop, control automatically passes to the beginning of the loop. for e.g.

```
main( )
        {
                int i,j;
                for(i = 1; i< = 2; i++)
                {
                        for(j=1; j<=2; j++)
                        {
                        if (i= =j)
                                continue;
                        printf("\n%d%d\n", i,j);
                }
        }
```

The output of the above program would be....

```
12
21
```

when the value of i equal to that of j, the continue statement takes the control to the <u>for loop (inner)</u> bypassing rest of the statements pending execution in the <u>for loop(inner)</u>.

The Case Control structure

switch Statement:-

The switch statement causes a particular group of statements to be chosen from several available groups. The selection is based upon the current value of an expression that is included within the switch statement. The form of switch statement is.

```
switch (integer expression)
{     case constant 1:
                do this;
                break;
        case constant 2:
                do this;
                break;
        case constant 3:
                do this;
                break;
        default;
                do this;
}
```

ASSIMILATION EXERCISE

Q.1 Write a loop that will calculate the some of every third integer beginning with i = 2 (i.e. calculate the some 2 + 5 + 8 + 11 + ———) for all values of i that are less than 100. Write the loop three different ways.

Q.2 Write a switch statement that will examine the value of integer variable called flag and print one of the following messages, depending on the value assigned to flag.

(a) HOT, if flag has a value of 1
(b) LUKE WARM, if flag has value of 2
(c) COLD, if flag has value of 3
(d) OUT OF RANGE, if flag has any other value.

Q.3 Describe the output that will be generated by each of following C programs.

(a)
```
# include <stdio.h>
main ( )
{
int i = 0, x = 0;
while (  i < 20) {
if ( i % 5 = = 0) {
x + = i;
printf (“%d”, x);
}
++i;
}
printf (“ \n x = %d”, x);
}
```

(b)
```
# include < stdio.h>
main ( )
{
int i, j, x = 0;
for ( i = 0; i < 5 ; ++ i)
for ( i = 0; j < i; ++ j) {
x + = ( i + j - 1);
printf ( “ %d”, x);
}
printf (“\n x = %d”, x)
}
```

SECTION - B

COMPETENCY OBJECTIVES

The objective of this Section is to provide the advanced features for programming with C language. It includes complete explanations of these features. At the end of the course, a student should be able to :-

- ❖ Understand and implement Arrays.
- ❖ Appreciate the use of functions.
- ❖ Use the standard library string functions.
- ❖ Develop dynamic data structures in C.
- ❖ Understand structures and unions in C.
- ❖ Apply graphics features in C language.

CHAPTER - 6

ARRAYS

What are Arrays

Let us suppose we wish to arrange the percentage marks obtained by 100 students in ascending order. In such a case there are two options to store these marks in memory:

(a) Construct 100 variables to store percentage marks obtained by 100 different students i.e “each variable containing one students marks.

(b) Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Clearly, the second alternative is better because it would be much easier to handle one array variable than handling 100 different variables

Now we can give a formal definition of array . An array is a collective name given to a group of similar quantities. These similar quantities could be percentage marks of 100 students, or salaries of 300 employee or ages of 50 employees. Thus an array is a collection of similar elements. These similar elements could be all ints, or all floats or all chars etc. Usually, the array of characters is called a ‘string’, where as an array of ints or floats is called simply an array. All elements of any given array must be of the same type i.e we can’t have an array of 10 numbers, of which 5 are ints and 5 are floats.

ARRAY DECLARATION

To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how. large an array we want.

for e.g. `int marks [30];`

Here int specifies the type of variable, marks specifies the name of the variable. The number 30 tells how many elements of the type int will be in our array. This number is often called the ‘dimension’ of the array. The bracket [ ] tells the compiler that we are dealing with an array.

ACCESSING ELEMENTS OF AN ARRAY

To access an individual element in the array we have to subscript it, that is we have to put the number in the brackets following the array name. All the array elements are numbered starting with 0. Thus, marks [2] is not the second element of array but it is actually the third element. Thus marks [i] refers to (i + 1) th element of the array.

Let us take an example of a program using array

```
main ( )
{ float avg, sum=0;
int i;
int marks [30]; /* array declaration*/
for ( i =0; i < = 29; i ++)
{
printf (“\n Enter marks “);
scanf (“%d”, &marks [i]);
}
for ( i = 0; i <= 29; i ++)
sum = sum +  marks [i];
avg = sum /30;
printf (“\n Average marks = % f”,  avg);
}
```

ENTERING DATA IN TO THE ARRAY

The section of code which places data in to an array is

```
for (i=0; i<= 29; i++)
{
printf (“\n Enter marks”)
scanf (“%d”, &marks [i]);
}
```

The above section will read about 30 elements numbered from 0 to 29 in to the marks array. This will take input from the user repeatedly 30 times.

READING DATA FROM ARRAY

```
for ( i=0; i <= 29; i++);
sum = sum + marks [i];
avg = sum / 30;
printf (“\n Average marks = % f”, avg );
```

The rest of the program reads the data back out of the array and uses it to calculate the average. The for loop is much the same, but now the body of loop causes each student’s marks to be added to a running total stored in a variable called sum. When all the marks have been added up, the result is divided by 30, the numbers of students to get the average.

Let us summarize the facts about array

(a) An array is a collection of similar elements.

(b) The first element in the array is numbered 0, the last element is 1 less than the size of the array.

(c) An array is also known as subscripted variable .

(d) Before using an array its type and dimension must be declared.

(e) However big an array is, its elements are always stored in contiguous memory locations.

ARRAY INITIALISATION

To initialise an array while declaring it. Following are a few examples which demonstrate this

```
int num [6]        =     {2, 4, 12, 5, 45, 5};
int n [ ]          =     {2, 4, 12, 5, 45, 5};
float press [ ]    =     { 12.3, 34.2, -23.4, - 11.3}
```

The following points should be noted

(a) Till the array elements are not given any specific values, they are suppose to contain garbage values.

(b) If the array is initialized where it is declared mentioning the dimension of the array is optional as in the 2nd example above.

MULTIDIMENSIONAL ARRAYS

In C one can have arrays of any dimensions. To understand the concept of multidimensional arrays let us consider the following 4 x 5 matrix

$$\begin{array}{cc|ccccc} & & \multicolumn{5}{c}{\text{Column numbers (j)}} \\ \hline & 0 & 10 & 4 & 3 & -10 & 12 \\ & 1 & 2 & 3 & 0 & 61 & 8 \\ \text{Row number (i)} & 2 & 0 & 16 & 12 & 8 & 0 \\ & 3 & 12 & 9 & 18 & 45 & -5 \end{array}$$

Let us assume the name of matrix is x

To access a particular element from the array we have to use two subscripts on for row number and other for column number the notation is of the form

X [i] [j] where i stands for row subscripts and j stands for column subscripts.

Below given are some typical two-dimensional array definitions

float table [50] [50];

char line [24] [40];

The first example defines tables as a floating point array having 50 rows and 50 columns. The number of elements will be 2500 (50 X50).

The second declaration example establishes an array line of type character with 24 rows and 40 columns. The number of elements will be (24 X 40) 1920 consider the following two dimensional array definition int values [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10. 11, 12, };

Thus the array values can be shown pictorially as

$$\begin{array}{cc|cccc} & & \multicolumn{4}{c}{\text{Column number}} \\ \hline & & 0 & 1 & 2 & 3 \\ \text{Row number} & 0 & 1 & 2 & 3 & 4 \\ & 1 & 5 & 6 & 7 & 8 \\ & 2 & 9 & 10 & 11 & 12 \end{array}$$

Values [0] [0] = 1	Values [0] [1] = 2	Values [0] [2] = 3	Values [0] [3] = 4
Values [1] [0] = 5	Values [1] [1] = 6	Values [1] [2] = 7	Values [1] [3] = 8
Values [2] [0] = 9	Values [2] [1] = 10	Values [2] [2] = 11	Values [2] [3] = 12

Here the first subscript stands for the row number and second one for column number. First subscript ranges from 0 to 2 and there are altogether 3 rows second one ranges from 0 to 3 and there are altogether 4 columns.

Alternatively the above definition can be defined and initialised as

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

Here the values in first pair of braces are initialised to elements of first row, the values of second pair of inner braces are assigned to second row and so on. Note that outer pair of curly braces is required.

If there are two few values within a pair of braces the remaining elements will be assigned as zeros.

Here is a sample program that stores roll numbers and marks obtained by a student side by side in matrix

```
main ( )
{
int stud [4] [2];
int i, j;
for (i =0; i < =3; i ++)
{
        printf ("\n Enter roll no. and marks");
        scanf ("%d%d", &stud [i] [0],  &stud [i] [1] );
        }
        for (i = 0; i < = 3; i ++)
        printf ("\n %d %d", stud [i] [0], stud [i] [1]);
        }
```

The above example illustrates how a two dimensional array can be read and how the values stored in the array can be displayed on screen.

ASSIMILATION EXERCISE

Q.1 Describe the array defined in each of the following statements

(a)	char name [30]	(d)	# define A 66
(b)	float c [6];		# define B 132
(c)	int params [5] [5]	(e)	double account [50] [20] [80]

Q.2 How can a list of strings be stored within a two-dimensional array ? How can the individual strings be processed? What library functions are available to simplify string processing?

Q.3 Write a C program that will produce to table of value of equation

$$y = 2\, e^{-0.1t} \sin 0.5t$$

Where t varies between 0 and 60. Allow the size of the t - increment to be entered as an input parameter.

CHAPTER - 7

FUNCTIONS

A computer program cannot handle all the tasks by it self. Instead its requests other program like entities - called ‘functions in C - to get its tasks done. A function is a self contained block of statements that perform a coherent task of some kind.

```
e.g
#include <stdio.h>
message();
  {
        message ( );
        printf (“\n Hello “);
  }
main ( )
  {
        message ( )
        printf (“\n I am in main “);
  }
output of the program will be
Hello
I am in main
```

Here main ( ) is the calling function and message is the called function. When the function message ( ) is called the activity of main ( ) is temporarily suspended while the message ( ) function wakes up and goes to work. When the message ( ) function runs out of statements to execute, the control returns to main ( ), which comes to life again and begins executing its code at the exact point where it left off.

```
The General form of a function is
function (arg1, arg2, arg3)
type arg1, arg2, arg3
{
statement 1;
statement2;
statement3;
statement4;
}
```

There are basically two types of functions

(i) Library functions e.g printf ( ), scanf ( ) etc
(ii) user defined function e.g the function message( ) mentioned above.

The following point must be noted about functions

(i) C program is a collection of one or more functions
(ii) A function gets called when the function name is followed by a semicolon for e.g.

```
main ( )
{
message ( );
}
```

(iii) A function is defined when function name is followed by a pair of braces in which one or more statements may be present for e.g.

```
message ( )
{
statement 1;
statement2;
statement 3;
}
```

(iv) Any function can be called from any other function even main ( ) can be called from other functions. for e.g.

```
main ( )
{
message ( );
}
message ( )
{
printf (" \n Hello");
main ( );
}
```

(v) A function can be called any number of times for eg.

```
main ()
{
message ( );
message ( );
}
message ( )
{
printf ("\n Hello");
}
```

(vi) The order in which the functions are defined in a program and the order in which they get called need not necessarily be same for e.g.

```
main ( );
{
message1 ( );
message2 ( );
}
message2 ( )
{
printf ("\n I am learning C");
}
message1 ( )
{
printf ( "\n Hello ");
}
```

(vii) A function can call itself such a process as called 'recursion'.

(viii) A function can be called from other function, but a function cannot be defined in another function. Thus the following program code would be wrong, since argentina is being defined inside another function main ( ).

```
main ( )
{
printf ("\n I am in main");
argentina ( )
{
printf {"\n I am in argentina");
}
}
```

(ix) Any C program contains at least one function

(x) If a program contains only one function, it must be main ( )

(xi) In a C program if there are more than one functional present then one of these functional must be main ( ) because program execution always begins with main ( )

(xii) There is no limit on the number of functions that might be present in a C program.

(xiii) Each function in a program is called in the sequence specified by the function calls in main ( )

(xiv) After each function has done its thing, control returns to the main ( ), when main ( ) runs out of function calls, the program ends.

WHY USE FUNCTIONS

Two reasons :

(i) Writing functions avoids rewriting the same code over and over. Suppose that there is a section of code in a program that calculates area of a triangle. If, later in the program we want to calculate the area of a different triangle we wont like to write the same instructions all over again. Instead we would prefer to jump to a 'section of code' that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.

(ii) Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided in to separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code in to modular functions also makes the program easier to design and understand.

(a) Functions declaration and prototypes

Any function by default returns an int value. If we desire that a function should return a value other than an int, then it is necessary to explicitly mention so in the calling functions as well as in the called function. e.g

```
main ( )
{
float a,b,
printf ("\n Enter any number");
scanf ("\% f", &a );
b = square (a);
printf ("\n square of  % f is % f", a,b);
}
square (Float x)
{
float y;
y = x * x;
return (y);
}
```

the sample run of this program is

```
Enter any number 2.5
square of 2.5 is 6.000000
```

Here 6 is not a square of 2.5 this happened because any C function, by default, always returns an integer value. The following program segment illustrates how to make square ( ) capable of returning a float value.

```
main ( )
{
float square ( );
float a, b;
printf (“\n Enter any number “);
scanf (“%f” &a);
b = square (a);
printf (“\n square of % f is % f, “ a, b);
}
float square (float x)
{
float y;
y= x *x;
return ( y);
}
sample run
Enter any number 2.5
square of 2.5 is 6.2500000
```

CALL BY VALUE

In the preceding examples we have seen that whenever we called a function we have always passed the values of variables to the called function. Such function calls are called ‘calls by value’ by this what it meant is that on calling a function we are passing values of variables to it.

The example of call by value are shown below ;

```
sum = calsum (a, b, c);
f =      factr (a);
```

In this method the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual argument in the calling function. the following program illustrates this

```
main ( )
{
int a = 10, b=20;
swapy (a,b);
printf (“\na = % d b = % d”, a,b);
}
swapy (int x, int y)
{
int t;
t = x;
x = y;
y = t;
printf ( “\n x = % d y = % d” , x, y);
}
```

The output of the above program would be;

```
x = 20 y = 10
a =10 b =20
```

CALL BY REFERENCE

In the second method the addresses of actual arguments in the calling function are copied in to formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them the following program illustrates this.

```
main ( )
{
int a = 10, b =20,
swapr (&a, &b);
printf (“\n a = %d b= %d”, a, b);
}
swapr (int *x, int * y)
{
int t;
t = *x
*x = *y;
*y = t;
}
```

The output of the above program would be

```
a = 20 b =10
```

ASSIMILATION EXERCISE

Q.1 Each of the following is the first live of a function definition explain the meaning of each

(a) float f (float a, float b) (c) Void f (int a)
(b) long f (long a) (d) char f (void)

Q.2 Write a function that will calculate and display the real roots of the quadratic equation: $ax^2 + bx + c = 0$ using the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Assume that a, b and c are floating - point arguments where values are given and that x1 and x2 are floating point variables. Also assume that $b^2 > 4 * a * c$, so that the calculated roots will always be real.

Q.3 Write a function that will allow a floating - point number to be raised to an integer power. In other words, we wish to evaluate the formula

$$y = X^n$$

where y and x are floating - point variables and n is an integer variable.

CHAPTER - 8

STANDARD LIBRARY STRING FUNCTIONS

For string handling C provides a standard set of library functions. Though there exists many such functions four of them will be discussed here.

The strcmp( ) Function

This function is used to check whether two strings are same or not. If both the strings are same it return a 0 or else it returns the numeric difference between the ASCII values of nonmatching characters e.q. the following program

```
# include <stdio.h>
main( )
{
        char string1 [ ] = “orange”;
        char string2 [ ] = “banana”;
        printf(“%d\n”, strcmp(string1, string2));
        printf(“%d\n”, strcmp(string2, “banana”);
        printf(“%d”, strcmp(string1, “Orange”));
        getch( );
        }
```

```
output
13
 0
32
```

In the first printf statement we use the strcmp( ) function with string1 and string2 as it arguments. As both are not equal (same) the function strcmp( ) returned 13, which is the numeric difference between “orange” and “banana” ie, between string2 and b.

In the second printf statement the arguments to strcmp() are string2 and “banana”. As string2 represents “banana”, it will obviously return a 0.

In the third printf statement strcmp( ) has its arguments “orange” and “Orange” because string1 represents “Orange”. Again a non-zero value is returned as “orange” and “Orange” are not equal.

strcpy( ) Function

The function copies one string to another for e.g. the following program

```
# include <stdio.h>
main( )
{
        char source [ ]  = “orange”;
        char target [20];
        strcpy(target, source);
        clrscr( );
        printf(“source: %s\n”, source);
        printf(“target:%s”, target);
        getch( );
        }
output will be
        source : orange
        target  :  orange
```

strcat( )

This function concatenates the source string at the end of the target string for e.g, “Bombay” and “Nagpur” on concatenation would result in to a string “Bombay Nagpur”. Here is an example of strcat( ) at work.

```
        main( )
        {
                char source [ ] = “Folks”;
                char target [30] = “Hello”;
                strcat(target, source);
                printf(“\n source string = %s”, source);
                printf(“\n target string = %s”, target);
        }
And here is the output
        source string = folks
        target string = Hello folks
```

strlen( )

This function counts the number of characters present in a string. Its usage is illustrated in the following program.

```
main( )
{
        char arr[ ] = “Bamboozled”
        int len1, len 2;
```

```
                len1 = strlen(arr);
                len2 = strlen("Hunpty Dumpty");
                printf("\n string = %s length = %d", arr, len1);
                printf("\n string = %s length = %d", "Humpty Dumpty", len2);
        }
```

The output would be

```
string = Bamboozled length=10
string = Humpty Dumpty length = 13
```

while calculating the length of the string it does not count '\0'.

ASSIMILATION EXERCISE

Q.1 Write a function xstrstr ( ) that will return the position where one string is present within another string. If the second string doesn't occur in the first string xstrstr ( ) should return a0.

For example in the string " some where over the rainbow", "over" is present at position"

Q.2 Write a program to encode the following strings such that it gets convert into an unrecognisable from. Also write a decode function to get back the original string.

CHAPTER - 9

DYNAMIC DATA STRUCTURES IN C

POINTERS

THE & and * Operators

A pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are used frequently in C, as they have a number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point. Pointers provide a way to return multiple data items from a function via function arguments to be specified as arguments to a given function.

Pointers are also closely associated with arrays and therefore provide an alternate way to access individual array elements.

Within the computer's memory, every stored data item occupies one or more adjacent memory cells. The number of memory cells required to store a data item depends on the type of data item. For example, a single character will be stored in 1 byte of memory integer usually requires two adjacent bytes, a floating point number may require four adjacent bytes.

Suppose V is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can be accessed if we know the location of the first memory cell. The address of V's memory location can be determined by the expression &V, where & is a unary operator, called the address operator, that evaluates the address of its operand.

Now let us assign the address of V to another variable, PV. Thus,

```
PV = & V
```

This new variable is called a pointer to V, since it "Points" to the location where V is stored in memory. Remember, however, that PV represents V's address, not its value. Thus, PV is called pointer variable.

address of V	value of V
PV	V

Relationship between PV and V (where PV = &V and V = *PV)

The data item represented by V can be accessed by the expression *PV where * is a unary operator, that operates only on a pointer variable. Therefore, PV and V both represent the same data item. Furthermore, if we write PV = &V and U = PV, then U and V will both represent the same values i.e., the value of V will indirectly be assigned to U.

Example :

```
int quantity = 179 ;
```

The statement instructs the system to find a location for the integer quantity and puts the value 179 in that location. Let us reassume that the system has chosen the address location 5000 for quantity.

$$\begin{tikzcd}
\text{Quantity} \arrow[r] & \text{Variable} \\
\boxed{179} \arrow[r] & \text{Value} \\
5000 \arrow[r] & \text{Address}
\end{tikzcd}$$

Representation of a variable

Remember, since a pointer is a variable, its value is also stored in the memory in another location.

The address of P can be assumed to be 5048.

Variable	Value	Address
Quantity	$\boxed{179}$	$\boxed{5000}$
P	5000	5048

Pointer as a variable

Declaring and initializing Pointers

Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

data type * Pt _ name

This tells the compiler three things about the variable Pt_name.

1. The * tells that the variable Pt_name is a pointer variable.
2. Pt_name needs a memory location.
3. Pt_name ponts to a variable of type data type.

Example : int * P ;

Declares the variable P as a pointer variable that points to an integer data type.

float * y ;

declares y as a pointer to a floating point variable.

Once pointer variable has been declared, it can be made to point to a variable using an assignment statement such as

P = & quantity ;

which causes P to point to quantity. P contains the address of quantity. This is known as pointer initialization.

Pointer expressions

Like other variables, pointer variables can be used in expressions. For example, if P1 and P2 are properly declared and initialized pointers, then the following statements are valid.

```
1)    Y      =    * P1  ;
2)    Sum    =    Sum + * P1  ;
3)    Z      =    S - * P2 / * P1  ;
4)    * P2   =    * P2 + 10  ;
```

Note that there is a blank space between / and * in the item 3 above.

If P1 and P2 are pointers then the expressions such as,

P1 + 4 ,   P2 - 2 ,   P1 - P2 ,   P1 ++ , — P2 are allowed

also,

```
Sum =Sum + *P2 ;
     P1 ++  ;
     - -P2  ;
P1 > P2
     P1 = = P2
     P1 ! = P2
```

are all allowed expressions.

The expressions such as,

```
        P1 / P2          or     P1 * P2      or     P1/3
```

are not allowed.

Pointer assignments

After declaring a pointer, pointer is assigned a value, so that it can point to a particular variable.

```
eg.         int * P  ;
            int i    ;
            P = & i ;
```

This is called assignment expression in which pointer variable P is holding the address of i.

Pointer arithmetic

Two pointer values can be added, multiplied, divided or subtracted together.

```
        eg.     if      int i  ;
                        int  j  ;
                        int * P , * q ;
        i = 5 ,  j = 10 ;
```

Now, various pointer arithmetic can be performed

```
        eg.             * j =  * i  + * j  ;
```

The value of variable j is changed from 10 to 15.

```
                        * j =  * j -  * i  ;
```

The value of variable j is changed from 10 to 5.

```
                        * i =  * i  ** j   ;
```

The value of i is changed from 5 to 50 ;
Consider another example,

```
        if there is array and a pointer is pointing to it
                        int i [10] ;
                        int * P ;
                        P = i   ;
```

Now, arithmetic operations like

```
                        P = P + 4  ;
```

Will move the pointer P from the starting address of the array to the fourth subscript of array.

Similarly, if P1 and P2 are both pointers to the same array, then P2 - P1 gives the number of elements between P1 and P2.

arithmetic operations like

P1/P2 or P1 x P2 or P/3 are not allowed.

Pointer Comparison

In addition to arithmetic operations, pointers can also be compared using the relational operators. The expressions such as

P1 > P2 , P1 = = P2 , P1 ! = P2 are allowed.

However, any comparison of pointers that refer to separate and unrelated variables make no sense. Comparisons can be used meaningfully in handling arrays and strings.

The dynamic allocation functions - malloc( ) and calloc( )

Most often we face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a company. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Such situations can be handled more easily and effectively by using what is called dynamic data structures.

DYNAMIC MEMORY ALLOCATION

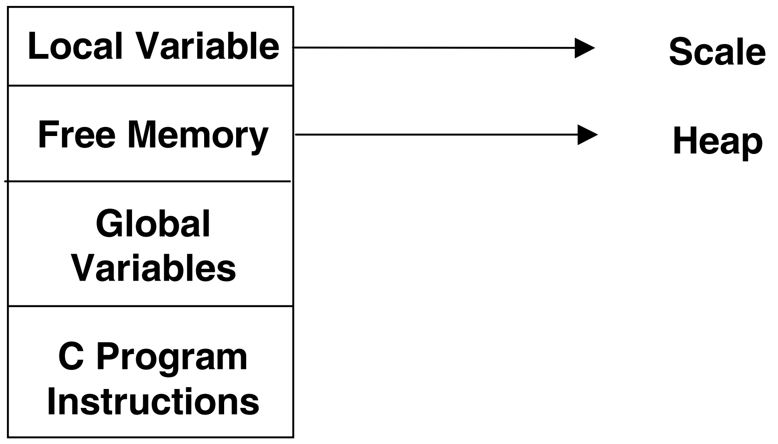
C language requires that the number of elements in an array should be specified at compile time. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages take the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as dynamic memory allocation. The library functions used for allocating memory are :

Function	Task
malloc ( )	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc ( )	Allocates space for an array of element, initializes them to zero and then returns a pointer to the memory.

Memory Allocation Process

Let us first look at the memory allocation process associated with a C program. Fig. below shows the conceptual view of storage of a C program in memory.



The program instructions and global and static variables are stored in a region known as permanent storage area and the local variables are stored in another area called stack. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. The free memory region is called the heap. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory “overflow” during dynamic allocation process. In such situations, the memory allocations functions mentioned above returns a NULL pointer.

ALLOCATING A BLOCK OF MEMORY

A block of memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form;

```
ptr = ( Cast type * ) malloc ( byte size ) ;
```

ptr is a pointer of type cast type. The malloc returns a pointer (of cast type) to an area of memory with size byte - size.

Example :

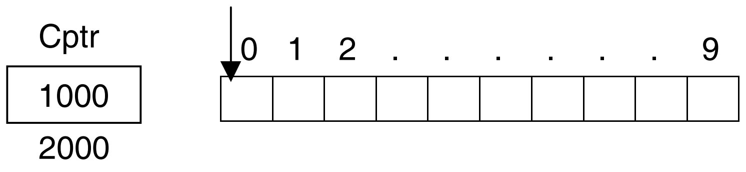
```
X = ( int * ) malloc ( 100 *sizeof ( int )) ;
```

On successful execution of this statement, a memory space equivalent to “100 times the size of an int” bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer X of type int.

Similarly, the statement

```
Cptr = ( char * ) malloc (10) ;
```

allocates 10 bytes of space for the pointer Cptr of type char



Remember, the malloc allocates a block of adjacent bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it foils, it returns a NULL. We should therefore check whether the allocation is successful before using the memory pointer.

Example :

Write a program that uses a table of integers whose size will be specified interactively at run time.

Program -

```
            # include <stdio. h>
            # include <stdlib.h>
            # define NULL O
            main ( )
                    {
                            int * P,  * table ;
                            int size ;
                    printf ( "\n What is the sizeof table ? " ) ;
                    scanf ( " % d", &size ) ;
                    printf ( "\n" ) ;

    if (( table = (int * ) malloc (size * sizeof (int)) = = NULL )
            {
            printf ("No space available \ n") ;
                    exit ( 1) ;
            }
    printf ("\n address of the first byte is % u\n", table );
    printf("\n Input table values");
    for ( P = table; P < table + size; P++ )
                    scanf ("%d", *P );

for ( P = table + size - 1; P > = table; P- - )
    printf ("%d is stored at address %u\n", *P, P );
}
```

Allocating Multiple Blocks of Memory

calloc is another memory allocation function that is normally used for requesting memory space at runtime for storing derived data types such as arrays and structures. While malloc allocates a single block of storage space, calloc allocates multiple blocks of storage, each of the same size, and then allocates all bytes to O. The general form of calloc is :

```
ptr = (Cast type * ) calloc ( n, elem_size );
```

The above statement allocates contiguous space for n blocks, each of size elem-size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

The following program allocates space for a structure variable.

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

                struct student
                        {
                        char name (25);
                        float age;
                        long int num;
                        } ;
                        typedef struct student record ;
                                record * ptr ;
                                int class_size = 30 ;

        ptr = ( record * ) calloc ( class_size, sizeof ( record )) ;

                - - - -
                - - - -
```

record is of type struct student having three number :
name, age and num. The calloc allocates memory to hold data for 30 such records. We should check if the requested memory has been allocated successfully before using the ptr. This may be done as follows:

```
if ( ptr == NULL )
{
printf ( “Available memory not sufficient”) ;
        exit ( 1 ) ;      }
```

POINTERS VS. ARRAY

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element suppose we declare an array X as follows :

```
static int X [ 6 ] = { 1, 2, 3, 4, 5, 6 } ;
```

Suppose the base address of X is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows :

ELEMENTS	x [0]	x[1]	x[2]	x[3]	x[4]	x[5]
VALUE	1	2	3	4	5	6
Address	1000	1002	1004	1006	1008	1010

BASE ADDRESS

The name X is defined as a constant pointer pointing to the first clement, x[0] and therefore the value of X is 1000, the location whose X[0] is stored. That is ;

X = & x[0] = 1000

If we declare P as an integer pointer, then we can make the pointer P to point to the array X by the following assignment :

P = X ;

This is equivalent to P = & X[0] ;

Now we can access every value of x using P++ to move from one element to another. The relationship between P and X is shown below :

```
P      =     & x[0] ( = 1000)
P+1    =     & x[1] ( = 1002)
P+2    =     & x[2] ( = 1004)
P+3    =     & x[3] ( = 1006)
P+4    =     & x[4] ( = 1008)
P+5    =     & x[5] ( = 1010)
```

The address of an element is calculated using its index and the scale factor of the data type. For instance,

```
address of X[3]   =    base address + (3 x  Scale factor of int)
                  =    1000  +  (3 x 2)     =      1006
```

When handling array, instead of using array indexing, we can use pointers to access array elements. Note that *(x+3) gives the value of X[3]. The pointer accessing method is more faster than array indexing.

POINTERS AND FUNCTIONS

When an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal fashion.

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling function using pointers to pass the address of variable is known as call by reference. The function which is called by reference can change the value of the variable used in the call.

```
eg.
        main ( )
          {
                int X ;
                X = 40 ;
                change ( & X ) ;
                printf ( " %d", X ) ;
           {
                change ( int * P )
           {
                * P = * P + 10 ;
            }
```

When the function change is called, the address of the variable X, not its value, is passed into the function change ( ). Inside change ( ), the variable P is declared as a pointer and therefore P is the address of the variable X. The statement,

```
* P =     * P + 10 ;
```

means add 10 to the value stored at address P. Since P represents the address of X, the value of X is changed from 50. Therefore, the output of the program will be 50 not 40.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function.

POINTERS TO FUNCTIONS

A function like a variable, has an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type ( * fp) (  ) ;
```

This tells the compiler that fp is a pointer to a function which returns type value.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

For example,

```
double (*P1)(  ), mul (  ) ;
        P1 = mul ;
```

declare P1 as a pointer to a function and mul as a function and then make P1 to point to the function mul. To call the function mul, we may now use the pointer P1 with the list of parameters. That is,

```
                    (*P1) (x,y)
is equivalent to    mul ( x,y )
```

FUNCTIONS RETURNING POINTERS

The way functions return an int, a float, a double or any other data type, it can even return a pointer. However, to make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function declaration. The following program illustrates this

```
  main ( )
{
    int      * P ;
    int      * fun ( ) ;
    P = fun ;
    printf ( “\n % Id”, P ) ;
    }
    int * fun ( )
    {
    int   i = 20;
    return (& i) ;
    }
```

In this program, function fun( ) is declared as pointer returning function can return the address of integer type value and in the body of the function fun ( ) we are returning the address of integer type variable i into P which is also integer type pointer.

POINTERS AND VARIABLE NUMBER OF ARGUMENTS

We use printf ( ) so often without realizing how it works correctly irrespective of how many arguments we pass to it. How do we write such routines which can take variable number of arguments? There are three macros available in the file “stdarg.h” called va_start, va_arg and va_list which allow us to handle this situation. These macros provide a method for accessing the arguments of the function when a function takes a fixed number of arguments followed by a variable number of arguments. The fixed number of arguments are accessed in the normal way, whereas the optional arguments are accessed using the macros va_start and va_arg. Out of these macros va_start is used to initialise a pointer to the beginning of the list of optional arguments. On the other hand the macro va_arg is used to initialise a pointer to the beginning of the list of optional arguments. On the other hand the macro va_arg is used to advance the pointer to the next argument.

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

                 main (   )
```

```
                    {
                            int max ;
                            max = findmax ( 5, 23, 15, 1, 92, 50 ) ;
                            printf ("\n max = % d", max ) ;
                            max = findmax (3, 100, 300, 29 ) ;
                            printf ("\n max = %d", max ) ;
                    }

                            findmax (int tot_num)
                            {
                            int max, count, num ;
                            va_list ptr ;
                    va_start ( ptr, tot_num ) ;
                    max = va_arg (ptr, int ) ;

            for ( count = 1 ; count < tot_num ; count + + )
                    {
                            num = va_arg (ptr, int ) ;
                            if ( num > max )
                                    max = num ;
                            }
                            return ( max ) ;
                    }
```

Here we are making two calls to findmax( ) first time to find maximum out of 5 values and second time to find maximum out of 3 values. Note that for each call the first argument is the count of arguments that are being passed after the first argument. The value of the first argument passed to findmax ( ) is collected in the variable tot_num findmax( ) begins with a declaration of pointer ptr of the type va_list. Observe the next statement carefully

```
va_start ( ptr, tot_num ) ;
```

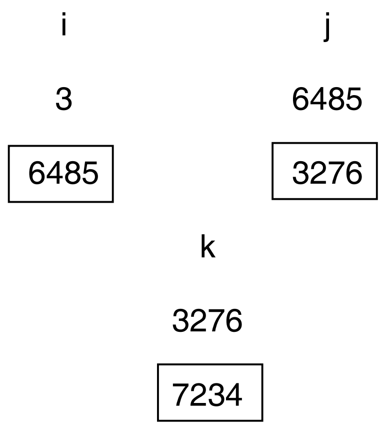
This statements sets up ptr such that it points to the first variable argument in the list. If we are considering the first call to findmax ( ) ptr would now point to 23. The next statement max = va_arg (ptr, int) would assign the integer being pointed to by ptr to max. Thus 23 would be assigned to max, and ptr will point to the next argument i.e. 15.

POINTERS TO POINTERS

The concept of pointers can be further extended. Pointer we know is a variable which contains address of another variable. Now this variable itself could be another pointer. These we now have a pointer which contains another pointer's address. The following example should make this point clear.

```
        main ()
            {
                int i = 3 ;
                int * j ;
                int * * k ;
                        j = & i ;
                        k = & j ;
printf ("\n address of i = % \d", & i );
printf ("\n address of i = % \d",  j );
printf ("\n address of i = % \d",  * k );
printf ("\n address of j = % \d", & j );
printf ("\n address of j = % \d",  k );
printf ("\n address of k = % \d", & k );
printf ("\n address of k = % \d", &k );

            }
```



In this program i is an integer type value, j is a pointer to this variable and k is another pointer type variable pointing to j.

i	j	k
3	6485	3276
6485	3276	7234

All the addresses are assumed addresses K is pointing to the variable j. These K is a pointer to pointer. In principle, there could be a pointer to a pointer's pointer, of a pointer to a pointer to a pointer's pointer. There is no limit on how far can we go on extending this definition.

ARRAY OF POINTERS

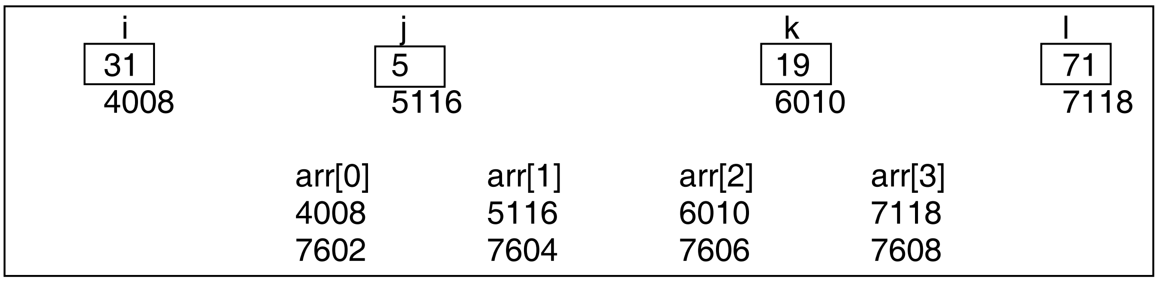
The way there can be an array of ints or an array of floats, similarly there can be an array of pointers. Since a pointer variable always contain an address, an array of pointers would be nothing but collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses. All rules that apply to an ordinary array apply to the array of pointers as well.

eg.

```
main ( )
  {
        int * arra [ 4 ];
        int      i = 31, j = 5, k = 19, L = 71, m;
        arra [0] = & i ;
        arra [1] = & j ;
        arra [2] = & k ;
        arra [3] = & l ;
for (m=0; m<=3 ; m+ +)
        printf (“\n% d”, * (arr[m])) ;
}
```

The output will be -

```
31
 5
19
71
```



ASSIMILATION EXERCISE

Q.1 What is the purpose of an automatic variable? What is its scope?

Q.2 A C program contains the following declaration:

static char * color[6] { “red”, “green”, “blue”, “white”, “black”, “yellow”}

(a) What is the meaning of color?

(b) What is the meaning of (color + 2)?

(c) What is the value of * color ?

(d) What is the value of * (color +2)?

CHAPTER - 10

STRUCTURES AND UNION

Introduction

A structure is a convenient tool for handling a group of logically related data items. structure help to organize complex data in a more meaningful way. It is powerful concept that we may after need to use in our program Design. A structure is combination of different data types using the & operator, the beginning address of structure can be determined. This is variable is of type structure, then & variable represent the starting address of that variable.

structure Definition

A structure definition creates a format that may be used to declare structure variables consider the following example.

```
struct book_bank
        {
        char title [20];
        char author [15];
        int pages;
        float price;
        };
```

Here keyword **struct** hold the details of four fields these fields are title, author, pages, and price, these fields are called **structure elements**. Each element may belong to different types of data. Here **book_bank** is the name of the structure and is called the structure tag.

It simply describes as shown below.

struct book-bank

Title	array of 20 characters
Author	array of 15 characters
Pages	integer
Price	float

]

The general format of a structure definition is as follows

```
struct teg_name
{
        data_type     member 1;
        data_type     member 2;
          - - -          - - -
          - - -          - - -
          - - -          - - -
}
```

Array of structures

Each element of the array itself is a structure. See the example shown below. Here we want to store data of 5 persons for this purpose, we would be required to use 5 different structure variables, from sample1 to sample 5. To have 5 separate variable will be inconvenient.

```
# include <stdio.h>
main( )
{
        struct person
        {
                char name [25];
                char age;
        };
        struct person sample[5];
                int index;
                char into[8];
        for( index = 0; index <5; index ++)
                {
                        print("Enter name;");
                        gets(sample [index]. name);
                        printf("%age;");
                        gets(info);
                        sample [index]. age = atoi (info);
                }
                for (index = 0; index <5; index++)
                {
                        printf("name = %5\n", sample [index]. name);
                        printf("Age = %d \n", sample [index]. age);
                                getch( );
                }
}
```

The structure type person is having 2 elements:

Name is an array of 25 characters and character type variable **age**

Using the statement:

struct person sample[5]; we are declaring a 5 element array of structures. Here, each element of sample is a separate structure of type person.

We, then defined 2 variable indexes and an array of 8 characters' **info.**

Here, the first loop executes 5 times, with the value of index varying from 0 to 4. The first printf statement displays. Enter name gets( ) function waits for the input string. For the first time this name you enter will go to sample[0]. name. The second printf display **age** the number you type is will be 5 stored as character type, because the member age is declared as character type. The function **atoi( )** converts this into an integer. atoi stands for **alpha to integer**. This will be stored in sample[0] age. The second **for** loop in responsible for printing the information stored in the array of structures.

structures within structures:-

structure with in a structure means nesting of structures. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
        {
                char name[20];
                char department[10];
                int basic_pay;
                int dearness_allowance;
                int city_allowance;
        }
                employee;
```

This structure defines name, department, basic pay and 3 kinds of allowance. we can group all the items related to allowance together and declare them under a substructure as shown below:

```
struct  salary
{
        char name [20];
        char department[10];

        struct
        {
        int dearness;
        int hous_rent;
        int city;
                }
                allowance;
        }
        employee;
```

The salary structure contains a member named **allowance** which itself is a structure with 3 members. The members contained in the inner, structure namely dearness, hous_rent, and city can be referred to as :

```
employee allowance. dearness
employee. allowance. hous_rent
employee. allowance. city
```

An inner-most member in a nested structure can be accessed by chaining all the concerned. structure variables (from outer-most to inner-most) with the member using dot operator. The following being invalid.

```
employee. allowance          (actual member is missing)
employee. hous_rent         (inner structure variable is missing)
```

Passing a structure as a whole to a Function:

structures are passed to functions by way of their pointers. Thus, the changes made to the structure members inside the function will be reflected even outside the function.

```
# include <stdio.h>
typedef  struct
        {
                char  *name;
                int acc_no;
                char acc_types;
                float balance;
        }       account;
main( )
{
        void change(account *pt);
        static account person = {“chetan”, 4323, ‘R’, 12.45};
        printf(“%s      %d %c %2.f \n”, person. name,
                        person.acc_type, person. acc_type,
                        person. balance);
change(&person);
        printf(“%s %d  %c  %2.f \n”, person.name, person.acc_type,
                        person.acc-type, person. balance);
        getch( );
}
        void change(account *pt)
{
        pt - > name =” Rohit    R”;
        pt - > acc_no = 1111;
        pt - > acc_type = ‘c’;
        pt - > balance = 44.12;
}
```

output

```
chetan      4323      R     12.45
Rohit  R    1111      c     44.12
```

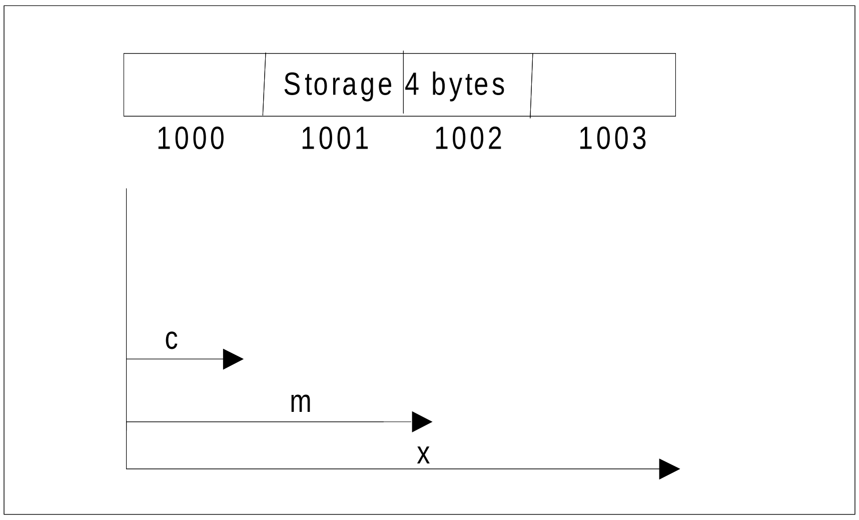
UNIONS

Unions, like structure contain members, whose individual data types may vary. This is a is major distinction between them in terms of storage .In structures each member has its own storage location, where as all the members of a union use the same location.

Like structures, a union can be declared using the keyword **union** is follows:

```
union item
        {
        int m;
        float x;
        char c;
        } code;
```

This declares a variable **code** of type **union** item. The union contains item members, each with a different date type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size



The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. The above figure shown how all the three variables share the same address, this assumes that a **float** variable requires 4 bytes of storage.

To access a union member, we can use the same syntax that we as for structure members, that is,

```
code. m
code. x
code. c          are all valid
```

When accessing member variables, we should make sure that we are accessing the member whose value is currently in storage. For example

```
code. m = 565;
code. x = 783.65;
printf("%d", code. m); would produce erroneous output.
```

```
# include <stdio.h>
main( )
{
union
        {
        int one;
        char two;
        } val;
val. one = 300;
printf("val. one = %d \n", val. one);
printf("val. two = %d \n", val. two);
        }
```

The format of union is similar to structure, with the only difference in the keyword used.

The above example, we have 2 members **int one** and **char two** we have then initialised the member '**one**' to 300. Here we have initialised only one member of the **union**. Using two **printf** statements, then we are displaying the individual members of the union **val** as:

```
val. one = 300
val. two = 44
```

As we have not initialised the char variable **two,** the second **printf** statement will give a random value of 44.

The general formats of a union thus, can be shown as.

```
union tag {
        member 1;
        member 2;
          - - -
          - - -
        member m;
                };
```

The general format for defining individual union variables:

Storage-class Union tag variable 1, variable 2,........., variable n;

Storage-class and **tag** are optional variable 1, variable 2 etc, are **union** variable of type **tag.** Declaring **union** and defining variables can be done at the same time as shown below:

```
Stroage-calss union tag {
          member 1;
          member 2;
            - - -
            - - -
            - - -
          member m;
          } variable 1, variable 2,   - - -  , variable n;
```

ASSIMILATION EXERCISE

Q.1
```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
};
```
The above structure requires __________ bytes of memory.

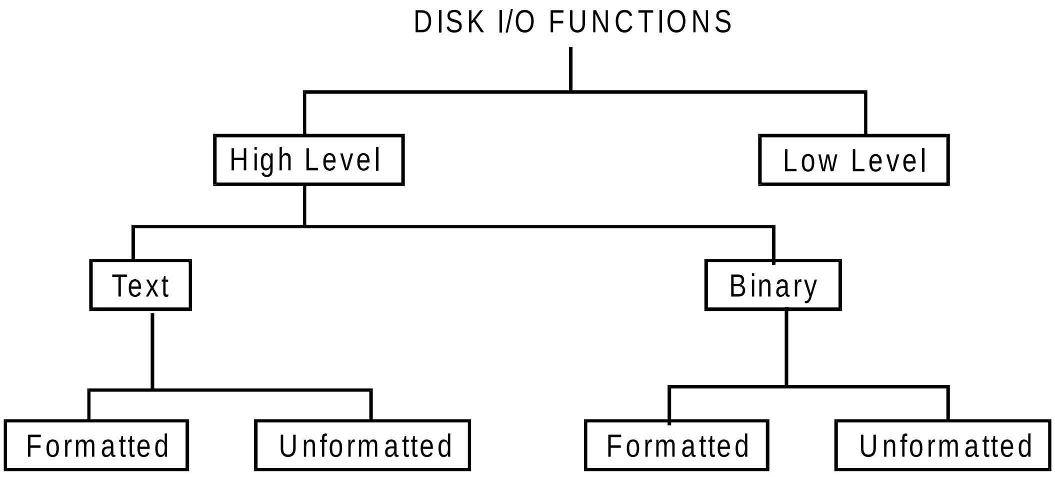
Q.2
```
union book_bank
{
char title[20];
char author[15];
int pages;
float price;
};
```
The above union requires __________ bytes of memory.

Q.3
```
struct marks
{
int maths;
int physics;
int chemistry;
char name[30]
};
```
Sort the above structure for 10 students according to the total of marks .

CHAPTER - 11

DISK I/O FUNCTIONS

scanf( ), printf( ), getch ( ) etc which we have studied were console I/O functions. Let us now turn our attention to disk I/O. Disk I/O operations are performed on entities called files. The brief categorisation of Disk I/O functions is given below



From above we can see that the file I/o functions are further categorised in to text and binary. This classification arises out of the mode in which a file is opened for input or output. Which of these two modes is used to open the file determines:

(a) How new lines (\n) are stored
(b) How end of file is indicated
(c) How numbers are stored in the file

Opening a file

We make the following declaration before opening a file

```
FILE * fp
```

Now let us understand the following statements,

```
FILE  * fp;
fp = fopen (“PR1.C”,”r”);
```

fp is a pointer variable which contains address of the structure FILE which has been defined in the header file “stdio.h”.

fopen( ) will open a file “PRI.C” in read mode. fopen( ) performs three important tasks when you open the file in
“r” mode:

(i) Firstly it searches on the disk the file to be opened.

(ii) If the file is present, it loads the file from the disk in to memory. Of course if the file is very big, then it loads the file part by part.

If the file is absent, fopen( ) returns a NULL. NULL is a macro defined in “stdio.h” which indicates that you failed to open the file.

(iii) It sets up a character pointer (which is part of the FILE structure) which points to the first character of the chunk of memory where the file has been loaded.

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

        FILE *fp;
        fp = fopen(“PRI.C”, “r”);
        if (fp= = NULL)
        { puts (“ cannot open file”);
           exit( );
        }
}
```

Closing the file

The closing of the file is done by fclose( ) through the statement,

```
fclose ( fp );
```

File Opening Modes

The “r” mode mentioned above is one of the several modes in which we can open a file. These are mentioned below:

(i) “r” Searches the file. If the file exists, loads it in to memory and sets up a pointer which points to the first character in it. If the file doesn’t exist it returns NULL.

(ii) “w” Searches file if the file exists it contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

Operations possible - writing to the file.

(iii) “a” Searches file. If the file exists, loads it in to memory and sets up a pointer which points to the first character in it. If the file doesn’t exist a new file is created. Returns NULL, if unable to open file. Operations possible - Appending new contents at the end of file.

(iv) “r+” Searches file. If it exists, loads it in to memory and sets up a pointer which points to the first character in it. If file doesn’t exist it returns NULL.

Operations possible - reading existing contents, writing new contents, modifying existing contents of the file.

(v) “w+” Searches file. If the file exists, it contents are destroyed. It the file doesn’t exist a new file is created. Returns NULL if unable to open file. Operations possible - writing new contents, reading them back and modifying existing contents of the file. “a+” Searches if the file exists, loads it in to memory and sets up a pointer which points to the first character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file. Operations possible - reading existing contents, appending new contents to the end of file. Canot modify existing contents.

Reading from a File(Unformatted Character)

To read the file’s contents from memory there exists a function called fgetc( )
e.g ch = fgetc(fp);

Writing to a File(Unformatted Character)

There exists a fputc( ) function that writes to the file fputc(ch, ft);
here value of ch variable will be written to file whose file pointer is ft.

Closing the File

When we have finished reading or writing from the file, we need to close it. This is done using the function fclose( ) through the statement.

fclose(fp); fp is file pointer here.

e.g Afile - copy program here is a program which copies the contents of one file in to another

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

```
        {
                FILE *fs, *ft;
                char ch;
                fs = fopen("pr1.c", "r");
                if(fs = = NULL)
                        {
                                puts("canot open source file")'
                                exit( );
                        }
                ft = fopen ("pr2.c", "w");
                it (ft = =NULL)
                        {
                                puts("canot open target file");
                                fclose(fs);
                                exit( );
                        }
                        while (1)
                        {
                                ch = fgetc(fs);
                                if(ch = = EOF)
                                        break;
                        else
                                fputc(ch,ft);
                        }
                        fclose(fs);
                        fclose(ft);
                        }
```

The fprintf and fscanf functions(Formatted I/o)

The general form of fprintf is

```
        fprintf(fp, "control string", list)
eg      fprintf(f1, "%s %d %f", name, age, 7.5);
```

Name is an array variable of type char and age is an int variable. The function fprintf will cause the value name age and 7.5 to be written to the file pointed by variable f1.

The general format of fscanf is

```
        fscanf(fp, "control string", list);
```

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the control string. eg.

```
        fscanf(f2, "%s %d", item, & quantity);
```

Like scanf fscanf also returns number of items that are successfully read. When the end of the file is reached, it returns the value EOF.

fseek function

fseek function is used to move the file position to a desired location within the file. It takes the following form:

fseek(file ptr, offset, position)

File ptr is a pointer to the file concerned, offset is a number variable of type long and position is an integer number. The offset specifics the number of positions(bytes) to the moved from the location specified by position.

The position can take one of the following three values

Values	**Meaning**
0	Beginning of file
1	Current position
2	End of file

offset may be positive meaning move forwards or negative meaning move backwards. The following examples illustrate the operation of the fseek function:

fseek(fp,0L,0)	Go to beginning
fseek(fp, 0L, 1)	Stays at current position
fseek(fp, 0L, 2)	Go to end of the file, past the last character of the file
fseek(fp, m, 0)	Move to (m+1)th byte in the file
fseek(fp, m, 1)	Go forwared by m bytes
fseek(fp, -m, 1)	Go backward by m bytes from the current position
fseek(fp, - m, 2)	Go backward by m bytes from the end

ftell

ftell takes a file pointer and returns a number of type long that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form

```
n = ftell(fp);
```

n would give the relative offset(in bytes) of the current position. This means that n bytes have already been read <u>(or written)</u>.
rewind takes a file pointer and resets the position to the start of the file.
<u>for</u> e.g.

```
rewind(fp);
n = ftell(fp);
n would return 0
```

Binary Mode

A file can be opened in binary mode as follows:

```
fp = fopen("Poem. txt", "rb")
```

Here fp is file pointer
Poem. txt is file name
rb denotes that file is opened in binary mode for read operation.

Text Mode Versus Binary Mode:-

(i) New Lines:-

In text mode, a newline character is converted into the carriage return -linefeed combination before being written to the disk. Like wise, the carriage return-line feed combination on the disk is converted back in to a newline when the file is read by a c program. However if file is opened in binary mode, as opposed to text mode, these conversions will not take place.

(ii) End of File:-

The difference is that in text mode when end-of-file is detected a special character whose ascil value is 26, is inserted after the last character in the file to mark the end of file. If this character is detected at any point in the file, the read function will return the EOF signal to the program.

As against this, there is no such special character present in the binary mode files to mark the end of file. The binary mode file keeps track of the end of file from the number of characters present in directory entry of the file.

Text Mode:-

The only function available for storing in a disk file is the fprintf( ) in text mode. Here numbers are stored as string of characters when written to the disk. These 1234, even though it occupies two bytes in memory, when transferred to the disk using fprintf( ), it would occupy four bytes, one byte per character. Similarly the floating point number 1234.56 would occupy 7 bytes on disk. These, numbers with more digits would require more disk space.

In binary by using the functions (fread( ) and fwrite( )) numbers are stored in binary format. It means each number would occupy the same number of bytes on disk as it occupies in memory.

Command Line Arguments

Two special identifiers, argc and argv are used to pass to main( ) the number of command line arguments and pointers to each argument we have to set up main( ) as follows.

```
main(int argc, char*argv[ ])
```

argc will then provide the number of command line arguments including the command itself- so argC its never less than 1.

The argv is an array of pointer to char or equivalently an array of strings. Each of argv[0], argv[1],... up to argv[argc-1] is a pointer to command line argument, namely a NULL terminated string. The pointer argv[argc] is set to NULL to mark the end of the array.

Suppose these is a program Vkcpy in which main contains argument argc, and argv means skeleton of program is as follows

```
main(int argc, char * argv[ ])
{       __________
        __________
        __________
        __________
        __________
}
```

suppose we now execute the program by typing VK COPY Hellow.C Hello.CBK Here argc = 3(command plus 2 arguments

```
argv[0]     points to "C:\VKCPY\0"
argv[1]     points to "HELLO.C\0"
argv[2]     points to "HELLO.CBK\0"
argv[3]     is NULL
```

ASSIMILATION EXERCISE

Q.1 Describe the different ways in which data files can be categorized in C.

Q.2 What is the purpose of library function feof ? . How the feof function be utilized within a program that updates an unformatted data file

CHAPTER - 12

GRAPHICS FEATURES IN C

OBJECTIVES

At the end of this chapter, the user will be able to

* Differentiate graphics and text mode
* Play with different colors in the text mode
* Draw graphical figures in graphics mode

INTRODUCTION

In our day today life, the graphical figures such as tables, chairs, containers etc. are seen more than the text. Hence it is thought to implement the graphical features in computers which are widely used in many applications. The use of graphics is intended an adjunct to text rather than being a substitute for text. Many computer languages such as BASIC, pascal, C etc., supports the graphics features.

The applications of graphics are in the field of computer Aided Drafting (CAD), Computer Aided Engineering (CAE), Computer Aided Instruction (CAI), Computer Aided Software Engineering (CASE),

GRAPHICS IN C

ANSI standard C does not define any text screen of graphics functions because of the capabilities of diverse hardware environments. But Turbo C version. 1.5 and higher support extensive screen and graphics support facilities. In this chapter the features supported by Turbo C are discussed.

MODE

Basically there are two different modes, namely text mode and graphics mode. in text mode, as the name states, it is possible to display or capture only text in terms of ASCII. But in Graphics mode, any type of figures can be displayed, captured and animated. Let us discuss these features now.

(i) *TEXT MODE*

In Text mode, it is possible to handle, only the text which are present in ASCII. Text mode display can be in two forms as 25 rows of 80 columns or 25 rows of 40 columns. The elements of text are characters. Text mode can have 2 colors in monochrome monitor and 16 colors in color monitor.

The printf ( ) function helps the text to be displayed on the monitor. but the following functions help to present the text in an attractive manner. To execute these special functions conio.h should be included in the program.

(i) textmode (int mode);

This function sets the number of rows and columns of the screen. the variable mode can take the values 0, 1, 2, or 3.

0	represents 40 column black and white
1	represents 40 column color
2	represent 80 column black and white
3	represents 80 column color

Example :

```
/* sets to 80 columns color mode */
textmode (3);
```

(b) clrscr ( );

This function clears the entire screen and locates the cursor in the top left corner (1,1,)

(c) gotoxy (int x, int y);

This function positions the cursor to the location specified by x and y. x represents the row number and y represents the column number

Example:

```
/* The following statement positions the cursor to 3rd row 4th column*/
gotoxy (3, 4);
```

(d) textbackground (int color);

This function gives the facility to Change the background color of the text mode. The valid colors for the CGA are from 0 to 6 namely BLACK, BLUE, GREEN, CYAN, RED, MAGENTA and BROWN.

Example:

```
/* background color is set to red*/
int col = 4;
textbackground (col)
```

(e) textcolor (int color);

This function sets the color in which the subsequent text is to be displayed. The supported colors are numbered from 0 to 15 and 128 for blinking.

Example /* the subsequent display of text will be in brown*/

```
int colr = 6;
textcolor (colr);
```

(f) delline ( );

It is possible to delete a line of text and after the deletion all the subsequent lines will be pushed up by one line.

Example:

```
\* deletes the 5th line *\
gotoxy (5,4);
delline ( );
```

(g) insline ( );

Inserts a blank line at the current cursor position.

Example:

```
gotoxy (3,5);
insline ( );
```

Example Program:

```
/* Program to display text using special functions*/
# include <conio.h>
main ( )
{
int n,m,;
/* clears the screen */
clrscr ( );
/* sets the text mode to 80 columns color*/
textmode (3);
/* SETS THE TEXT COLOR*/
textcolor (4);
/* sets the text background color */
textbackground (2);
/* Positions to 5th row and 14th column*/
gotoxy (5,15);
printf (“Enter two numbers:);
scanf (“%d %d”, &n, &m);
gotoxy (10,15);
printf (“Entered numbers are %d and %d \n\n”, n,m);
}
```

(ii) GRAPHICS MODE

In this mode it is possible to display text as well as graphical figures. The basic element of the graphical pictures is picture element which is also called as pixel. The resolution of the monitor is measured in terms of pixels and it varies with respect to the type of the monitor.

The monitor type can be monochrome, CGA, EGA, VGA, etc. Depending on the monitor type and resolution, the graphics pictures and colors vary. Let us see some of the graphics functions now. To execute these function “graphics.h” file should be included in the c program.

(a) initgraph (int *driver, int *mode, char *path);

This function is used to initialise the graphics system and load the appropriate specified graphics driver and the video mode used by the graphics function. The path is to specify the place in which the graphics driver files are available.

The driver is specified as 0 to 10 representing the monitor types DETECT, CGA, MCGA, EGA, EGA64, EGAMONO, IBM8514, HERCMONO, ATT400, VGA and PC3270.

The mode specifies the resolution of the video. It can take the values as follows.

MODE	VALUE	RESOLUTION
CGAC0	0	320 X 200
CGAC1	1	320 X 200
CGAC2	2	320 X 200
CGAC3	3	320 X 200
CGAH1	4	640 X 200

The path specifies the system path from where the graphics driver files are to be searched. If the files are in current directory then the path is a null string.

Whenever any graphics figure has to be drawn this intigraph ( ) function should be used to initialise the graphics mode on the video.

Example:

```
        int driver, mode;
        driver =1;
        mode = 4;
        initgraph (&driver, &mode, " ");
```

/* initialise driver to CGA mode to high resolution mode and the driver files are expected to be present in the current directory */

(b) restorecrtmode ( );

This function restores the screen to the mode that it had prior to the call to initgraph ( ).

Example :

```
        restorecrtmode ( ); \* restores back*\
```

(c) setpalatte (int index, int color);

This function chooses an index for palette and matches the color with the index and this color sets the background color of CGA mode.

Example :

```
        setpalette ( 0, GREEN);
        \* Set the Background to Green*\
```

(d) Putpixel (int x, int y, int color);

This function illuminates the pixel represented by x and y coordinates in the color represented by color

Example :

```
        putpixel (10, 20, RED);
        /* illuminates the pixel (10, 20) in red */
```

(e) getpixel (int x, int y);

This function returns the color in which the pixel (x,y) is illuminated

Example

```
        color = getpixel (10, 20)
        /* returns the color of the pixel (10, 20) */
```

(f) moveto (init x, int y);

This function moves cursor to (x, y) position.

Example

```
moveto (100, 150);
/* positions the cursor to (100, 150)*/
```

The following program illustrates the application of the above discussed functions.

It sets the graphics mode, illuminates a pixel in a colour and stores the colour of the pixel in a variable.

EXAMPLE PROGRAM 95

```
# include <graphics.h>
main ( )
{
int driver, mode, colr;
driver = CGA;
mode = CGAC3
/* driver and modes are set */
initgraph (&driver, &mode, " ");
/*graphics mode is initialized*/
setpalette (0, RED); /* sets palette*/
moveto ( 75, 100); /* moves to (75, 100)*/
putpixel (75, 100, RED); /* illuminates pixel (75, 100) in red */
colr = getpixel (75, 100);
/* colr holds the color number of RED which is the color of the pixel (75,100)*/
roatorecrtmode ( );
/* restores back to the mode prior to the invokation of graphics initialization*/
}
```

(g) lineto (int x, int y);

This function draws a line from the current cursor position to (x,y)

Example

```
lineto (150, 175);
/* draws line from current cursor to (150, 175)*/
```

(h) line (int x1, int y1, int x2, int y2)

This function draws a line from (x1,y1) to (x2, y2)

Example

```
line (10, 50, 10, 100);
/* draws a vertical line */
```

(i) bar (int x1, int y1, int x2, int y2);

This draws a rectangle with diagonal from (x1, y1) to (x2, y2);

Example

```
bar (10, 25, 100, 75);
/* Draws a rectangle with diagonal from (10,25) to (100, 75)*/
```

(j) bar3d (int x1, int y1, intx2, int y2, depth, topflag);

This function provides a 3 dimensional view of rectangle boxes. This draws a rectangle with diagonal from (x1, y1) to (x2, y2) and with depth specified in the variable depth. If the top flag is non zero, a top is added to the bar, and hence 3-Dimensional view is possible. Otherwise the bar has no top.

The following program illustrates the application of the functions discussed above.

It displays rectangle along with two diagonals.

EXAMPLE PROGRAM 96

```
# include <graphics.h>
main ( )
{
int driver = CGA, mode, colr;
mode = CGAC3;
/* driver and modes are set*/
intigraph (&dirver, &mode, " ");
/* graphics mode is initialized */
line (50, 40, 100, 80);
/*draws a line from (50,40) to (100, 80)*/
moveto (51, 41);
lineto (99, 79);
/* draws left diagonal */
moveto (99, 41);
lineto (51, 79);
/* draws right diagonal*/
restorecrtmode ( );
/* restores back to old mode */
}
```

(k) circle (int x, int y, int radius);

This function draws a circle centered at (x, y) with the radius specified by the variable radius (in terms of number of pixels) in the current drawing color.

Example

```
circle (100, 100, 50);
/* draws a circle with centre (100,100) and radius 50 */
```

(l) **arc (int x, int y, int start, int end, int radius);**

This function draws an arc of the circle with radius as specified in the variable radius and with centre at (x, y). Start and end are given in degrees to mention the portion of the circle that form the arc.
Example

```
arc (100, 100, 0, 90, 20);
/* draws the first quarter of the circle, arc with centre (100,100) and radius 20 pixels*/
```

(m) pieslice (int x, int y, int start, int end, int radius);

This function works in the same way as arc, but it provides the 2 radii from centre to start and end.
Example

```
pieslice ( 100, 100, 90, 180, 50);
\* draws a pie with centre at (100, 100) and radius 50 pixels. The second quarter of the
circle is presented *\
```

(n) ellipse (int x, int y, int start, int end, int xrad, int yrad);

This function draws an ellipse with xrad as radius along x axis and yrad as radius along y axis. The start and end should be 0 to 360 for full ellipse. Arcs of ellipse can also be drawn by changing the start and end values as used in the function arc ( ).
Example

```
ellipse (100, 50, 0, 360, 30, 15);
/* draws full ellipse with centre at (100, 50), xradius 30 and yradius 15 */
```

(o) setcolor (int color);

This function sets the drawing color as specified by the variable color. The subsequent graphical figures will be in the color specified by the variable color.
Example

```
setcolor (4); /* sets the color to red */
```

The following program illustrates the application of the functions discussed above.

It displays a circle and ellipse drawn in different colors.

Example program 97:

```
# include <graphics.h>
main ( );
{
int driver, mode, colr;
driver = CGA;
mode = CGAC3;
/* driver and modes are set */
initgraph (&driver, &mode, “ “);
/* graphics mode is initialized*/
setcolor (3);
circle (50, 50, 25);
/* draws circle of radius 25 pircell and centre at (50, 50)*/
ellipse (50, 50, 0 360, 75, 50);
/* draws full ellipse with centre at (50, 50), x radius as 75 and y radius as 50 pixels in
color 5*/
restorecrtmode ( );
}
```

(p) setfillstyle (int pattern, int color);

This sets the pattern in which the closed boundaries can be filled and the color to fill the boundary. The fixed patterns are represented by the values from 0 to 11
Example

```
setfillstyle (2, 5);
/* sets pattern to 2 and fill color as 5 */
```

(q) setfillpattern (char *pattern, int color);

This function helps the tiling or halftoning. This sets the pattern in which subsequent boundaries can be filled. The pattern should be of 8 bits length.
Example

```
char p [9] = “ /2AB34CD
*p = “12AB 34CD”;
setfillpattern (p, RED)
/* set the pattern top */
```

(r) floodfill (int x, int y,int border);

Starting from the pixel (x,y) this function fills the area bounded by the color border. If the boundary is not closed, then the entire screen is filled.

Example

```
setcolor (3);
circle (100, 100, 50);
floodfill (100, 100, 3);
/* fills the circle bounded by 3 */
```

(s) setlinestyle (int style, unsigned pattern, int width);

This function determines the way the subsequent lines should look. Style can be any one of SOLID_LINE, DOTTED_LINE, CENTRE_LINE, DASHED _LINE, USERBIT_LINE. The width can be NORM_WIDTH or THICK_WIDTH. The pattern should be specified for USERBIT_LINE.
Example

```
unsigned p = 11101101;
setlinestyle (CENTRE_LINE, 0, NORM_WIDTH);
lineto (100, 150);
/* Draws a styled line from current position to (100, 150) */
setlinestyle (USERBIT _LINE, p, NORM_WIDTH);
lineto (200, 190);
/* Draws a styled line specified by the user from current position to (100, 190)*/
```

(t) outtext (string);

In graphics mode user text can also be printed at the specified location by outtext.
Example

```
outtext (10, 10, “graphics mode text”);
/* displays text at (10,10)*/
```

(u) cleardevice ( );

This function clears the screen in graphics mode.

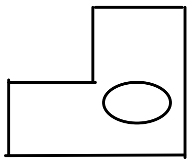
Example Program 98:

```
/* Program to display concentric circles*/
# include <graphics.h>
main ( )
{
        int driver, mode, i;
        driver = CGA;
        mode = CGAC3
        /* driver and modes are set */
        initgraph (&driver, &mode, “ “)
        /*graphics mode is initialized*/
        setcolor (3);
        for ( i = 25, i < 150, i+= 25)
        circle (50, 50, i);
        /* draws concentric circle of radius i pixels and centre at (50, 50) in color 3 */
        restorecrtmode ( );
        /* restores back to old mode */
}
```

ASSIMILATION EXERCISE

Q.1 What is difference between textmode and graphics mode

Q.2 Write a program to generate the following figure.



Q.3 Write C program using graphics to draw the following figures :