Programming Using C

UNIT-1

Introduction to computers

A computer is an electronic device capable of manipulating numbers and symbols under the control of a set of instructions known as computer program.

They are different stages of computers (generation)

1.First Generation Computers

1. Vacuum tubes were used which produce more heat
2. Speed of computing was measured in milliseconds
3. Limited storage capacity
4. punched cards were used for I/O operation

2. Second – Generation Computers

1. Transistors and diodes were used.
2. Speed of computing was measured in microseconds
3. Consider about reduction of heat
4. Remarkable improvement in reliability
5. Storage capacity was increased
6. Magnetic tapes were used instead of punching cards.

3. Third Generation Computers

1. Integrated Circuits were used.
2. Speed is measured in nanoseconds
3. Occupied less space.
4. devices like visual display unit for I/O devices

4. Fourth – Generation Computers

1. Use of micro processor chip
2. Speed was measured in nano and picoseconds

3. Occupied less space
4. Commonly available as personal computers
5. Mini & micro Computers are developed from micro-processor

5. Fifth – Generation Computers:

1. Use of super large-scale integration (SLSI) chip in computer (super computers)
2. Capable of performing millions of instructions per seconds (MIPS)
3. Processing speed is high.
4. Use of RICS (reduced instructions set computing) for processing
5. Super computers are expensive.

Types of Computers

1. Mainframe Computers
2. Mini Computers
3. Micro Computers
4. Super Computers

Mainframe Computers work at a high speed, and have a high storage capacity

Mini Computers are medium and powerful Computers.

Micro Computer are the commonly used as general purpose Computer

Data Storage in a Computer

1. 4bits = 1 Nibble
2. 8bits = 1 byte
3. 1024 bytes = 1k or 1kb (kilobyte)
4. 1024KB = 1MB (mega byte)
5. 1024MB = 1GB (Gega byte)
6. 1024GB = 1TBC Terabytes

Organization of Computer:

1. Arithmetic and Logical unit
2. Memory unit
3. Control unit
4. Input unit
5. Output unit

The Input and Output units are used to receive and display Inputs & Solutions

Common i/p & o/p devices : Keyboard, mouse, monitor, printer

The CPU (Central Processing Unit) Consists of.

1. ALU (Arithmetic Logic Unit)
2. CU (Control Unit)
3. MU (Memory Unit)

1. The Control Unit Controls all the activities of the Computer. It sends commands and control signals and finds the sequence of instruction to be executed.
2. Memory Unit is the place where all input data and results are stored. Computer memory is also available in the form of Random Access Memory (RAM)
3. ALU Consists of CKTs for arithmetic operations(+,-,*,/) and logical operations (<,>,>=,<=,==,!=)

Connected components of CPU are called peripherals

Input devices	Output devices
1. Keyboard	1. Printer
2. Mouse	2. Monitor
3. Joystic	3. Dot Matrix Printer
	4. Laser printers
	5.LCD

Storage Devices :

1. Floppy disk
2. Hard disk
3. Compact disk

Computer Main Memory :

Primary memory	RAM (Random Access memory)
Secondary memory	ROM (Read only memory)
	Hard disk

RAM : It is a temporary storage medium in a computer. The data to be processed by the computer are transferred from a storage devices or a keyboard to RAM results from a executed program are also stored in RAM. The data stored will be erased when the computer is off.

ROM (Read only Memory) : This is a non-volatile or data storage medium which stores start up programs (operating systems). This essentially stores the BIOS (Basic Input Operating System)

Note : Basically Computer System components communicate it binaries as (0‟s & 1‟s, 0 refers OFF state,1 refer ON state)

Languages of different Generation Computer.

1. First – Generation Language : All the instructions are in the binary form and are referred to as machine level or low level language (LLL). It is very difficult to read the instructions written in binary
   Eg : 00110101011101110001, 101100001010101

2. Second – Generation Language: all the instruction are in the forms of mnemonics. The symbolic instruction language called as Assembly Language. All the symbolic instructions are converted into binaries with the help of translator called Assembles. ASCII (American Standard Code For Information Interchange) is commonly used for translation of source Program into object program

Source Program	Eg : ADD A, B, R,	
	More R,S	
Assembler	Translated by Assemble	
Object Program	0101	10101010
	0100	00001101

3. Third – Generation Language : These are written in English with symbols and digits. Then are known as High level language (HLL). common high level languages are c,c++, COBOL, BASIC, FORTRAN, PASCAL, etc.

For execution the program is translation into binary form by compiler or interpreter.

$$\begin{tikzcd}
\text{Source program} \arrow[d] \\
\text{Compiler} \arrow[d] \\
\text{Object Prog.} \arrow[d] \\
\text{Linker} \arrow[d] \\
\text{Executable Program} \arrow[d] \\
\text{Run the program}
\end{tikzcd}$$

4. Fourth – Generation Language (4GL‟s) : is a programming language or programming environment designed with a specific purpose in mind, such as the development of commercial business software. In the history of computer science, the 4GL followed the 3GL in an upward trend toward higher abstraction and statement power. The 4GL was followed by efforts to define and use a 5GL.

Development of "C" (Introduction and history)

"C" is a programming language developed at AT & T Bell Laboratories of USA in 1972. It was developed Dennis Ritche in late 1970‟s. it began to replace the more familiar languages of that time like PL/1, ALGOL etc.

1. "C" became popular because of its reliability, simple and easy to use
2. It was friendly capable and reliable
3. ALGOL 60 was developed and did not become popular because it was too general and too abstract.
4. They developed "CPU" (Combined Programming Language)
5. Next as it could not come up to make ALGOL 60 better one they moved to "BCPL" (Basic Combines Programming Language. Developed by martin Richard Cambridge university)
6. At the same time a language called "B" written by ken Thompson at AT & T‟‟S. Bell laboaratories as a further simplification of BCPL.
7. "C" s compactness and coherence is mainly due to it‟s one man language. Ex- LISP, AASCA

Year	**Lang**	**Developed by**	**Remarks**
1960	ALGOL	International Committe	too general, too abstract
1963	CPL	camebridge university	Hard to Learn & implementation
1967	BCPL	Camebridge university	could deal only special problem
1970	B	AT&T	could deal only special problem
1972	C	AT & T	Lost Generality of BCPL, B restored

Programming Using C

Note : C is a middle level language because it was due to have both a relatively good programming efficiency and relativity good machine effecience.

Features of “C” Language :

1. It is robust language because of rich set of binary in – function
2. It is efficient and fast because of its variant data-types and powerful operation.
3. It is highly Portable i.e., programs written in one computer can be run on another
4. It is well suited for structure program, thus allows the user to think about the problem in the terms of functional blocks.
5. Debugging, testing and maintenance is easy
6. ability to extend itself, we can continuously add our own functions to the program.

Complier : This reads the entire source program and converts it to the object code. It provides error not of one line, but errors of the entire program. It executes as a whole and it is fast

Interpreter : It reads only one line of a source program at a time and converts it into an object code. In case of errors/same will be indicated instantly. It executes line by line and it is slow.

Linker is a function which links up the files that an present in the operating system, it also links the files for the hardware and makes the system ready for executing.

Preprocessor : This is a program, that processes the source program before it is passed on to the compiler. The program typed in the editor is the source code to the preprocessor, then it passed the source code to the compiler. It is not necessary to write program with preprocessor & activity

Preprocessor directories are always initialized at the beginning of the program. it begins with the symbol (#) hash. It place before the main() function

Eg: # include <station>

# define PI 3.14

Character Set : The characters that can be used to form words and expressions depends upon the computer to which the program is run

The Characters in C are

1. Letters A-X, a-z, both upper and lower
2. Digits 0-9
3. Special character, +,-,*,”,;,./,
4. which spaces newline, horizontal tab;,carriage return ,blank space

“C”Tokens:

Individual words and punctuation marks are characters. In a “C” program the smallest individual units are known as “C” tokens. It has 6types of token‟s

Keywords : Keywords are reserved words by compiler. Keywords are assigned with fixed meaning and they cannot be used as variable name. No header file is needed to include the keywords.

There are 32 keywords

Eg : auto, break, double, int, float

Identifiers :

These are the names of variables ,functions and arrays, these are the user defined names

Eg : # define NUM 10

# define A 20

“NUM”, “A” are user – defined id

Constants : constants in “C” are applicable to the values which not change during the execution of a program.

Integer Constants : Sequence of numberr 0-9 without decimal points, fractional part or any other symbols. It requires two or four bytes, can be +ve, -ve or Zero the number without a sign is as positive.

Eg: -10, +20, 40

Real Constants : Real constants are often known as floating constants.

Eg: 2.5, 5.521, 3.14 etc.

Character Constants

1. Single character const : A single character constants are given within a pair of single quote mark.
   Eg : „a‟, „8‟, etc.

String Constant : These are the sequence of character within double quote marks

Eg : “Straight” “India”, “4”

Variables : This is a data name used for storing a data, its value may be changed during the execution. The variables value keep‟s changing during the execution of the program

Eg : height, average, sum, etc.

Data types: C language is rich in data types

ANSI – American National Standard Institute

ANSI C Supports Three classes of data types.

1. Primary data type(fundamental)
2. Derived data types
3. User defined data types
   All “C” compiler supports 5 fundamental data types
1. Integer (int)
2. Character (char)
3. floating point (float)
4. double-precession (double)
5. void

Range of data types :

Data type	**Bytes in Ram**	**Range of data type**
char	1 bytes	-128 to 127
int	2 bytes	-32, 768 to 32,767
float	4 bytes	3.4c-38 to 3.4 c+ 38
double	8 bytes	1.7C – 308 to 1.7c +308

Integer Types : Integers are whole numbers with a range of variables supported by a particular machine.

In a signed integer uses one bit for sign and 15 bits for magnitude

C has three classes of integer storage

short int

1. int
2. long int

It has a set of qualifiers i.e.,

1. sign qualifier
2. unsigned qualifier

short int uses half the range of storage amount of data, unsigned int use all the bits for the magnitude of the number and are positive.

$$\begin{tikzcd}
\text{short int} \arrow[d] \\
\text{int} \arrow[d] \\
\text{long int}
\end{tikzcd}$$

Datatype	**Size**	**Range**
char or signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int or signed int	2 byte	-32,768 to 32, 767
unsigned int	2 bytes	0 to 65535

Floating Point Datatype: Floating Point numbers are stored with 6 digits of precision. Those are defined with keyword float. When the accuracy is not sufficient then the datatype double can be used. double gives a precesion of 14 digits these known as double precesion numbers. Still for a better process we can use long double which uses 80 bits.

$$\begin{tikzcd}\text{float} \arrow[d] \\ \text{double} \arrow[d] \\ \text{long double}\end{tikzcd}$$

<u>Character Datatype :</u> Character are usually stored in 8 bits

<u>Void datatype :</u> A void type has no value this is usually used to specify the return type of function , this function does not return any value to calling function

<u>Declaration of Variable :</u>

It tells the complier what the variable name is used, what type of date is held by the variable.

Syn: datetype v1,v2,….vn;

```
        Eg : int a, b; float
             sum; double
             ratio;
```

<u>representation of Constant</u>

```
                const int r = 10;
```

<u>Assigning values to variables</u>

```
        Eg : int x,y;
               x= 10;
               y=5;
```

<u>Type def :</u> Defined as type definition by using typedef we can create new datatype.

```
        Typedef  type  data _ame;
```

Type ---- datatype

Dataname---- Name of that type.

Programs : Program for variable declaration

```
main( )
{
        float x,p;
        x=10.1;
        p=5.2;
        printf (“x = %f”, x);
        printf (“p =  %f”, p);
}
```

O/P :

```
        x= 10.10000

        p = 5.2
```

Type def program :

```
                # include < stdio.h>
                main ( )
                {
                typedef int amt ;
                amt Rupees = 20;
                printf (“ Rupees %d“, Rupees);
                }
```

Output : Rupees 20.

Operators

Operator: An operator is a symbol that tells the Computer to perform certain mathematical or logical manipulations.

Expression: An expression is a sequence of operands and operators that reduces to single value

Eg: 10+25 is an expression whose value is 35

C operators can be classified into a no. of categories.

They include:

1. Arithmetic
2. Relational
3. Logical
4. Assignment
5. Increment and Decrement
6. Conditional
7. Bitwise
8. Special

Arithmetic Operators: C provides all the basic arithmetic operators, they are +, -, *, /, % Integer division truncates any fractional part. The modulo division produces the remainder of an integer division.

Eg:	a + b	a – b	a * b
	-a * b	a / b	a % b

Here „a" and „b" are variables and are known as operands. % cannot be used for floating point data. C does not have an operator for exponentiation.

Integer Arithmetic: When the operands in an expression are integers then the expression is an integer expression and the operation is called integer arithmetic. This always yields an integer value. For Eg. a = 14 and n = 4 then a - b = 10

a + b = 18

a * b = 56

a / b = 3

a % b = 2

14 % -3 = 2

Note : During modulo division,the sign of the result is always the sign of the first operand (the dividend)

- 14 % 3 = -2

-14 % - 3 = 2

1). Write a program to illustrate the use of all Arithmetic operator

```
main ( )
{
        int  sum, prod , sub, div, mod, a, b ;
        printf("Enter values of a, b :") ;
        scanf(" /.d  %d", & a, & b) ;
          sum = a+b ; printf("sum
         = %d", sum); sub = a-b;
        printf("sub = %d", sub);
                prod = a * b ;
        printf("prod = %d", a* b);
                div = a/b;
        printf(" Div = %d", div);
                mod = a % b ;
        printf(" mod = %d",a % b);
}
```

2). WAP to convert given no. of days into years, months days

3). WAP to use various relational operators and display their return values.

```
main ( )
{
    printf(“ in  condition  :      Return Values In”);
    printf(“ In 10! = 10         :      %5d”, 10! = 10);
    printf(“ In 10 = 10          :      %5d” , 10 == 10);
    printf(“ In 10>=10           :      %5d”, 10>=10);
    printf(“ In 10<+100          :      %5d”,  10<100);
    printf(“ In 10! = 9          :      %5d”,  10!=9);
}
```

Real Arithmetic / Floating Pont Arithmetic:

Floating Point Arithmetic involves only real operands of decimal or exponential notation. If x, y & z are floats, then

x = 6.0/7.0 = 0.857143

y = -1.0/3.0 = 0.333333

z = 3.0/2.0 = 1.500000

% cannot be used with real operands

Mixed mode Arithmetic: When one of the operands is real and the other is integer the expression is a mixed mode arithmetic expression.

Eg: 15/10.0 = 1.500000

15/10 = 1

10/15 = 0

-10.0/15 = -0.666667

Relational Operator: These are the operators used to Compare arithmetic, logical and character expressions.the value of a relational express is either one or zero .it is 1 if one is the specified relation is true and zero if it is false For eg:

10 < 20 is true   20<10 is false

The relational operators in C are

Operator	**Meaning**
<	is less than
< =	is less than or equal to
>	is greater than or equal to
> =	is greater than or equal to
= =	is equal to
! =	is not equal to

O/P

```
Condition      :     Return values

10! = 10       :     0
10 = = 10      :     1
10> = 10       :     1
10! = 9        :     1
```

4). WAP to illustrate the use of Logical Operators

```
void main ( )

{        clrscr  ( );
         printf("In     5>3 && 5<10        :      %3d",  5>3&&5<10);
         printf(" In    8<5 || 5= =5       :      % 3d", 8<5 || 5= =5);
         printf("In     !(8 = =8)          :      %3d",  !(8= =8) ;
 }
```

```
O/P   5>3    &&    5<10           :      1

      8<5    ||    5= =5          :      1

      !(8 = =8)                   :      0
```

5). WAP to show the effect of increment and decrement operators

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

```
O/P     200   210
        220   220
         12     10
```

Logical operator : Logical Operators are used when we want to test more than one condition and make decisions. here the operands can be constants, variables and expressions Logical operators are &&, ||, !

Eg: a > b && x = = 10

Logical or compound relational Expression

Truth Table && ||,   !

OP1	OP2	OPJ&OP2	OP1 ‖OP2	OP	!
1	1	1	1	0	1
1	0	0	1	1	0
0	1	0	1		
0	0	0	0		

Assignment Operator: Used to assign the result of an expression to a variable. „= „is the assignment operator. In addition C has a set of „short hand" assignment operators of the form

$$\begin{tikzcd} & \text{Var} \arrow[dl] & \text{Op} \arrow[d] & = \text{Exp :} \arrow[d] \\ \text{Variable} & & \text{operator Binary arithmetic} & \text{shorthand assignment operator} \end{tikzcd}$$

```
        var     op = exp;
        is equivalent to
        var = var op exp;
Eg:     x + = 1;        == >    x = x+1
        x+ = y+1        == >    x = x+y+1
```

6) WAP to print whether a given number is even or odd

```
main()

{       int     a, b
        printf(" Enter a number ");
        scanf(" %d", & a);
                b = a%z;
        ((b = =o)?  printf("Even"): printf("odd");
}
```

7) WAP to print logic 1 if input character is capital otherwise o

```
main ( )
{
        char x ; int y;
        printf((" \n nter a character" );
        scanf(" % C ", & x);
        y = (x>=65 && x <=90? 1:0);
        printf(" y : %d", y);
}
```

O/P

```
 1) Enter a character   :       A
2) Enter a character            :       a
                         y      :       o
```

Shorthand operator	**Assignment operator**
a + = 1	a = a+1
a - = 1	a=a-1
a * = n+1	a = a* (n + 1)
a / = n+1	a = a/(n+1)
a % = b	a = a % b

Increment and Decrement Operators:

++ and - -

The Operator + + adds 1 to the operand while -- subtracts 1, Both are unary operators

```
Eg :    ++x     or      x ++    == > x+=1       == > x=x+1
.       -- x    or      x- -    == > x-=1       == > x=x-1
```

A Profix operator first adds 1 to the operand and then the result is assigned to the variable on left. A postfix operator first assigns the value to the variable on the left and the increments the operand.

```
Eg:     1)  m = 5;                      2). m = 5
            y = ++m;                        y = m++
O/P     m =6,    y=6                      m=6, y=5
```

Conditional operator: is used to check a condition and Select a Value depending on the Value of the condition.

Variable = (condition)? Value 1 : Value 2:

If the Value of the condition is true then Value 1 is e valued assigned to the varable, otherwise Value2.

Eg: big = (a>b)? a:b;

This exxp is equal to

```
if  (a>b)
big  =  a;
else
big = b;
```

Bitwise operator : are used to perform operations at binary level i. e. bitwise. these operators are used for testing the bits, or Shifting them right or left . These operators are not applicable to float or double. Following are the Bitwise operators with their meanings.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive – OR
<<	Left Shift
>>	Right Shift
~	Complement

Eg" consider a = 13 & b = 6 as 8 bit short int (1byte)

<< Left Shift

a = 13   Binary   00001101

b = 6   00000110

Consider a << 2 which Shifts two bits to left , that is 2 zeros are inserted at the right and two bits at the left are moved out.

00001101

Moved

00110100

Finally the result is 00110100 . Deci 52 (13x4)

Note : when you shift a bit towards left its Decimal Value is multiplied by Two (2).

a =13   13= 00001101

a >>2   00000011

000000 11 Decimal 3 (13/4)

9) WAP to illustrate the use of size of operator

```
main ( )
 {  int    x = 2;
    float y = 2;
     printf (“  in  size of ( x ) is  %d bytes “,  sizeof  ( x ));
     printf (“  in   size of ( y ) is  %d bytes “,  sizeof  ( y ));
     printf (“  in   Address of  x = % u and y = % u “, & x, & y);
 }
```

```
o/p     sizeof ( x ) = 2

        sizeof ( y ) = 4

        Address of  x = 4066  and  y  =  25096
```

~ Complement

convert 0‟s & to 1‟s and 1‟s to O‟s .

& (Bitwise logical and)

op1	op2	&	|
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

```
a = 13      0000 1101
b = 6       0000 0110
a & b       0000 0100
```

| (Bitwise Logical or)

```
a = 13     0000 1101
b = 6      0000 0110
a 1b       0000 1111
```

^ (Bitwise Exclusive or)

op1	op2	^
0	0	0
0	1	0
1	0	1
1	1	0

```
a = 13      0000 1101
b = 6       0000 0110
a ^ b       0000 1011
```

special operators : these are two other operators in c.

sizeof operator : is used to find the on. of bytes occupied by a variable / data type in computer memory.

```
eg :     sizeof (float)   returns  4
         int  m, x [ 50 ]
         sizeof (m)   returns  2
         sizeof ( x )   returns  100 ( 50 x 2 )
```

i) **comma operator** : can be used to link the related expressions together. A comma- linked: list of expressions are evaluated left to right and the value of right-most exp is the value of combined expression.

Eg : value = (x = 10, y = 5, x = y)

First 10 is assigned to x

then 5 is assigned to y

finally x + y i .e. which 15 is assigned to value .

since comma has the lowest precedence of all operator, the parantheses are necessary . **Operator - precedence & Associativity**

precedence is nothing but priority that indicates which operator has to be evaluated first when there are more than one operator.

Associativity : when there are more than one operator with same precedence [priority] then we consider associativity , which indicated the order in“ which the expression has to be evaluated. It may be either from Left to Right or Right to Left.

eg : $\underbrace{5 * 4}_{1} + \underbrace{10 / 2}_{2}$

$= \underbrace{20 + 5}_{3}$

=25

Category	**operator**	**operation**	**precedence**	**Associactivity**
Highest	( )	Function call		
Precedence	[ ]	Array Subscript		
	->	C Indirect Component Selector	1	L → R
	: :	C scope access / resolution		
	.	C direct component selector		
unary	!	Logical negation (NOT)		

	~	Bitwise is complement		
	+	unary plus		
	_	unary minus		
	++	pre/pest increment	2	R→ L
	_ _	pre/pest decrement		
	&	Address		
	*	Indirection		
	Sizeof	retuns size of operand in bytes		
Member Access	.*	Dereference		
	→ *	Dereference	3	L→R
Multiplication	*	Multiply		
	/	Divide	4	L→R
	%	Remainder (Modules)		
Additive	+	Binary Plus	5	L→R
	-	Binary mains		
Shift	<<	Shift left	6	L→R
	>>	Shift Right	6	L→R
Relational	<	Less than		
	<=	Less then equal to		
	>	Greater than	7	L→R
	>=	Greater than equal to		
Equality	= =	Equal to		
	!=	Not Equal to	8	L→R

Bitwise AND	&	Bitwise AND	9	L→R
Bitwise XOR	^	Bitwise XOR	10	L→R
Bitwise OR	|	Bitwise OR	11	L→R
Logical AND	&&	Logical AND	12	L→R
Logical Or	||	Logical OR	13	L→R
Conditional	? :	(a?x:y means" if a then x else y"	14	R→L
	=	Simple Assignment		
	*=	Assign product		
	/=	Assign quotient		
	%=	Assign remainder (modulus)	15	R→L
Assignment	+=	Assign sum		
	- =	Assign Difference		
	&=	Assign Bitwise AND		
	^=	Assign Bitwise XOR		
	1=	Assign Bitwise OR		
	<<	Assign Left Shift		
	>>	Assign Right Shift		
Comma	,	Evaluate	16	L→R

Note : Unary, Conditional & Assignment operators are evaluated from Right to Left, remaining operators are from Left to Right

Type Casting: Normally before an operation takes pace both the operands must have the same type. C converts One or both the operands to the appropriate date types by "Type conversion". This can be achieved in 3 ways.

Implicit Type conversion : In this the data type /Variable of lower type (which holds lower range of values or has lower precision) is converted to a higher type (which holds higher range of values or has high precision). This type of conversion is also called "promotion".

If a „char" is converted into „int" it is called as Internal promotion

```
Eg:     int I;

        char C;

        C = „A";

        I = C;
```

Now the int Variable I holds the ASCII code of the char „A"

a) An arithmetic operation between an integer and integer yields an integer result.

b) Operation b/w a real yields a real

c) Operation b/w a real & an integer always yields a real result

```
Eg:     5/2 = 2             2/5 = 0

        5.0/2 = 2.5         2.0/50. = 0.4

        5/2.0 = 2.5         2/5.0 = 0.4

        5.0/2.0 = 2.5       2.0/5.0 = 0.4
```

<u>Assignment Type Conversion:</u> If the two Operands in an Assignment operation are of different data types the right side Operand is automatically converted to the data type of the left side.

Eg: Let „k" is an int var & „a" is a float var

int k; yes float a; yes

k= 5/2	2	k=2/5	0	a = 5/2	2.0
k=5.0/2	2	k=2.0/5	0	a = 5.0/2	2.5
k=55.0/2	2	k=2/5.0	0	a = 5/2.0	2.5
k=5.0/2.0	2	k=2.0/5.0	0	a = 2/5	0.0
				a = 2.0/5	0.4
				a = 2.0/0.5	0.4

10) WAP to read & display characters & strings

Using unformatted I/O functions

```
main ( )
{          char ch,s[20]   ;
           printf("press a special char");
           printf(" press any char");
                     ch = getch (  );
            printf("you pressed");
                      putchar ( ch );
           printf("press any number")
           ch = getch (  );
          printf(" Enter your name");
           gets (s);
          printf(" Your name is ");
           puts (s);
}
```

O/P

```
press Special c          * ↵          ch = *
Special char is     *
Press any char                        ch = z
You pressed         z
Press any number    1                 ch = 1
You pressed         1
Enter your name     C Language
```

Your name is C language

Explicit Type Conversion: When we want to convent a type forcibly in a way that is different from automatic type conversion, we need to go for explicit type conversion.

(type name) expression;

Type name is one of the standard data type. Expression may be a constant variable Or an expression this process of conversion is called as casting a value.

Eg: x = (int) 7.5

A = (int) 21.3/(int) 4.5

Y =(int) (a + b)

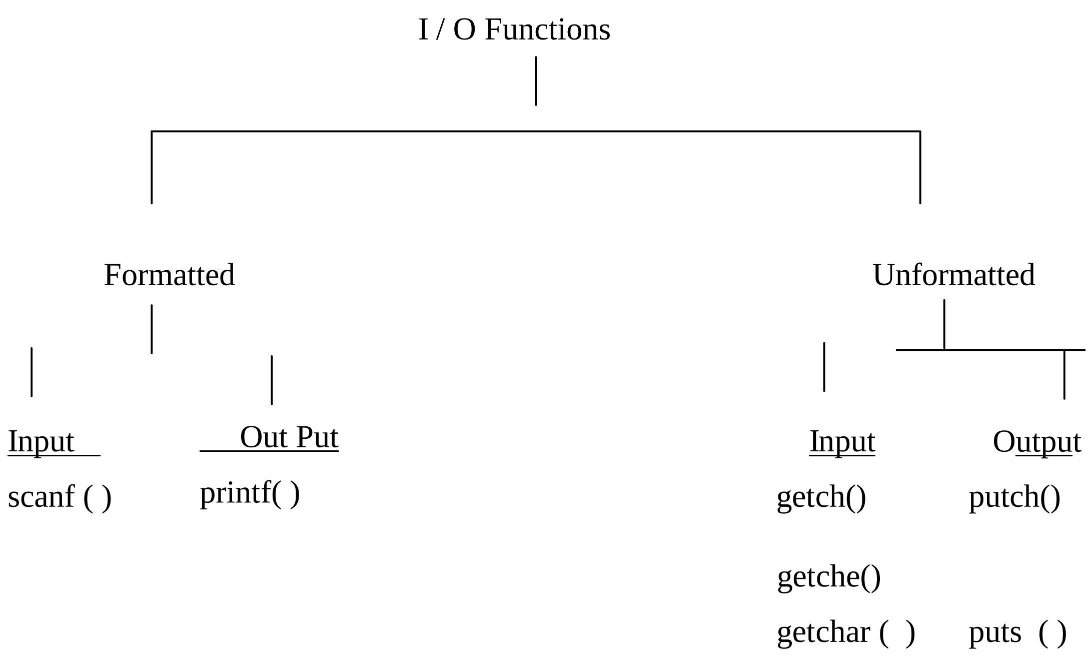
P = (double)sum/n

Basic Input output :

C has many input output functions in order to read data from input devices and display the results on the screen.

Formatted Functions: These functions read and write all types of data values. They require a conversion symbol to indents the data type using these functions the O/P can be presented in an aligned manner.

Classification:



gets ( )

Data Types with conversion symbol (format string)

	Data Type	Conversion symbol
Integer	Short integer	%d or % i
	Short unsigned	% u
	long signed	% ld
	Long unsigned	% lu
	Unsigned hexadecimal	% x
	Unsigned octal	% 0
real	float	% f or % g
	double	% lf
	signed character	%c
	unsigned char	%c
	string	%s

Escape Sequences with their ASC|| values

Escape Sequence	Use	ASC|| Value
\n	New line	10
\b	Backspace	8
\f	Form feed	12
\‟	Single Quote	39
\\	Back slash	92
\o	Null	0
\ t	Horizontal tab	9
\ r	Carriage return	13
\ a	Alert	7
|?	Question marks	63
\“	Double Quote	34

\v	Vertical tab	11

a). scanf ( ) function is used to read values using key board. It is used for runtime assignment of variables.

The general form of scanf( ) is

scanf("format String " , list_of_addresses_of_Variables);

The format string contains

- Conversion specifications that begin with % sign

Eg: Scan f(" %d %f %c", &a &b, &c)

„&" is called the "address" operator. In scanf( ) the „&" operator indicates the memory location of the variable. So that the Value read would be placed at that location.

printf( ): function is used to Print / display values of variables using monitor:

The general form of printf( ) is

printf("control String " , list_of_ Variables);

- Characters that are simply printed as they are
- Conversion specifications that begin with a % sign
- Escape sequences that begin with a „\" sign.

Eg: Program

```
main ( )
{
int avg = 346;
float per = 69.2;
printf(" Average = %d \n percentage = %f", avg, per);
}
```

O/P Average = 346

Percentage = 69.200000

printf( ) examines the format string from left to right and prints all the characters until it encounter a „%‟ or „\‟ on the screen. When it finds % (Conversion Specifier) it picks up the first value. when it finds „\‟ (escape sequence) it takes appropriate action (\n-new line). This process continues till the end of format string is reached.

Eg: Program ( )

```
main ( )

{

float per;

printf(“Enter values for avg & per”);

scanf(“ %d      %f”, & avg, & per);

printf( “ Average = %d \n Percentage = %f”, avg. per);

}
```

O/P:

```
Enter values for avg & per  346 69.2

Average = 346

Percentage = 69.200000
```

Unformatted functions character I/O functions

getchar ( ) function is used to read one character at a time from the key board

Syntax ch = getchar ( ); where ch is a char Var.

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

```
        printf("Enter a char");
        ch = getchar  ( );
        printf("ch =%c", ch);
        }
O/P     Enter a char M
        M
        ch = M
```

When this function is executed, the computer will wait for a key to be pressed and assigns the value to the variable when the "enter" key pressed.

putchar ( ): function is used to display one character at a time on the monitor.

Syntax: putchar (ch);

Ex char ch = „M‟

putchar (ch);

The Computer display the value char of variable „ch‟ i.e M on the Screen.

getch ( ): function is used to read a char from a key board and does not expect the "enter" key press.

Syntax: ch = getch ( );

When this function is executed ,computer waits for a key to be pressed from the dey board. As soon as a key is pressed, the control is transferred to the nextline of the program and the value is assigned to the char variable. It is noted that the char pressed will not be display on the screen.

getche ( ): function is used to read a char from the key board without expecting the enter key to be pressed. The char read will be displayed on the monitor.

Syntax: ch = getche ( );

Note that getche ( ) is similar to getch ( ) except that getche ( ) displays the key pressed from the dey board on the monitor. In getch ( ) „e" stands for echo.

Strng I/O functions

gets ( ) function is used to read a string of characters including white spaces. Note that wite spaces in a strng cannot be read using scanf( ) with %s format specifier.

Syntax: gets (S); where „S" is a char string variable

Ex: char S[20];

gets (S);

When this function is executed the computer waits for the string to be entered

CONTROL STRUCTURES / STATEMENTS

- A program is nothing but the execution of sequence of one or more instructions.
- Quite often, it is desirable to alter the sequence of the statements in the program depending upon certain circumstances.
  (i.e., we have a number of situations where we may have to change the order of execution of statements based on certain conditions)

(or)

- Repeat a group of statements until certain specified conditions are met.
- This involves a kind of decision making to see whether a particular condition has occurred or not and direct the computer to execute certain statements accordingly.
- Based on application, it is necessary / essential
  (i) To alter the flow of a program
  (ii) Test the logical conditions
  (iii) Control the flow of execution as per the selection these conditions can be placed in the program using decision-making statements.

„C" supports mainly three types of control statements.

I. **Decision making statements**
   1) Simple **if** Statement
   2) **if – else** Statement
   3) **Nested if-else** statement
   4) **else – if Ladder**
   5) **switch** statement

II. **Loop control statements**
   1) **for** Loop
   2) **while** Loop
   3) **do-while** Loop

III. **Unconditional control statements**
   1) **goto** Statement
   2) **break** Statement
   3) **continue** Statement

I. Decision Making Statements

(1) Simple“if”statement:

The „if‟ statement is a powerful decision making statement and is used to control the flow of execution of statements.

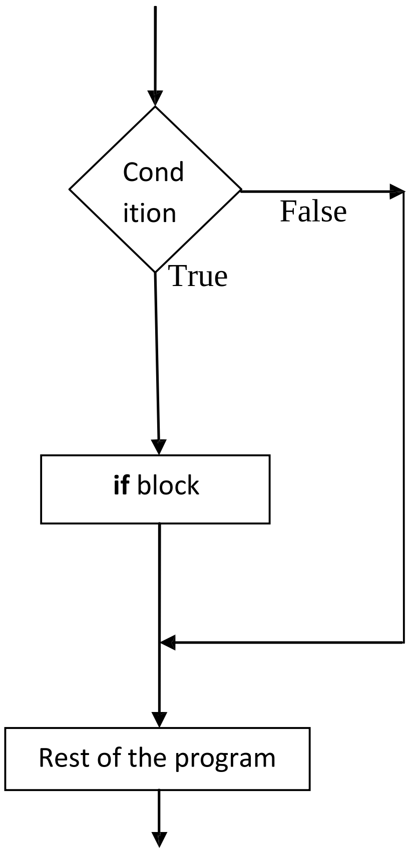
Syntax:

```
                                          if (Condition or test expression)
if (Condition or test expression)            {
        Statement;              OR               Statement;
  Rest of the program                         }
                                          Rest of the program;
```

- It is basically a “Two-way” decision statement (one for TRUE and other for FALSE)
- It has only one option.
- The statement as executed only when the condition is true.
- In case the condition is false the compiler skips the lines within the “if Block”.
- The condition is always enclosed within a pair of parenthesis ie ( ) .
- The conditional statement should not the terminated with Semi-colons (ie ;)
- The Statements following the “if”-statement are normally enclosed in Curly Braces ie { }.
- The Curly Braces indicates the scope of “if” statement.
- The default scope is one statement. But it is good practice to use curly braces even with a single statement.
- The statement block may be a single statement or a group of statements.
- If the Test Expression / Conditions is TRUE, the Statement Block will be executed and executes rest of the program.

- If the Test Expression / Condition is FALSE, the Statement Block will be skipped and rest of the program executes next..

Flow chartfor“**if** ”statement:



❖ Write a program to check equivalence of two numbers. Use “if” statement.

```
# include<stdio.h>

# include<conio.h>

void  main( )

 {

    int  m,n;

    clrscr( );

    printf(“\n Enter two numbers:”);

    scanf(“%d %d”, &m, &n);

      if((m-n)= =0)

    printf(“\n two numbers are equal”);
    getch();
  }
```

Output:

```
Enter two numbers: 5    5
Two numbers are equal.
```

- The two numbers are entered.
  - They are checked by "if"-Statement whether their difference is „0‟ or not.
  - If „true‟ the numbers are equal and the message is displayed as shown in the output.

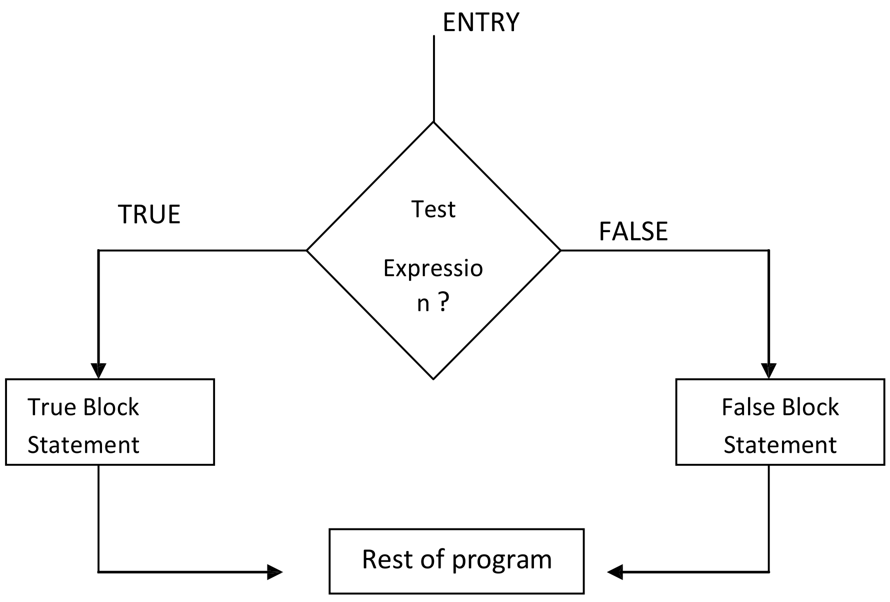
(2) "if-else"Statement:

- It is observed that the *if* statement executes only when the condition following *if* is true.
- It does nothing when the condition is false.
- In *if-else* either True-Block or False – Block will be executed and not both.
- The "else" Statement cannot be used without "if".

Syntax:

```
 if ( Test Expression or Condition )
   {
        Statements;           /*true block (or) if block */
   }
else
   {
        Statements;           /* false block (or) else block */
   }
```

Flow chart



Example 1:

Write a program to print the given number is *even* or *odd*.

```
# include<stdio.h>
# include<conio.h>
main( )
  {
        int   n;
        clrscr( );
        printf("Enter a number:");
        scanf("%d", &n);
         if( (n%2)==0 )
                printf("\n The given number is EVEN ");
         else
                printf("\n The given number is ODD ");
        getch( );
  }
```

Output:

```
Run 1:

        Enter a number: 24

        The given number is EVEN

Run 2:   /* that means one more time we run the program */

        Enter a number: 17

        The given number is ODD
```

Example 2:

Develop a program accept two numbers and find largest number and print.

```
# include<stdio.h>

# include<conio.h>

main( )

  {

        int   a,b;

        clrscr( );

        printf("Enter Two numbers:");

        scanf("%d%d", &a,&b);

         if( a>b )

                printf("\n %d is largest number",a);

         else

                printf("\n %d is largest number",b);

        getch( );

  }
```

Output:

```
Run 1:

        Enter Two numbers: 13       30

        30 is largest number

Run 2:   /* that means one more time we run the program */

        Enter Two numbers: 235      174

        235 is largest number
```

(3) <u>Nested"if–else"Statement:</u>

- Using of one *if-else* statement in another *if-else* statement is called as *nested if-else* control statement.
- When a series of decisions are involved, we may have to use more than one *if-else* statement in nested form.

Syntax:

```
if ( Test Condition1)
   {
        if ( Test Condition2)
          {
               Statement -1;
          }
        else
          {
               Statement -2;
          }
   }
else
   {
        if ( Test Condition3)
          {
               Statement -3;
          }
```

```
                else
                  {
                      Statement -4;
                   }
            } /* end of outer if-else */
```

- If Test Condition-1 is true then enter into outer if block, and it checks Test Condition-2 if it is true then Statement-1 executed if it is false then else block executed i.e Statement-2.
- If Test Condition -1 is false then it skips the outer if block and it goes to else block and Test Condition-3 checks if it is true then Statement-3 executed, else Statement-4 executed.

Example 1:

Program to select and print the largest of the three float numbers using nested "if-else" statements.

```
# include<stdio.h>
# include<conio.h>
 main( )
  {
        float  a,b,c;
        printf("Enter Three Values:");
        scanf("%f%f%f ", &a, &b, &c);
        printf("\n Largest Value is:") ;
        if(a>b)
          {
            if(a>c)
                printf(" %f ", a);
            else
                printf(" %f ", c);
          }
```

```
            else
             {
                  if (b>c)
                       printf(" %f ", b);
                  else
                       printf(" %f ", c);
              }
        getch( );
    }
```

Output:

```
        Run 1:          Enter three values: 9.12        5.34      3.87

                        Largest Value is: 9.12

        Run 2:          Enter three values: 45.781      78.34     145.86

                        Largest Value is: 145.86

        Run 3:          Enter three values: 4.0         8.0         2.0

                        Largest Value is: 8.0
```

(4) <u>The“else–if”Ladder</u>:

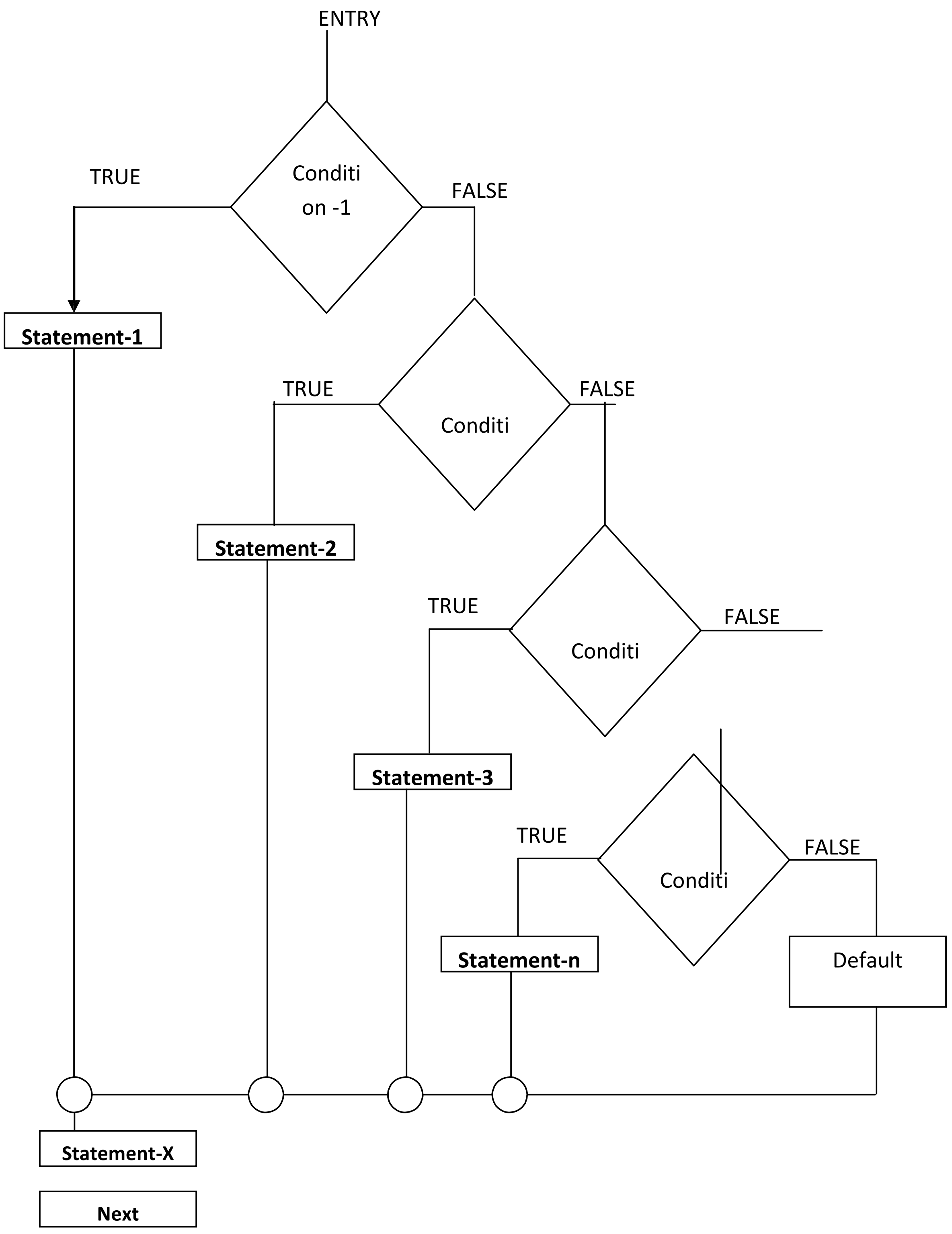
- This is another way of putting *if* „s together when multiple decisions are involved.
- A multipath decision is a chain of *if* ‟s in which the statement associated with each *else* is an *if*.
- Hence it forms a ladder called *else–if* ladder.

Syntax:

```
if (Test Condition -1)
        Statement -1;
   else  if ( Test Condition -2)
           Statement -2;
       else  if ( Test Condition -3)
               Statement -3;
                              :
                              :
                              :
                              :
              else  if ( Test Condition –n)
                      Statement –n;
                    else
                           default statement;
Rest of the Program Statements-X;
```

- The above construction is known as *else if ladders*.
- The conditions are evaluated from top to bottom.
- As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the Rest of the Program Statement–X (skipping rest of the ladder).
- When all the „n‟ conditions become false, then the final else containing the default statement will be executed.

Flow Chart:


ENTRY
TRUE
Conditi on -1
FALSE
Statement-1
TRUE
Conditi
FALSE
Statement-2
TRUE
Conditi
FALSE
Statement-3
TRUE
Conditi
FALSE
Statement-n
Default
Statement-X
Next

❖ Write a program to read three numbers and find the largest one by using “else-if” ladder.

```
# include<stdio.h>

# include<conio.h>

main( )

{

            int  a, b, c

            clrscr ( ) ;

            printf(“Enter 1st number:”);

            scanf(“%d”, &a);

            printf(“Enter 2nd number:”);

            scanf(“%d”, &b);

            printf(“Enter 3rd number:”);

            scanf(“%d”, &c);

               if ((a>b) && (a>c))

                      printf(“Highest Number is: %d”, a);

                 else if ((b>a) && (b>c))

                       printf(“Highest Number is: %d”, b);

                 else

                        printf(“Highest Numbers is: %d”, c);

            getch( );

   }
```

Output:

Run-1:

```
Enter 1st number: 52

Enter 2nd number: 74

Enter 3rd number: 90

Highest Numbers is: 90
```

Run-2:

```
Enter 1st number: 81

Enter 2nd number: 237

Enter 3rd number: 65

Highest Numbers is: 237
```

(5) <u>**The“switch-case”Statement:**</u>

- 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 which is included within the *switch* statement.
- The *switch* statement is a multi-way branch statement.
- In a program if there is a possibility to make a choice from a number of options, this structured selected is useful.
- The *switch* statement requires only one argument of *int* or *char* data type, which is checked with number of case options.
- The *switch* statement evaluates expression and then looks for its value among the *case* constants.
- If the value matches with *case* constant, then that particular *case* statement is executed.
- If no one *case* constant not matched then *default* is executed.
- Here *switch, case* and *default* are reserved words or keywords.

- Every *case* statement terminates with colon “**:**”.
- In *switch* each *case* block should end with *break* statement, i.e.

Syntax:

```
switch(variable or expression)
   {
      case Constantvalue-1: Block -1;
                              (or)
                           Statement-1;
                           break;
      case Constantvalue-2: Block -2;
                              (or)
                           Statement-2;
                           break;
       _  _    _ _   _   _ _    _
       _  _    _ _   _   _ _    _
      case Constantvalue-n: Block -n;
                              (or)
                           Statement-n;
                          break;
     default: default – block; (or)  Statement;
}
```

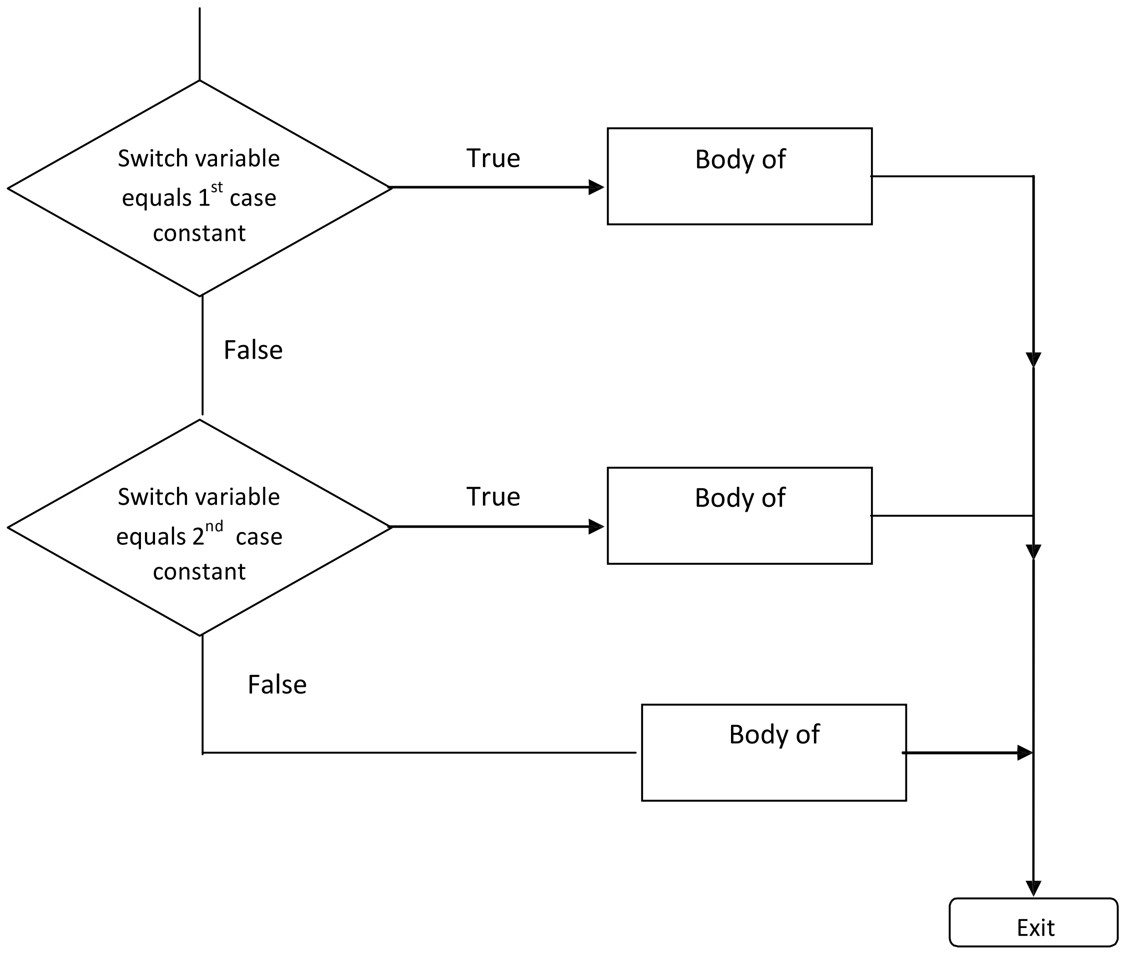
(i) The switch( ) Organization:

- The entire *case* structure following *switch( )* should be enclosed with pair of curly braces { }.
- In the block the variable or expression can be a character or an integer.
- Each *case* statement must contain different constant values.
- Any number of *case* statements can be provided.
- If the *case* structure contains multiple statements, they need not be enclosed within pair of curly braces.

(ii) The switch( ) Execution:

- When one of the cases satisfies, the statements following it are executed.
- In case there is no match, the default case is executed.

Flow Chart:


Switch variable equals 1st case constant
True
Body of
False
Switch variable equals 2nd case constant
True
Body of
False
Body of
Exit

Write a program to provide multiple functions such as 1. Addition 2. Subtraction 3. Multiplication 4. Division 5. Remainder 6. Larger out of two 7. Exit using “switch” statement.

```
# include<stdio.h>

# include<conio.h>

main( )

{

        int a, b, c, ch;

        clrscr ( ) ;

        printf(“\t = = = = = = = = = = = = = = =”);

        printf (“n\t MENU”);

        printf(“\n\t=  = = = = = = = = = = =”);

        printf(“\n \t [1] ADDITION” );

        printf(“\n \t [2] SUBTRACTION” );

        printf(“\n \t [3] MULTIPLICATION” );

        printf(“\n \t [4] DIVISION” );

        printf(“\n \t [5] REMAINDER” );

        printf(“\n \t [6] LARGER OUT OF TWO” );

        printf(“\n \t [7] EXIT” );

        printf(“\n \t = = = = = = = = = = =”);

        printf(“ \n\n\t ENTER YOUR CHOICE:”);

        scanf(“%d”, &ch);

          if(ch < = 6 && ch >=1)

            {

              printf(“ENTER TWO NUMBERS:”);

              scanf(“%d %d”, &a, &b);

            }

        switch(ch)

           {
```

```
      case  1:      c = a+b ;
                    printf(" \n Addition: %d", c);
                    break;
      case  2:      c=a-b;
                    printf("\n Subtraction: %d", c);
                    break;
      case  3:      c = a* b ;
                    printf("\n Multiplication: %d", c);
                    break;


     case  4:       c = a / b;
                    printf("\n Division: %d", c);
                    break;
     case  5:       c = a % b;
                    printf(" \n Remainder: %d", c);
                    break;
     case  6:       if (a > b)
                         printf("\n \t %d is larger than %d", a, b);
                      else if (b > a)
                           printf(" \n \t %d is larger than %d ", b, a);
                        else
                              printf("\n \t %d and %d are same", a, b);
                    break;
     case  7:       printf( " \ n Terminated by choice");
                    exit( );
                    break;
     default:       printf(" \ n invalid choice");
   }
getch ( );
```

```
}
```

Output:

```
        = = = = = = = = =
              MENU
        = = = = = = = = =
        [1] ADDITION
        [2] SUBTRACTION
        [3] MULTIPLICATION
        [4] DIVISION
        [5] REMAINDER
        [6] LARGER OUT OF TWO
        [7] EXIT
       = = = = = = = = = = = = = = =
       Enter your choice: 6
       Enter two numbers: 8  9
       9 is larger than 8
```

❖ Write a program to display the traffic control signal lights based on the following.
  - If user entered character is R or r then print RED Light Please STOP.
  - If user entered character is Y or y then print YELLOW Light Please Check and Go.
  - If user entered character is G or g then print GREEN Light Please GO.
  - If user entered some other character then print THERE IS NO SIGNAL POINT.

```
# include<stdio.h>
# include<conio.h>
main( )
{
        char L;
        clrscr( );
        printf(“ \n Enter your Choice( R,r,G,g,Y,y):”);
        scanf(“%c”, &L);
```

```
          switch(L)
          {
             case „R‟:
             case  „r‟: printf("RED Light Please STOP");
                           break;
             case „Y‟:
             case  „y‟: printf("YELLOW Light Please Check and Go");
                            break;
             case „G‟:
             case  „g‟: printf("GREEN Light Please GO");
                            break;
             default:   printf("THERE IS NO SIGNAL POINT ");
            }
          getch( );
    }
```

Output:

Run-1:

```
Enter your Choice(R,r,G,g,Y,y): g
GREEN Light Please GO
```

Run-1:

```
Enter your Choice(R,r,G,g,Y,y): G
GREEN Light Please GO
```

❖ Write a program to convert years into (1) Minutes (2) Hours (3) Days (4) Months (5) Seconds. Using switch ( ) Statements.

```
# include<stdio.h>
# include<conio.h>
main( )
{
  long int  ch, min, hrs, ds, mon, yrs, sec;
```

```
    clrscr( ) ;
    printf("\n  [1] MINUTES \n  [2] HOURS \n  [3] DAYS \n  [4] MONTHS \n
              [5] SECONDS \n  [6] EXIT \n Enter your Choice:");
    scanf ("%d", &ch);
    if(ch > 0 && ch < 6)
     {
       printf("Enter Years:");
       scanf("%d", &yrs);
     }
     mon = yrs * 12;
     ds = mon * 30;
     hrs = ds * 24;
     min = hrs * 60;
     sec = min * 60;
switch(ch)
 {
     case 1:   printf("\n Minutes : %ld", min);
                break;
     case 2:   printf(" \n Hours : % ld", hrs);
                break;
     case 3:   printf(" \n Days : %ld", ds);
                break;
     case 4:   printf( " \n Months: %ld", mon);
                break;
     case 5:   printf(" \n Seconds : %ld", Sec);
                break;
     case 6:   printf("\n Terminated by Choice");
                exit( ) ;
                break;
```

```
        default:  printf(" \n Invalid Choice");
    }
  getch ( ) :
}
```

(II) Loop Control Statements:

Loop: A loop is defined as a block of statements which are repeatedly executed for certain number of times.

1) **The"for"loop:**

* The *for* loop statement comprises of 3 actions.

- The 3 actions are **"initialize expression", "Test Condition expression" and "updation expression" "**
- The expressions are separated by Semi-Colons (**;**).
- The loop variable should be assigned with a starting and final value.
- Each time the updated value is checked by the loop itself.
- Increment / Decrement is the numerical value added or subtracted to the variable in each round of the loop.

Syntax:

```
for(initialize expression; test condition; updation )
  {
       Statement-1;
       Statement-2;
  }
```

(i) The initialization sets a loop to an initial value. This statement is executed only once.

(ii) The test condition is a relational expression that determines the number of iterations desired or it determines when to exit from the loop.

- The *for* loop continues to execute as long as conditional test is satisfied.
- When the condition becomes false the control of the program exits from the body of *for* loop and executes next statements after the body of the loop.

(iii) The updation(increment or decrement operations) decides how to make changes in the loop.

- The body of the loop may contain either a single statement or multiple statements.

*** *for* loop can be specified by different ways as shown**

Syntax	**Output**	**Remarks**
(i) for (; ;)	Infinite to loop	No arguments
(ii) for (a=0; a< =20;)	Infinite loop	„a" is neither increased nor decreased.
(iii) for (a=0; a<=10; a++) printf("%d", a)	Displays value from 1 to 10	„a" is increased from 0 to 10 curly braces are not necessary default scope of for loop is one statement after loop.
(iv) for (a=10; a>=0; a--) printf(„%d",a);	Displays value from 10 to 0	„a" is decreased from 10 to 0.

❖ Print the first five numbers starting from one together with their squares.

```
#include<stdio.h>
#include<conio.h>
main( )
  {
        int  i;
        clrscr( ) ;
        for(i = 1; i <=5; i++)
        printf("\n Number: %d its Square: %d", i, i*i);
        getch( );
   }
```

Output :

```
Number: 1 its Square: 1
Number: 2 its Square: 4
Number: 3 its Square: 9
Number: 4 its Square: 16
Number: 5 its Square: 25
```

❖ Program to display from 1 to 15 using for loop and i=i+1.

```
# include<stdio.h>

# include<conio.h>

 main( )

  {

        int  i;

        clrscr( );

        printf("\n The Numbers of 1 to 15 are:");

        for(i=1; i < =15; i=i+1)

              printf("\n%d   ", i);

        getch( );

  }
```

Output :

```
The Numbers of 1 to 15 are:

1   2   3   4   5   6   7   8   9    10    11    12    13      14       15
```

(1.1) <u>Nested"for"loop</u>:

- We can also use loop within loops.
- i.e. one *for* statement within another *for* statement is allowed in C. (or „C" allows multiple *for* loops in the nested forms).
- In nested *for* loops one or more *for* statements are included in the body of the loop.

* ANSI C allows up to 15 levels of nesting. Some compilers permit even more.

- Two loops can be nested as follows.

Syntax:

```
            for( initialize ;  test condition ;  updation)   /* outer loop */
             {
                  for(initialize ;  test condition ;  updation) /* inner loop */
                   {
                          Body of loop;
                   }
             }
```

- The outer loop controls the rows while the inner loop controls the columns.

```
for(row =1; row<=rowmax ; ++ row)
{
  for (column =1;column<=colmax; ++ column)
    {
       y = row * column;
       printf("%4d", y);
    }
  printf( "\n");
}
```

❖ Program to perform subtraction of 2 loop variables. Use nested for loops.

```
# include<stdio.h>
# include<conio.h>
void  main( )
{
  int  a, b, sub;
  clrscr( );
  for (a=3; a > =1; a - - )
  {
    for(b=1;b<=2;b++)
         {
             sub = a – b;
             printf(“a=%d  b=%d  a-b = %d \n”, a,b, sub);
         }
   }
 getch( );
}
```

Output:

```
a=3   b =1      a-b =2
a=3   b =2      a-b =1
a=2   b =1      a-b =1
a=2   b =2      a-b =0
a=1   b =1      a-b =0
a=1   b =2      a-b =-1
```

Programs on “for”LOOP:

- Print the first 5 numbers starting from one together with their squares and cubes.

```
# include<stdio.h>
# include<conio.h>
main( )
{
  int  i;
  clrscr( ) ;
  for(i=1; i<=5; i++)
  printf(“\n Number:%d its square: %d  its cube: %d”, i, i*i, i*i*i);
  getch( );
}
```

Output:

```
        Number: 1 its square: 1    its cube: 1
        Number: 2 its square: 4    its cube: 8
        Number: 3 its square: 9    its cube: 27
        Number: 4 its square: 16   its cube: 64
        Number: 5 its square: 25   its cube: 125.
```

❖ Write a program to print five entered numbers in Ascending Order.

```
# include<stdio.h>
# include<conio.h>
main ( )
{
        int  a, b, c, d, e, Sum = 0, i;
        clrscr( ) ;
        printf(“ \n Enter five Numbers:”);
        scanf( “ %d %d %d %d %d %d”, &a, &b, &c, &d, &e);
        printf( “ \ n Numbers in Ascending Order is :”)
        Sum = a + b + c + d + e;

         for(i=1; i<=Sum; i++)
          {
               if(i = = a || i = =b || i = = c || i = = d || i = = e)
                 {
                        printf(“ %3d “,i);
                  }
           }
        getch();
    }
```

Output:

```
Enter five numbers:   5   8    7    4     1

Numbers in Ascending Order is : 1    4    5    7    8
```

<u>ExamplesonNested“for”loop:</u>

(1) Write a program to display the stars as shown below

```
*

* *

* * *

* * * *

* * * * *
```

```
# include<stdio.h>

# include<conio.h>

main ( )

 {

      int  x, i, j ;

      printf(“How many lines stars (*) should be print f? :”);

      scanf(“%d”, &x);

      for(i=1; i<=x; i++)

       {

          for (j=1; j < =i;  j++)

            {

                 printf( “*”);

            }

           printf( “ \n”);

         }

       getch( );

   }
```

Output:

```
How many lines stars (*) should be print d ? : 5
```

```
*

* *

* * *

* * * *

* * * * *
```

(2) Write a program to display the series of numbers as below

```
1

1  2

1  2  3

1  2  3  4


4  3  2  1

3  2  1
```

```
# include<stdio.h>

# include<conio.h>

main(  )

{

  int  i,j,x;

  clrscr( );

  printf(“ \n Enter Value of x :”);

  scanf(“ %d”, &x);

       clrscr( ) ;       /*optional*/

  for(j=1; j < =x, j++)

   {

    for(i=1; i < = j; i++)

        printf(“%3d”, i);

    printf(“ \n”);

   }
```

```
     printf(“ \n”);
     for(j=x; j>=1; j--)
     {
       for(i=j;  i>=1;i--)
       printf(“%3d”,  i);
       printf(“ \n”);
     }
    getch( );
 }
```

Output:

```
Enter value of X : 4

                    1

                    1  2

                    1  2  3

                    1  2  3  4
```

(2) <u>The“while”loop</u>:

The simplest of all the looping structures in C is.

Syntax:

Initialization Expression;

while(Test Condition)

```
        {
              Body of the loop
              Updaion Expression
        }
```

- The *while* is an entry – controlled loop statement.
- The test condition is evaluated and if the condition is true, then the body of the loop is executed.
- The execution process is repeated until the test condition becomes false and the control is transferred out of the loop.
- On exit, the program continues with the statement immediately after the body of the loop.
- The body of the loop may have one or more statements.
- The braces are needed only if the body contains two or more statements.
- It"s a good practice to use braces even if the body has only one statement.

```
= = = = = = = = = = 
 Sum = 0; \ * Initialization * /
 n = 1;
 while(n<=10) \* Testing * /
  {
     Sum = Sum + n * n;
      n = n+1;        \* Incrementing * /
   }
 printf("sum = %d \n", Sum) ;
= = = = = = = = = = =
```

❖ Program to add 10 consecutive numbers starting from 1. Use the while loop.

```
# include<stdio.h>
# include<conio.h>
  main( )
   {
       int   a=1, Sum=0;
       clrscr( ) ;
       while(a<=10)
```

```
            {
                Sum = Sum + a;
                a++;
             }
          printf("Sum of 1 to 10 numbers is: %d", sum);
          getch( );
      }
```

Ouput:

```
        Sum of 10 numbers is: 55
```

❖ Program to calculate factorial of a given number use while loop.

```
# include<stdio.h>
# include<conio.h>
main ( )
{
         long int  n, fact =1;
        clrscr( ) ;
        printf( "\n Enter the Number:");
        scanf("%ld", &n);
        while(n>=1)
            {
                fact = fact*n;
                n - - ;
            }
        printf(" \n factorial of given number is %d", fact);
        getch( );
}
```

Output :

```
        Enter the Number :5                    /*  logic 5 * 4 * 3 * 2 * 1 = 120 */
```

Factorial of given number is 120

(3) The“do-while“loop:

- In do-while, the condition is checked at the end of the loop.
- The do-while loop will execute at least one time even if the condition is false initially.
- The do-while loop executes until the condition becomes false.

Syntax:

```
 Initialization Expression;
do
  {
        Body of the loop
        Updation Expression;
   }  while ( Test Condition);
```

```
--------------------------    ------------
  I = 1;              / * initializing * /
  Sum = 0;
  do
   {
      Sum = Sum + I;
      I = I +2;               / * increment * /
    }
    While (sum < 40 || I < 10);   \ * Testing * /
    printf(“% d  %d \n”, I, sum);
------------------------------------------------
```

❖ Program to check whether the given number is prime or not.

```
# include<stdio.h>

# include<conio.h>
```

```
main( )
{
          int  n, x=2;
          clrscr( );
          printf( “Enter the number for testing (prime or not”);
          scanf(“%d”, &n);
               do
                 {
                   if(n%x = = 0)
                    {
                        printf(“ \n the number %d is not prime”, n);
                        exit(0);
                    }
                  x++;
               } while ( x < n);
         printf(“ \n the number %d is prime”, n);
         getch( ) ;
    }
```

Output:

```
Enter the number for testing ( Prime or not) : 5
The number 5 is prime.
```

❖ Program to compute the factorial of given number using do-while loop.

```
# include<stdio.h>
# include<conio.h>
main( )
{
    int  a, fact = 1;
    clrscr( );
    printf( “ \n Enter the Number:”);
    scanf( “ %d”, &a);
   do
     {
         fact = fact * a;
         a - -;
       } while (a >= 1);
    printf( “ \n factorial of given number is %d”, fact);
    getch( );
 }
```

Output:

```
       Enter the number : 5
       Factorial of given number is 120.
```

(III) Unconditional Control Statements

(1) <u>The“break”Statement:</u>

- A break statement terminates the execution of the loop and the control is transferred to the statement immediately following the loop.
- i.e., the break statement is used to terminate loops or to exit from a switch.
- It can be used within a for, while, do-while, or switch statement.
- The break statement is written simply as **break;**

Example:

```
    switch (choice = = toupper(getchar( ))
            {
                 case  „R‟: printf(“Red”);
                            break;
                case  „W‟: printf(“White”);
                            break;
                 case  „B‟: printf(“Blue”);
```

```
                    break;
                default:   printf("Error");
            }
```

- Notice that each group of statements ends with a break statement, (in order) to transfer control out of the switch statement.
- The last group does not require a break statement; since control will automatically be transferred out of the switch statement after the last group has been executed.

(2) <u>The"continue"Statement:</u>

- The continue statement is used to bypass the remainder of the current pass through a loop.
- The loop does not terminate when a continue statement is encountered.
- Instead, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop.
- The continue statement can be included within a *while*, a *do-while*, a *for* statement.
- It is simply written as "continue".
- The continue statement tells the compiler "Skip the following Statements and continue with the next Iteration".
- In „while" and „do" loops continue causes the control to go directly to the test – condition and then to continue the iteration process.
- In the case of „for" loop, the updation section of the loop is executed before test-condition, is evaluated.

(1)
```
while (Test condition)
{
   - - - - - - - -
  if ( - - - - - - -)
    continue;
   --------------------
  --------------------
 }
```

(2) do

```
          {
            --------------------
           if ( - - - - - - )
           continue;
            -------------------
           -------------------
          } while(test – condition);
```

(3)

```
          for(initialization; test condition; increment)
           {
             - - - - - - - - - -
             if( - - - - - - -)
             continue;
            -------------------
            -------------------
          }
```

❖ Program to show the use of continue statement

```
# include<stdio.h>
void  main( )
{
   int  i=1, num, sum =0;
   for(i=0; i < 5; i ++)
   {
     printf(" Enter an integer:");
     scanf( "%d", &num);
     if(num < 0)
     {
       printf("you have entered a negative number");
       continue ;        /* skip the remaining part of loop */
```

```
      }
     sum + = num;
   }
printf("The sum of the Positive Integers Entered = % d \ n", sum);
}
```

Run 1:

```
Enter an integer: 7
Enter an integer: 3
Enter an integer: 10
Enter an integer: 15
Enter an integer: 30
The sum of the positive integers entered = 65
```

Run -2:

```
Enter an integer: 10
Enter an integer: -15
You have entered a negative number
Enter an integer: 15
Enter an integer: - 100
You have entered a negative number
Enter an integer: 30
The sum of the positive integers entered = 55.
```

<u>**Break and Continue:** (Program)</u>

* Create an infinite for loop. Check each value of the for loop. If the value is even, display it other wise continue the iterations. Print "Even" numbers from 1 to 21. Use break statement to terminate the program.

```
# include<stdio.h>
# include<conio.h>
void  main( )
{
  int  i = 1;
  clrscr( ) ;
  printf(“ \n \t \tTable of Even numbers from 1 to 20”);
  printf(“ \n \t \t = = = = = = = = = = = = = = = = = = = = \n”);
  for ( ;   ;)          /*=> Infinite loop (No Arguments).*/
  {
       if( i = = 21)
            break;
       else  if( i % 2 = = 0)
                 {
                    printf(“%d \t”, t);
                    i++ ;
                    continue;
                  }
               else
                {
                    i++ ;
                    continue;
                }
   }
getch( );
}
```

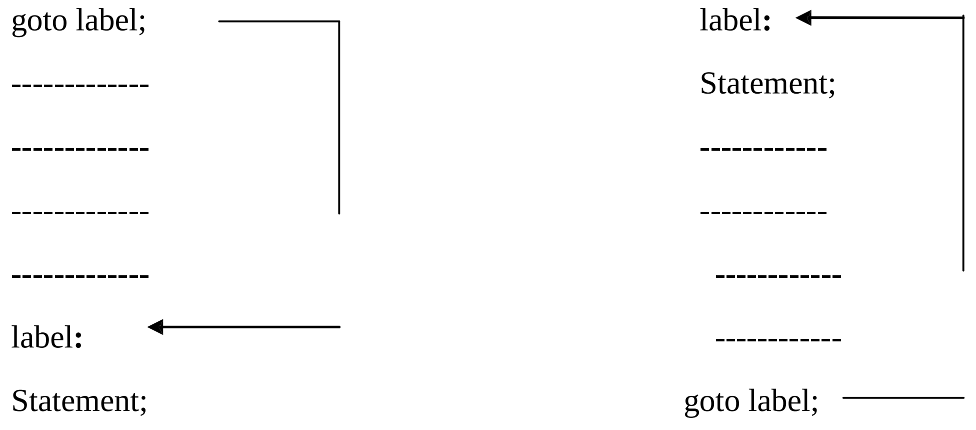
Output:

```
Table of Even numbers from 1 to 20
= = = = = = = = = = = = = = = = = = = = = = = = = ==  = = = =
```

2 4 6 8 10 12 14 16 18

(3) <u>The“goto”Statement</u>:

- C supports the “goto‟ statement to branch unconditionally from one point to another in the program.
- Although it may not be essential to use the “goto” statement in a highly structured language like „C‟, there may be occasions when the use of goto is necessary.
- The goto requires a label in order to identify the place where the branch is to be made.
- A label is any valid variable name and must be followed by a colon(**:**).
- The label is placed immediately before the statement where the control is to be transferred.
- The label can be any where in the program either before or after the goto label statement.



Forward Jump **Backward Jump**

- During running of a program, when a statement like “goto begin;” is met, the flow of control will jump to the statement immediately following the label “begin:” this happens unconditionally.

- „goto‟ breaks the normal sequential execution of the program.
- If the “label:” is before the statement “goto label;” a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a „backward jump‟.
- If the “label:” is placed after the “goto label;” some statements will be skipped and the jump is known as a “forward jump”.

* Write a program to detect the entered number as to whether it is even or odd. Use goto statement.

```
# include<stdio.h>

# include<conio.h>

# include<stdlib.h>

void  main( )

{
    int   x;
    clrscr( );
    printf("Enter a Number:");
    scanf("%d", &x);
    if(x % 2 = = 0)
        goto  even;
    else
        goto  odd;
    even:
      printf("\n %d is Even Number");
      return;
    odd:
      printf(" \n %d is Odd Number");
}
```

Output:

```
Enter a Number : 5
5 is Odd Number.
```

ALGORITHM:

- The approach or method that is used to solve a specific problem is known as an Algorithm.
- A set of pseudo code or broken English steps written sequentially to solve a problem is termed as on algorithm.
- Any documentation that clearly holds the procedure of problem solving, including pictorial representations are termed as algorithms.

Example:

```
Step 1:        start
```

Step 2: - - -

Step 3: - - -

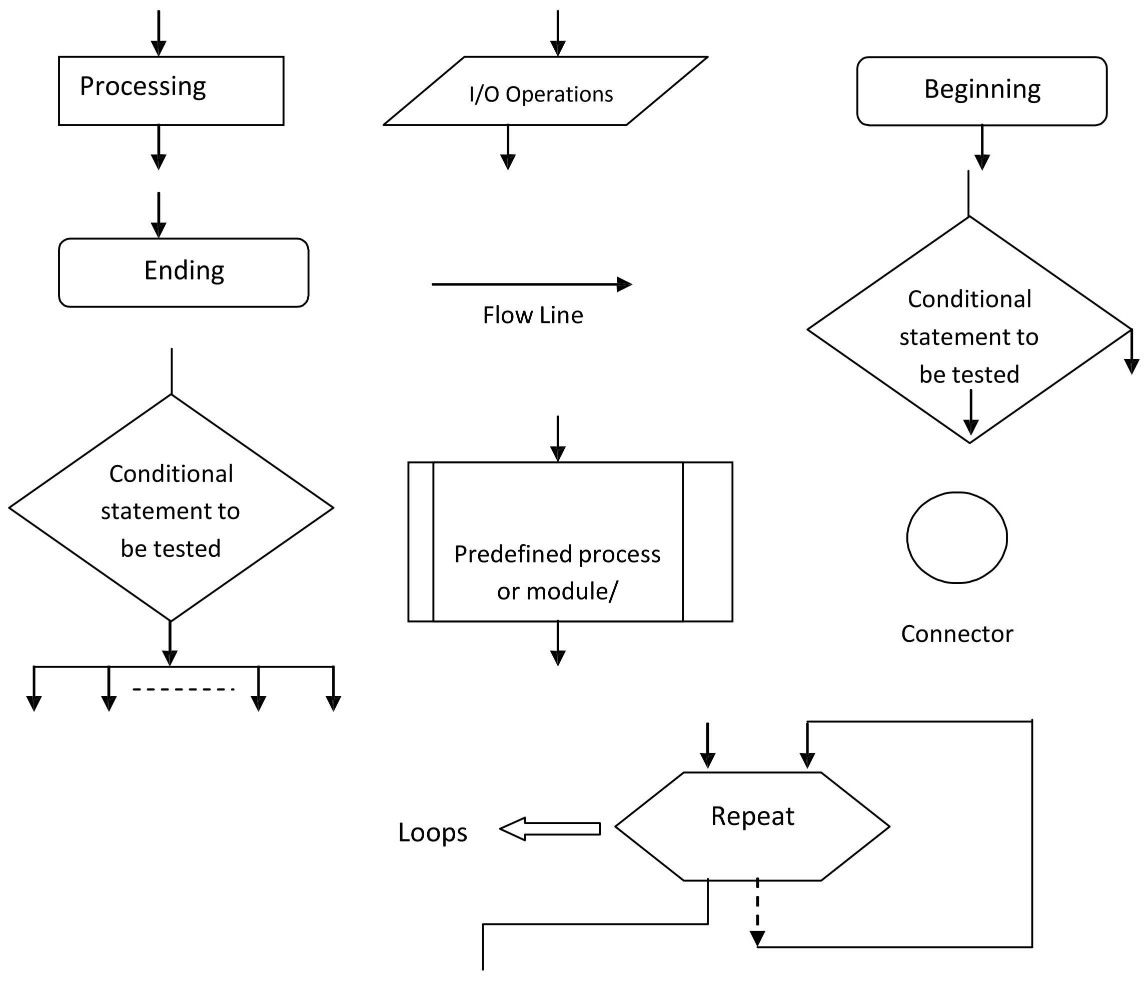
Step 4: stop

- Algorithm is a step-by step procedure.

FLOW CHART:

- The most common method of describing an algorithm is through the use of flowcharts.
- The flow chart presents the algorithm pictorially.
- All the operations to the performed and all the paths of processing to be followed while solving problem are indicated in a flow chart.
- Flow charts are represented using different geometric shapes.

Flow Chart Symbols:

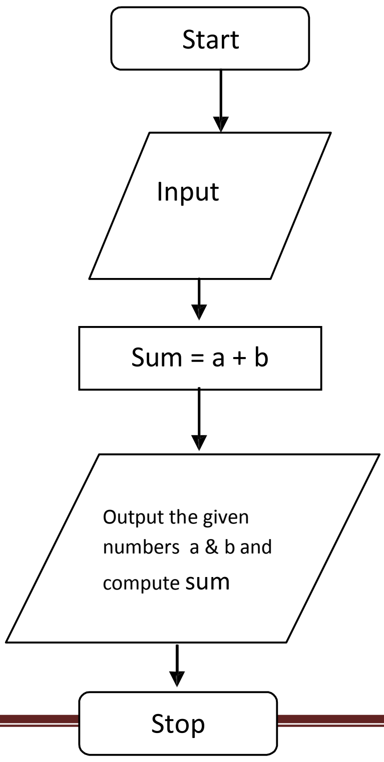

Processing
I/O Operations
Beginning
Ending
Flow Line
Conditional statement to be tested
Conditional statement to be tested
Predefined process or module/
Connector
Loops
Repeat

Example (1):

Program to find the sum of two given numbers.

```
# include<conio.h>              /* including header file * /
# include<stdio.h>
main( )
 {
     int a,b, sum;                      /* Declaration of variables * /
     clrscr( );
     printf( “ \n Enter two numbers ”);
 /* A message is displayed on the monitor indicating the action to be taken by the user */
    scanf(“ %d %d”, &a, &b);                  /* The input statement follows */
    printf( “ \n a = %d \t b = %d ”, a,b);
    sum = a + b;                                          /* computing the sum * /
    printf( “ sum = %d \ n”, sum);              /* printing the sum * /
    getch( );
}
```

Flow Chart:

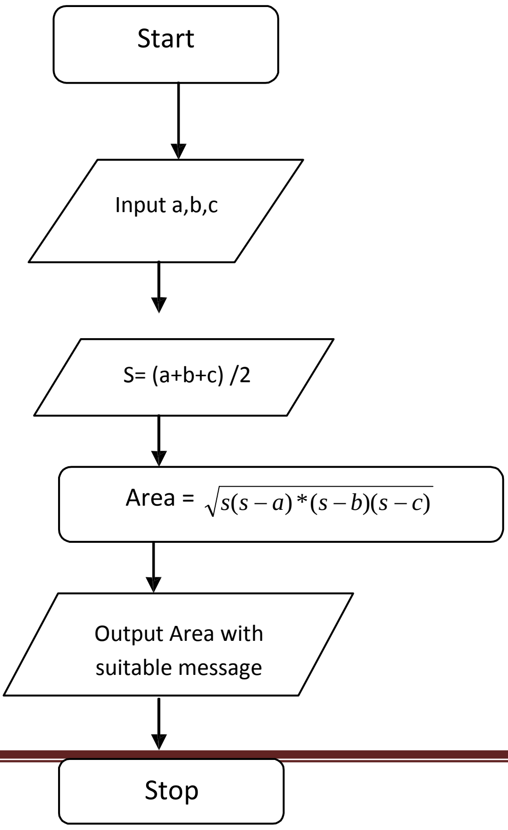


Example (2):

Program to compute the area of a triangle.

```
# include<conio.h>                /* including Header files */
# include<stdio.h>
# include<math.h>
main( )
{
   float  a,b,c, s, area;              /* declaration of variables */
   clrscr( ) ;
   printf( “ \n Enter the 3 sides of the triangle ”);
   printf( “such that sum of any 2 sides is greater than 3rd side \n”);
   scanf(“ %f  %f  %f ”, &a, &b, &c);
   printf(“ \n 3 sides of the triangle are: a = %f  \t  b = %f  \t  c=%f ”, a , b , c);
     s = (a+b+c)/2;        /* computing value of  s */
  area = sqrt ( s* (s-a) * (s-b) * (s-c));           /* computing area */
  printf( “ \n Area of the Triangle is: %f ”, area);
  getch( );
 }
```

Flow Chart:

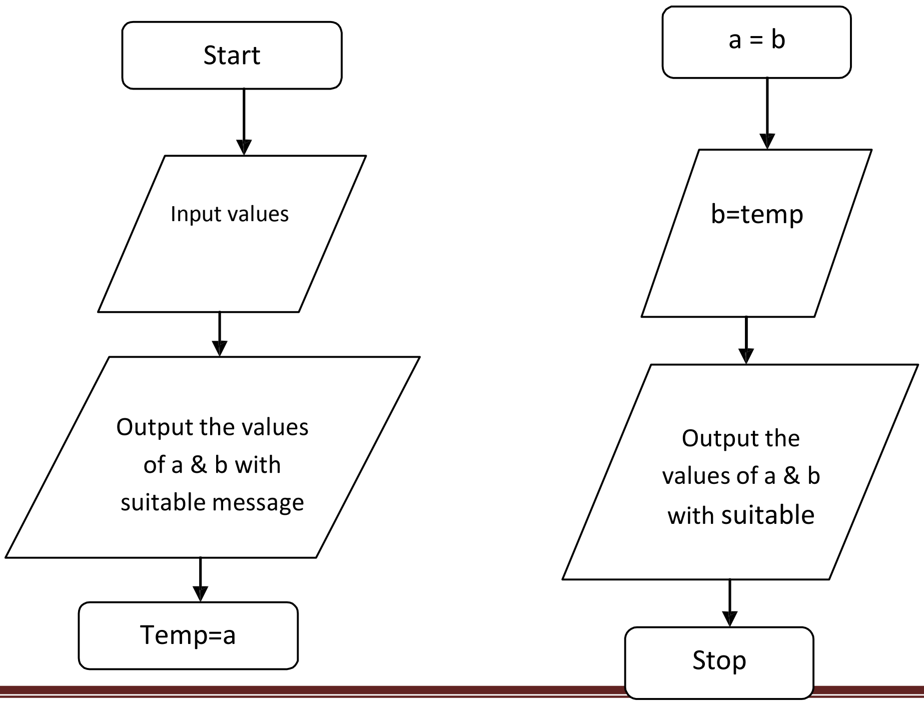

Start
Input a,b,c
S= (a+b+c) /2
Area = $\sqrt{s(s-a)*(s-b)(s-c)}$
Output Area with suitable message
Stop

Example (3):

Program to swap the contents of two variables

```
# include<stdio.h>
# include<conio.h>
main( )
{
    int  a, b, temp;                   /* Declaration of variables */
    clrscr( ) ;
    printf(“ \n Enter two numbers ”);
    scanf(“ %d %d ”, &a, &b);                        /* input statement */
    printf(“ \n a = %d  \t b= %d ”, a,b);
      temp = a;
      a = b ;
      b = temp;              /* swapping contents of a & b */
    printf(“After swapping \n a = %d \t b=%d ”, a, b);
    getch( );
}
```

Flow Chart:


Start
Input values
Output the values of a & b with suitable message
Temp=a
a = b
b=temp
Output the values of a & b with suitable
Stop

UNIT –III : FUNCTIONS

Introduction :

Functions are subprograms which are used to compute a value or perform a task. They cannot be run independently and are always called by the main ( ) function or by some other function.

There are two kinds of functions

1. Library or built–in functions
2. User–designed functions

1. Library or built-in functions are used to perform standard operations eg: squareroot of a number sqrt(x), absolute value fabs(x), scanf( ), printf( ), and so on. These functions are available along with the compiler and are used along with the required header files such as math.h, stdio. h, string.h and so on at the beginning of the program.
2. User defined functions are self–contained blocks of statements which are written by the user to compute a value or to perform a task. They can be called by the main() function repeatedly as per the requirement.

USES OF FUNCTIONS :

1. Functions are very much useful when a block of statements has to be written/executed again and again.
2. Functions are useful when the program size is too large or complex.Functions are called to perform each task sequentially from the main program. It is like a top-down modular programming technique to solve a problem
3. Functions are also used to reduce the difficulties during debugging a program

USER DEFINED FUNCTIONS :

In C language, functions are declared to compute and return the value of specific data type to the calling program. Functions can also written to perform a task. It may return many values indirectly to the calling program and these are referred to as void functions.

FUNCTION DECLARATION :

The general form of a function declaration is

```
type name (type arg1, type  arg2 …….. type  argn)

{

<local declaration >

--------------------

< statement block>

--------------------

return (variable or expression)

}
```

Where type is the data type of the value return by the function and arguments expected.

arg1, arg2…. argn are the arguments which are variables which will receive values form the calling program, name is the name of function by which the function is called by the calling program.

There is a local declaration of variables. These variables are referred as local variables, are used only inside the function. The statement block consists of a set of statements and built-in functions which are executed when the function is called. The result is returned to the calling program through a return statement that normally appears at the end of a function block. This function block starts and ends with braces { }.

Function main() :

1. main() is the starting function for any C program. Execution commences from the first statement in the main () function
2. It returns int value to the environment that called the program. Usually zero is returned for normal termination of the main(). Non zero is returned to convey abnormal termination
3. It uses no parameter. But it may use two specific parameters
4. Recursive call is allowed for main () function also
5. Only the function body varies from programmer to programmer main (). Function heard follows the common syntax by either having no parameter or only two standard parameters.
6. The program execution ends when the closing brace of the in main is reached.

FUNCTION PROTOTYPE :

When a C program is compiled, the compiler does not check for data type mismatch of actual arguments in the function call and the formal arguments in the function declaration. To enable the compiler to check the same, a function prototype declaration is used in the main program.

Function prototype is always declared at the beginning of the main() program.

ACTUAL AND FORMAL ARGUMENTS

Passing of values between the main program and the function takes place through arguments.

The arguments listed in the function calling statements are referred to as actual arguments. These actual values are passed to a function to compute a value or to perform a task.

The arguments used tin the function declaration are referred as formal arguments. They are simply formal variables that accept or receive the values supplied by the calling function.

Note: The number of actual and formal arguments and their data types should match.

The function call sends two integer values 10 and 5 to the function

int mul(int x, int y) which are assigned to x and y respectively.

The function computers the product x and y assigns the result to the local variable p, and then returns the value 25 to the main() where it is assigned to y again..

Rules to call a function :

The following rules are used to call a function is a program:

1. A function has a statement block which is called by the main( ) or any other function.
2. When the data type in a function declaration is omitted the function will return a value of the type integer.

3. The data type of the formal argument may be declared in the next line which follows the function declaration statement.

Formal Parameter List :

The parameter list declares the variables that will receive the data sent by the calling program.

They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as formal parameters.

FUNCTION CALLS:

A function can be called by simply using the function name followed by a list of actual parameters (or arguments)

```
main()
{       int y ;
        y = mul (10,5);             / * function call * /
        printf (“‘%d \n”,y) ;
}

int mul(int x,int y)
{
int p ;                 / * local variables x=10, y=5 * /
p= x * y ;
return(p);
}
```

When the compiler encounters a function call the control is transferred to the function mul(), this function is then executed line by line as described and the value of p is returned when the return statement is encountered. This value is assigned to y.

Parameters can also be used to send values to the calling programs.

Parameter list contains declaration of variables separated by commas and surrounded by parenthesis.

Examples:

```
float quadratic ( int a , (n+b, )n+c) {………}
double power 9double x, int n) {……….}
float mul (float x, float y) {………..}
int sum (int a, int b) {………}
```

There is no semicolon after closing parenthesis.

The declaration of parameter variables cannot be combined int sum (int a,b) is invalid.

A function need not always receive values form the calling program.

In such cases, functions have no formal parameters,

To indicate that the parameter list is empty we use the keyword void between parenthesis as

```
void  printline(void)
{
}
```

This function neither receives any input values nor returns back any value.

Many compilers accept an empty set of parenthesis without specifying anything as void printline ( )

But it is good programming style to use void to indicate null.

Note:

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.

3. The types must match the types of parameters in the function definition in number and order.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters, the list is written as void.
6. The return type is optional, when the function returns int type data.
7. The return type must be void of no value is returned.
8. When the declared type do not match with the types on the function definition compiler will produce an error.

FUNCTIONS ACCEPTING MORE THAN ONE PARAMETER:

A function can accept more than one parameter. The parameters are separated by commas. For example, consider the following program having a function maxfunc() that accepts three parameters and computes their maximum.

```
#include <stdio.h >

int maxfunc(int i, int j, int k)

{

        int max ;

        if (i> = j && i> = k)

         max = i ; else

        if (j> = k) max

              = j;

        return max;

}

void main ( )

{

        int m, a,b,c;

        printf (" input 3 numbers:") ;

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

        m= maxjunc (a,b,c);

        printf(" The maximum is%d \n",m);

}
```

Run:

```
        Input 3 numbers:  4
```

```
                6

                5

The maximum is  6
```

The main function calls the function maxfunc(). Three integer variables passed to it are separated by commas. The return value is assigned to m and is displayed. Since maxfunc() comes before the function main separate function and function definition are unnecessary.

The variables m,a,b and c are declared in main. They can be used only in the function main. An attempt to use them in maxfunc() causes a compile time error. Similarly, the variables max,i,j and k (i,j,k being the parameters which the function accepts) belong to the maxfunc(). These variables cannot be accessed in main. The region of the program where an identifier can be accessed is called the "Scope of the identifier". Thus the scope of „a‟ is the function main, while the scope of i is the function maxfunc().

The variables with the same name can exist in both main and maxfunc().

USER DEFINED AND LIBRARY FUNCTIONS :

We used the functions such as printf and scanf which are already written, compiled and placed in the C-library and are called library functions.

Functions which can be defined by the user are called user defined functions.

CONCEPTS ASSOCIATED WITH FUNCTIONS:

1. Function declaration or function prototype
2. Function definition (function declaration and function body)
3. Combination declaration and function definition
4. Passing arguments
5. Return statement
6. Function call

Parts of function :

A function declaration can appear outside all the other functions. In this case all functions know about the other function declarations and can call this function. Functions can also be declared within other functions. In this case only the function within which the declaration is present will know about it.

Ex:

```
void main()
 {
   int i;
   double cube (double);
 }
```

<u>Parts of a function</u>

```
void main( )
{
   void func1();   →      function declaration
…………….
…………….
func1();           →       function call
…………….
…………….
}
void func1()
{
……………
                  Function body              function definition
……………
}
```

Function declaration:

A function declaration provides the following information to the compiler

- The name of the function
- The Type of the value returned (optioned, default is integer)
- The number and the type of arguments that must be supplied in a call to the function.

When a function call is encountered, the compiler checks the function call with its declaration. So that correct argument types are used. A function declaration has the following syntax:

return type function name (type, type ….. type);

return type specifies the data type of the value in the return statement. A function can return any data type , if there is no return value, the keyword void is placed before the function name. The function declaration terminates with a semicolon.

Ex: double cube(double);

The above declaration informs the complier that the function cube has argument of type double. The function cube returns double value. The complier knows how many bytes to retrieve and how to interpret the value returned by the in function declarations are also called prototypes, since they provide a model or blue print of the function.

Function definition:

The function definition is similar to the function declaration but does not have the semicolon. The first line of the function definition is called a function declarator. This is followed by the function body. It is composed of the statements that make up the function, delimited by braces. The declarator and declaration must use the same function name, number of arguments, arguments types, and the return type. No function definition is allowed with in a function definition.

Ex:

```
void main(void)
```

```
{
int i,j;            /* variable declaration */
long fact (unsigned int num); /* function declaration or in prototype */
}
```

The definition of the function cube is given below :

```
double cube (double dnum)
{
return dnum * d num * dnum;
}
```

A function can contain any number of statements.The statements are enclosed with in curly braces { and }. The function definition may include declarations of variables and declaration of other functions. Note that the return statement need not enclose the return value with in parenthesis.

Elimination of function declaration:

The programmer can place function declarations anywhere in the program. If the functions are defined before they are called, then the declarations are unnecessary.

Ex:

```
#include <stdio.h>
int max (inta, int b)
{
return a>b ? a: b;
}
void main ()
{
int i,j, imax;
printf( “ enter two numbers:”);
```

```
        scanf("%d%d; & ;, &;);

        i max =max (i,j);

        printf(" the maximum of %d and %d is %d", i,j, i max);

        }
```

Function return type:

Functions in C may or may not return values. If a function does not return a value the return type in the function definition and declaration is specified as void. Otherwise, the return type is specified as a valid data type.

```
main()

{

unsigned sq = squareint (32); /*function  call *)

printf(" The square of 32 is ./.n\n", sq), /*  control returns here */

}
```

The function squareint is called and the return value is assigned to the variable sq. the control is returned to the printf statement after the statements with in the function definition of squareint are executed.

FUNCTION PARAMETERS:

Function parameters are the means of communication between the calling and the the called function. They can be classified into formal parameters and actual parameters. The formal parameters are the parameters given in the function declaration and function definition. The actual parameters, often known as arguments, are specified in the function call.

Ex:

```
        int sum(int a, int b)                /* This and the following body (in curly braces c
        {                              constitutes the function definition */

                return  a+b;

        }

void main(void)
```

```
{
int x,y,z;
z = sum (x,y);
}
```

Definition of function that does not return anything is as follows:

```
void functionname(parameter list)
{
Statement/Statements ;
return;        /* optional since it is at the end of the in anyway and the in has no ref value */
}
```

The definition of a function that returns a value of type . Type name has the following syntax:

```
Typename Functionname (parameterlist)
{
Statement/Statements ;
return value;        /* return keyword must be used. And it must be followed by a value
                        that matches the return type specified by typename */

}
```

Even in case of functions having return values multiple return statements can exist. Parameter list is the of arguments separated by commas.

Function Call :

A function call is specified by the function name followed by the values of the parameters enclosed with in parenthesis, terminated by a semi colon (;).

Ex: unsigned squareint(unsigned x)

```
    {
        return  x * x;
     }
```

There are two ways in which we can pass arguments to the function:

Call by value

Call by reference.

Call by value :

In this type value of actual arguments are passed to the formal arguments and the operation is done on the formal arguments. Any changes made in the formal arguments does not effect the actual arguments because formal arguments are photocopy of actual arguments. Hence when the function is called by the call by value method, it does not effect the actual contents of the actual arguments. Changes made in the formal arguments are local to the block of called function. Once control returns back to the calling function the changes made vanish.

Ex: program to send values by call by value

```
main( )
{
int x,y, change (int, int);
clrscr();
printf(" \n enter values of x & y : ");
scanf("%d %d ", & x, & y);
change(x,y) ;
printf("\n In main ( ) x=% d y = % d", x,y);
return 0;
}
change(int a, int b)
{
k = a;                                    o/p: enter values of x & y : 5 4
a=b;                                    In change (1) x=4 y=5

b=k;

}
```

In the above program, the variables a and b defined in function definition are known as formal parameters or dummy parameters or place holders. The variables x and y are actual parameters, they specify the values that are passed to the function change x and y are arguments in the function change x and y arguments in the function call.

The number of arguments in the function call and the function declarator must be the same.

The date type of each of the arguments in the function call should be the same as the corresponding parameter in the function declaration statement.

The names of the arguments in the function call and the names of parameters in the function definition can be same or different.

C Provides two mechanisms to pass arguments to a in

- pass arguments by value, and

- pass arguments by address or by pointers

An argument may take the form of a constant, variable, expression, pointer, structure etc.

<u>Category of functions :</u> A function, depending on whether arguments are present or not and whether a value is returned or not may belong to one of the following categories.

<u>1.Function with no arguments and no return values:</u>

-when a function has no arguments, it does not receive any data from the calling function

-when a function does not return a value, the calling function does not receive any data from the called function.

-There is no data transfer between the calling function and the called function.

<u>2.Function with arguments but no return values:</u>

- The main function has no control over the way the functions receive input data.

- we shall modify the definitions of both the called functions to include arguments as follow.

```
void printline(char ch)

void value (float p, float r, int n)
```

- The arguments ch, p, r and n are called formal arguments
- The calling function can now send values to there arguments using function calls containing appropriate arguments
- For example the function call value (500, 0.12, 5) would send the values 500,0.12 & 5 to the function void value (float p float r, int n) and assigns 500 to p, 0.12 to r and 5 to n values 500, 0.12 and 5 are the actual arguments, which become the values of the formal arguments inside the called function. The actual and formal arguments should match in number, type and order.

The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument.

3. Function with arguments with Return values

- The function value in previous program receives data from the calling function through arguments. But does not send back any value.
- Rather, it displays the results of calculations at the terminal.
- We may not always wish to have the result of a function displayed.
- We may use it in the calling function for further processing
- Sometimes, different programs may require different output formats for displays of results.
- There short comings can be overcome by handling over the result of a function to its calling function where the returned value can be used as required by the program
- A self contained and independent function should behave like a „black box‟ that receives a predefined form of input and output a desired value
- Such functions will have two way data communications between them

```
#include < stdio.h >

#include < conio.h>

main()

{       int fact(int k)         /* function prototype declaration * /

        int n,r,ncr ;

        clrscr();

        printf( “/ n Enter values of n and r;”);

        scanf (“%d %d “. & n, &r:”);
        ncr = fact (n) / (fact (r) ;”);

        printf (“/n value of ncr - %d”, ncr);

        getch( );

}

int fact(int k) /* recursive function to find factorial */

{

        if(k = = 1)

        return(1);

        else

        return (k*fact (k+));

}
```

When the program is executed, the user has to enter value of n and r. The function is called to find factorials of n, r, and (n-r). The value for ncr is obtained and printed as shown below

```
o/p. Enter values to n and r : 5 : 3

     Value of ncr = 10
```

<u>Recursion :</u>

A function calling itself again and again to compute a value is known as recursive function or recursion function or recursion. Normally a function is called by the main program or by some other function but in recursion the same function is called by itself repeatedly.

<u>Use of recursion function :</u>

1. Recursion functions are written less number of statements.
2. Recursion is effective where terms are generated successively to compute value.
3. Recursion is useful for branching process. Recursion helps to create short code that would otherwise be impossible .

<u>Program</u>: Write a recursive fuction to find the factorized of a given integer. Use it to find ncr = n1 r1 (n-r);

EXTRA LAB PROGRAMS :

1.Write a function to add two matrices of order m x n. also write main program which can road the values of the matrices and print the resultant matrix.

```
# include ,stdio‟h>

# include <con:o.n>

# include <stdl b.h>


int a(10) (10), b(10)(10), c(10)(10), m,n,I,j;, /* global declarationjs */


main ( )

{

void matadd( );

clrscr();

printf(“ in how many rows and (columns?”)‟

scanf(“ %d % d; & m, & n);

printf(” /n Enter A matrix values in”)

for (i=0; i<n ; i**)

for (j=0; j<n; j**)

    scanf (“% d”, & a(i) (j));


matadd();     /*  call function *

printf(“in resultant matrix is /n “);

for (i=d; i<m; i+ +)

{

for ( j =0; j<n : j++)

printf(“:%d”. c(i)(j));

printf (;,n”);
}
getch ( );
}
void matadd ( ) // * function def */
{
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
                c[i][j] = a[i][j] + b[i][j];
}
```

When this program is executed, the user has to enter the order of matrix m and n and the values of the matrices. A sample input and out put of the program is

```
How many rows and columns ? 2 2

Enter A matrix values
    2       -2
    0        4

Enter B matrix values
    6       2
    3       -5

Resultant matrix is
    7       0
    4       -1
```

2.Write a function in C to find the GCD of two integers. 13 Using the function, write a program to find the GCD of 3 integers

```
# include <stdio.h>
# include <coni0.h>
```

```
main()
{
        int a,b,c,d1, d2,d3;
        int gcd (int, int);
        clrscr();
        printf(“ in Enter three integers : “);
        scanf (“%d %d ; & a, & b, & c);
        d1 = gcd(a,b);
        d2 = gcd (a,c)
        d3 = gcd(b,c);
        if(d1= = d3)
                printf (“ in greatest common divisior is %d”, d1);
        else
                printf (“in greatest common divisior is % d”, gcd (d1,d3));
        else
                printf („in greatest common divisior is %d”, gcd (d1,d3));
        printf (“ in in press any key to continue… “);
}

int gcd(int x, int y)
{
     int nr, dr, r;
      if(x>=y)
      {
                nr= x;
                dr = y;
      }
      r = nr.% dr;
     return (dr);
}
```

When the program is executed, the user has to enter the three integers. The function is called to find the common divisor in (a,c), (A,C) and (b,c). if the GCD obtained in these three calls are equal then the GCD will be displayed. Otherwise the function is called to find the common divisor in (d1,d3) or (d1,d2) to print the greatest common divisor

o.p:

```
        Enter three integers : 45 27 81

        Greatest common divisor is 9

        Press any key to continue.
```

3.Write a C program which calls a function reverse ( ) which excepts a string and displays its reverse.

The string is read in the main program and the function called with this string as an argument to print the reverse of the string

```
# include < stdio‟h>

# include < conio.h>

# include < string.h>


main ()

{

        char reverse (char s( j);

        char s((20);

        clrscr();

        printf (" in Enter a string:")‟

        gets(st);       /* call the function to print the reverse string */

        printf (" in %s is reversed as %s", st, reverse (st));

        getch ( );
}
char  reverse (char str ( ))
{
        int  i,j;
        char  rst (30);
        j=0;
        i=strlen (st)-1;
        while(i > = 0)
        {
          rst[j] = st[i]
          i---;
          j++;
       }

rst[j] = „\0‟;
return (rst);
}
```

When this program is executed, the user has to enter a string and the reverse of string will be printed .

o/p:

```
        enter a string : WELCOME
        WELCOME is reversed as EMOCLEW
```

<u>Objective Questions</u>

1. Function may be placed in the main function [c]

(a) above main ( ) (b) below main ( )

(c) above or below main( ) (d) none

2. Calling a function again and again is called [b]

(a) iteration (b) recursion

(c) both (a) & (b) (d) none

3. Declaration of a function name along with the type and optional dummy arguments at the beginning of calling program is [c]

(a) called function (b) calling function

(c) prototype (d) all the above

4. Function prototype is always declared in the main program [b]

(a) ending of i9n (b) beginning of the main ( )

(c) after calling function (d) after called function

5. Variables declares inside the body of a function or main program are reserved as

(a) formal variables (b) actual variables

(c) global variables (d) local variables

6. Global variables are placed in the main function [a]

(a) outside main for (b) inside main in

(c) in function definition (d) because calling function

7. A void function can also use a return statement without a return value [b]

(a) return x (b) return 0

(c) return (d0 none

8. By default the function returns. [a]

(a) integer value (b) float value

(c) char value (d) none of the above

9. The meaning of keyword void before the function name means [a]

(a)function should not return any value (b) float value

(c) char value (d) none of the above

10. The main ( ) is a [b]

(a) Library function (b) User defined function

(c) keyword (d) None of the above

Call by reference :

In this type instead of passing values, addresses are passed. Function operates on addresses rather than values. Here the formal arguments are pointers to the actual arguments. In this type , formal arguments point to the actual argument. Hence changes made in the arguments are permanent.

Ex: A program to send a value by reference to the user defined function.

```
main ( )
{
int  x,y ,change (int *, int *);
clrscr();
printf (“/n Enter values of x& y” );
scanf (“%d %d “, & x, & y);
change (&x, &y);

printf ( "/n in main ( ) x =%d % d y = ./. d ", x,y);

return 0;

}

change (int * a, int * b)

{

int * k;

*k = *a;

*a = * b;

*b = * K;

printf (" /n in chage ( ) x =% d y=%d", *a, *b);

}

Enter values of x & y 5 4

In change ( ) x = 4 y = 5
```

ARRAYS

- The fundamental data types, namely char, int, float, double are used to store only one value at any given time.
- Hence these fundamental data types can handle limited amounts of data.
- In some cases we need to handle large volume of data in terms of reading, processing and printing.
- To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items.
- „C" supports a derived data type known as **array** that can be used for such applications.

Arrays:

- An array is a fixed size sequenced collection of elements of the same data type.
- It is simply grouping of like type data such as list of numbers, list of names etc.

Some examples where arrays can be used are

1) List of temperatures recorded every hour in a day, or a month, or a year.
2) List of employees in an organization.
3) List of products and their cost sold by a store.
4) Test scores of a class of students

5) List of customers and their telephone numbers.

Array:

An array is collection of same data type elements in a single entity.

Or

An array is collection of homogeneous elements in a single variable.

- It allocates sequential memory locations.
- Individual values are called as elements.

Types of Arrays:

* We can use arrays to represent not only simple lists of values but also tables of data in two or three or more dimensions.

- One – dimensional arrays
- Two – dimensional arrays
- Multidimensional arrays

ONE –DIMENSIONAL ARRAY:

A list of items can be given one variable name using only one subscript and such a variable is called a single – subscripted variable or a one – dimensional array.

Declaration of One-Dimensional Arrays :

- Like any other variables, arrays must be declared before they are used.
  The general form of array declaration is

Syntax:

<datatype> <array_name>[sizeofarray];

- The datatype specifies the type of element that will be contained in the array, such as int, float, or char.
- The size indicates the maximum number of elements that can be stored inside the array.
- The size of array should be a constant value.

Examples:

```
float  height[50];
```

- Declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid.

```
int   group[10];
```

- Declares the group as an array to contain a maximum of 10 integer constants.

char name[10];

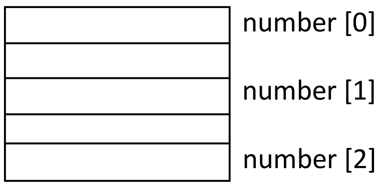
- Declares the name as a character array (string) variable that can hold a maximum of 10 characters.

Ex: int number[10];

- The subscript should begin with number „0".
  i.e. x[0].
- To represent a set of five numbers, say (35, 40, 20, 57, 19) by an array variable "number".
  We may declare the variable "number" as

  int number[5];

- Hence the computer reserves five storage locations as shown



- The values to the array elements can be assigned as

  number[0] = 35;

  number[1] = 40;

  number[2] = 20;

  number[3] = 57;

  number[4] = 19;

- This would cause the array number to store the values as

number[0]	35
number[1]	40
number[2]	20
number[3]	57
number[4]	19

Valid Statements :

- a = number[0] + 10;
- number[4] = number[0] + number[2];

- number[2] = x[5] + y[10];
- value[6] = number[i] * 3;

Example-1: Write a program to print bytes reserved for various types of data and space required for storing them in memory using array.

```
# include<stdio.h>

# include<conio.h>

main ( )

 {

    int    a[10];

    char   c[10];

    float  b[10];

    clrscr( );

    printf("the type „int‟ requires %d bytes", sizeof(int));

    pirntf(" \n The type „char‟ requires %d bytes", sizeof(char));

    printf(" \n The type „float‟ requires %d bytes", sizeof(float));

    printf(" \n %d memory locations are reserved for ten „int‟ elements", sizeof(a));

    printf (" \n %d memory locations are reserved for ten „char‟ elements",sizeof(c));

    printf (" \n %d memory locations are reserved for ten „float‟ elements",sizeof(b));

    getch( );

}
```

Output:

```
The type „int‟ requires 2 bytes

The type „char‟ requires 1 bytes

The type „float‟ requires 4 bytes

20 memory locations are reserved for ten „int‟ elements

10 memory locations are reserved for ten „char‟ elements

40 memory locations are reserved for ten „float‟ elements.
```

Example-2: Write a program using a single subscripted variable to read and display the array elements.

```
# include<stdio.h>
# include<conio.h>
main( )
{
   int   i ;
   float   x[10];
   printf(“Enter 10 real numbers: \n”);           /* reading values into Array */
   for (i=0; i <10; i++)
    {
         scanf(“ %f ”, &x[i]);
     }
         /* printing of x[i] values */
  printf(“The array elements are:”);
  for(i=0; i < 10; i++)
      {
         printf(“%d \t”, x[i]);
      }
  getch( );
 }
```

Output:

```
Enter 10 real numbers:
  1.1    2.2    3.3.    4.4    5.5    6.6    7.7    8.8    9.9    10.10
```

The array elements are:

```
1.1     2.2     3.3.     4.4     5.5     6.6     7.7     8.8     9.9     10.10
```

Data type and their required bytes

<u>Data type</u>	<u>Size</u>
char	1 byte
int	2 bytes
float	4 bytes
long	4 bytes
double	8 bytes

<u>Initialization of One –Dimensional Arrays:</u>

- After an array is declared, its elements must be initialized. Otherwise they will contain "garbage".
- An array can be initialized at either of the following stages.
  - (i) At compile time
  - (ii) At run time.

(1) <u>Compile Time Initialization</u>:

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

The general form of initialization of array is

datatype array_name[size] = { list of values };

The values in the list are separated by commas.

Example:

```
int   number[3] = {0,0,0};
```

- i.e., we will declare the variable number as an array of size 3 and will assign zero to each element.
- If the number of values in the list is less than the number of elements, then only that many elements will be initialized.

- The remaining elements will be set to zero automatically.
  Ex:

  float total[5] = { 0.0, 15.75, -10};

- The size may be omitted.
- In such cases, the compiler allocates enough space for all initialized elements.
  Ex:

  int counter [ ] = {1,1,1,1};

- This will declare the counter array to contain four elements with initial values 1.
- Character arrays may be initialized in a similar manner.
  char name[ ] = { „J‟, „o‟, „h‟, „n‟, „\0‟};

- This declares the name to be an array of five characters initialized with the string "John" ending with a null character.
- Alternative declaration is **char name[ ] = "John";**
- Compile time initialization may be partial. i.e., the number of initializers may be less than the declared size. In such cases the remaining elements are initialized to zero, if the array type is numeric.And NULL if the type is char.
  Ex: int number[5] = {10,20};

- Will initialize the first two elements to 10 & 20 respectively and the remaining elements to zero.
  Similarly

  char city[5] = {„B‟};

  will initialize the first element to „B‟ and the remaining four to NULL.

- If we have more initializers than the declared size, the compiler will produce an error.
  int number[3] = {10,20,30,40};

  will not work. It is illegal in C.

(2) <u>Run Time Initialization:</u>

- An array can be explicitly initialized at run time.
- This approach is usually applied for initializing long arrays.

Ex:

```
 - - - - - -

 - - - - - -

for(i=0; i<100; i++)

 {

   if (i<50)
```

```
            sum [i] = 0.0;
        else
            sum [i]= 1.0;
       }
    -----------
```

- Here the first 50 elements of the array "sum" are initialized to zero.
- While remaining 50 elements are initialized to 1.0 at run time.
- We can also use a read function such as „scanf" to initialize an array.
  Ex:

```
        int   x[3];
        scanf(" %d  %d  %d ", &x[0], &x[1], &x[2]);
```

- This will initialize array elements with the values entered through the key board.
- Character arrays are called strings.
- There is a slight difference between an integer array and a character array.
- In character array NULL („\0") character is automatically added at the end.
- In other types of arrays no character is placed at the end.
- Hence by using NULL character compiler detects the end of the character array.

Example-1: Write a program to display character array with their address.

```
    # include<stdio.h>
    # include<conio.h>
     void  main( )
       {
           char   name[10] = {„A", „R", „R", „A", „Y"};
            int  i = 0;
           clrscr( ) ;
           printf(" \n character memory location \n");
           while(name[i] ! = „\0")
             {
                  printf(" \n [%c] \t  [%u]", name[i], &name[i]);
                   i++;
             }
        }
```

Output:

```
Character Memory Location
            [A]            4054
            [R]            4055
            [R]            4056
            [A]            4057
            [Y]            4058
```

Example-2: Program to find out the largest and smallest element in an array.

```
# include<stdio.h>
#include<conio.h>
main( )
      {
         int  i,n;
         float  a[50], large, small;
         printf(“size of vector/value:”);
         scanf(“%d”, &n);
         printf(“ \n vector elements are \n”);
         for(i=0; i<n; i++)
           scanf(“ %f ”, &a[ i ]);
         large = a[0]; small
         = a[0]; for(i=1;
         i<n; i++)
           {
              if(a[ i ] > large)
                   large = a[ i ];
              else  if(a[ i ] < small)
                   small = a[ i ];
                  }
         printf(“\n Largest element in vector is %8.2f  \n”, large);
         printf(“ \n smallest element in vector is %8.2f \n”, small);
         getch( );
       }
```

Output:

```
        Size of vector : 7
        Vector elements
        34.00   – 9.00    12.00    10.00    - 6.00     0.00    36.00
        Largest element in vector is 36.00
        Smallest element in vector is – 9.00
```

Example-3:

Program to sort the vector elements in ascending order.

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

 main( )
  {
    int   i,j,k,n;
    float  a[50], temp;
    printf(“size of vector:”);
    scanf(“%d”, &n):
    printf(“ \n vector elements are : \n”);
    for( i=0; i < n; i++)
        scanf(“ %f  ”, &a[ i ]);
    for( i=0; i < n-1; i++)
        for( j=i+1;j<n; j++)
          {
              if(a[ i ] > a[ j ])
               {
                temp = a[ i ];
                a[ i ] = a[ j ];
                a[ j ] = temp;
              }
           }
   printf(“\n vector elements in ascending order : \n”);
   for( i=0; i<n; i++)
      printf(“ %8.2f ”, a[ i ]);
    getch( );
  }
```

Output:

```
Size of Vector : 8
Vector elements are
        91.00   20.00   1.00    7.00    34.00   11.00   -2.00   6.00
Vector elements in ascending order;
        -2.00   1.00    6.00    7.00    11.00   20.00   34.00   91.00
```

TWO DIMENSIONAL ARRAYS:

There could be situations where a table of values will have to be stored.

Consider a student table with marks in 3 subjects.

Student	Mat	Phy	Chem.
Student # 1	89	77	84
Student # 2	98	89	80
Student # 3	75	70	82
Student # 4	60	75	80
Student # 5	84	80	75

- The above table contains a total of 15 values.

- We can think this table as a matrix consisting of 5 rows & 3 columns.
- Each row represents marks of student # 1 in all (different) subjects.
- Each column represents the subject wise marks of all students.
- In mathematics we represent a particular value in a matrix by using two subscripts such as Vij.
- Here V denotes the entire matrix Vij refers to the value in " i "th row and " j "th column.

EXAMPLE:

In the above table V_{23} refers to the value „80‟.

- C allows us to define such tables of items by using two-dimensional arrays.

Definition:

A list of items can be given one variable name using two subscripts and such a variable is called a two – subscripted variable or a two – dimensional array.

Two – Dimensional arrays can be declared as.

```
<data type>    <array name>[row size] [column size] ;
```

- The above table can be defined in „C‟ as
  int V[5][3];

Representation of two Dimensional array in memory.

	column -0	column-1	column -2
	[0] [0]	[0] [1]	[0] [2]
Row. 0	89	77	84
	[1] [0]	[1] [1]	[1] [2]
Row. 1	98	89	80

	[2] [0]	[2] [1]	[2] [2]
Row. 2	75	70	82
	[3] [0]	[3] [1]	[3] [2]
Row. 3	60	75	80
	[4] [0]	[4] [1]	[4] [2]
Row. 4	84	80	75

Initializing Two- Dimensional Arrays:

- Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.
  int table[2] [3] = {0,0,0,1,1,1};
- This initializes the elements of first row to zero and the second row to one.
- This initialization is done row by row.
- The above statement can be equivalently written as
  int table[2][3] = {{0,0,0},{1,1,1}}**;**

  we can also initialize a two – dimensional array in the form of a matrix as shown.

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

Commas are required after each brace that closes of a **row**, except in case of last **row**.

- If the values are missing in an initializer, they are automatically set to zero.
  Ex: **int table [2] [3] = {**

```
                {1,1},          1  1  0
                {2}             2  0  0
              };
```

This will initialize the first two elements of the first row to one,

- The first element of the second row to two and all other elements to zero.
- When all the elements are to be initialized to zero, the following short-cut method may be used.

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

- The first element of each row is explicitly initialized to zero while the other elements are automatically initialized to zero.
- The following statement will also achieve the same result

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

(1) Write a program to display the elements of two dimensional array.

```
# include<stdio.h>

# include<conio.h>

void  main( )

{

   int  i,j;

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

   clrscr( );

   printf(“elements of an array \n \n”);

   for( i=0; i<3; i++)

    {

        for ( j=0; j<3; j++)

               printf (“%d\t”, a[ i ][ j ]);

            } /* end of inner for loop */

         printf(“\n”);

     }  /* end of outer for loop */

  getch( );

}  /* end of main() function */
```

Output:

```
Elements of an Array

        1       2       3

        4       5       6
```

```
7 8 9
```

(2) Write a program to display 2-Dimensional array elements together with their addresses.

```
# include<stdio.h>
# include<conio.h>
void  main( )
 {
       int  i,j;
       int  a[3][3] = {{1,2,3{,{4,5,6},{7,8,9}};
       clrscr( );
       printf(“Array Elements and address \n \n”);
       printf(“col-0        col -1   col-2      \n”);
       prinf(“--------      --------   ------- \ n”);
       printf(“Row 0\t”)
       for( i=0;  i<3; i++)
        {
          for( j=0; j<3; j++)
              printf(“%d [%u]”, a[ i ][ j ], &a[ i ][ j ]);
          printf(“ \n Row %d ”, i+1);
        }
      getch( );
   }
```

Output:

```
         Array Elements and address
                        Col – 0         col – 1         col -2
         Row 0          1 [4052]        2[4054]         3[4056]
         Row 1          4 [4058]        5[4060]         6[4062]
```

```
Row 2       7 [4064]     8 [4066]     9[4068]
```

(3) Program to read the matrix of the order upto 10 x 10 elements and display the same in matrix form.

```
# include<stdio.h>
 # include<conio.h>
void  main( )
 {
      int  i, j, row, col, a[10][10];
    clrscr( ) ;
    printf("\n Enter Matrix Order  upto (10 x 10) A :");
    scanf(" %d %d ", &row, &col);
    printf("\n Enter Elements of matrix A: \n");
    for( i=0; i<row ; i++)
      {
          for( j=0; j<col; j++)
            {
            scanf("%d", &a[ i ][ j ]);
          }
      }
    printf(" \n The matrix is: \n");
    for( i=0; i<row; i++)
      {
        for( j=0; j<col; j++)
          {
               printf(" %d ", a[ i ][ j ]);
            }
        printf("\n");
      }
    getch( );
  }
```

Output :

```
Enter order of matrix upto (10 x 10) A : 3   3
Enter Elements of a matrix A :
                              3     5     8
                              4     8     5
                              8     5     4
The matrix is
              3     5     8
              4     8     5
              8     5     4
```

(4) Program read the elements of the matrix of the order upto 10 x 10 & transpose its elements.

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

```
            {
              int   i,j, row, col, a[10][10], b[10][10];
              clrscr( );
              printf(" \n Enter order of matrix upto (10 x 10) A:");
              scanf(" %d %d ", &row, &col);
              printf("\n Enter Elements of matrix A: \n");
              for( i=0; i < row; i++)
                {
                  for( j=0; j<col; j++)
                      scanf(" %d ", &a[ i ][ j ]);
                }
      /* transposing logic simply copying one matrix elements to another in reverse
   order */
              for( i=0; i < row; i++)
                {
                  for( j=0; j < col; j++)
                        b[ j ][ i ]=a[ i ][ j ];
                }
              printf(" \n The Matrix Transpose is \ n");
              for( i=0; i<row; i++)
                {
                 for( j=0; j<col; j++)
                    printf("%d", b[ i ][ j ]);
                 printf (" \ n");
                }
              getch( );    }
```

Output:

```
              Enter order of matrix upto (10 * 10) A : 3    3
              Enter Elements of matrix A:
                                        3      5      8
                                        4      8      5
                                        8      5      4
              The Matrix  Transpose is
                                        3      4      8
                                        5      8      5
                                        8      5      4
```

(5) Program to perform addition and subtraction of two matrices. Whose orders are upto 10 x 10.

```
          # include<stdio.h>
          # include<conio.h>
          main(  )
           {
              int  i,j,r1,c1, a[10][10], b[10][10];
              clrscr(  );
              printf("Enter Order of Matrix A & B upto 10 x 10:");
```

```
     scanf("%d %d", &r1, &c1);
     printf("Enter Elements of Matrix of A: \n");
     for( i=0; i < r1; i++)
       {
          for( j=0; j<c1; j++)
               scanf(" %d ", &a[ i ][ j ]);
       }
     printf("Enter Elements of Matrix of B: \ n");
     for( i=0; i < r1; i++)
       {
         for( j=0; j < c1; j++)
               scanf(" %d ", &b[ i ][ j ]);
       }
     printf("\n Matrix Addition \n");
     for( i=0; i < r1; i++)
       {
         for( j=0; j < c1; j++)
               printf("%d\t", a[ i ][ j ] + b[ i ][ j ]);
          printf (" \n");
       }
     printf("n Matrix Subtraction/Difference \n");
     for( i=0; i < r1; i++)
       {
         for( j=0; j < c1; j++)
               printf("%d\t", a[ i ][ j ] – b[ i ][ j ]);
         printf("\n");
       }
     getch( );
   }
```

Output:

```
Enter order of Matrix A & B upto 10 x 10 : 3   3
Enter Elements of Matrix of A:
4       5       8
2       9       8
2       9       4
Enter Elements of Matrix of B:
1       3       5
0       5       4
6       7       2
Matrix Addition
5       8       13
2       14      12
8       16      6

Matrix Subtraction
3       2       3
```

```
2    4    4
-4   2    2
```

(6) Write a C program to multiply A matrix of order mxn with B matrix of order nxl.

Solution: Consider two matrices A and B of order 2 x 2. Multiply them to get the resultant matrix C.

i.e. $A_{m \times n}$ X $B_{n \times l}$ produces the resultant matrix $C_{m \times l}$

$$\begin{bmatrix} 2 & -2 \\ 0 & 4 \end{bmatrix}\begin{bmatrix} 6 & 2 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 14 \\ 16 & 20 \end{bmatrix}$$

```
 /* program for matrix multiplication * /
# include<stdio.h>
# include<conio.h>
# include<math.h>
 main( )
 {
      int  a[10][10], b[10][10], c[10][10], m , n , i , j , l , k ;
      clrscr( ) ;
      printf(“ \n Enter Order of A matrix :”);
      scanf(“%d %d”, &m, &n);
       /* loop to read values of A matrix */
      printf(“Enter a matrix \n”);
      for( i=0; i<m; i++)
          for ( j=0; j<n; j++)
                scanf(“%d”, &a[ i ][ j ]);
      printf(“\n Enter order of B matrix:”);
      scanf(„%d %d”, &n, &l);
      /* loop to read values of B matrix */
      printf(“Enter B matrix \n”);
      for( i=0; i<n; i++)
          for ( j=0; j<l; j++)
                scanf(“%d”, &b[ i ][ j ]);
/* loop to multiply two matrices */
      for( i=0; i<m; i++)
       {
           for ( j=0; j<l; j++)
            {
                c[ i ][ j ] = 0;
                for( k=0; k < n; k++)
                     c[ i ][ j ] = c[ i ][ j ] +a[ i ][ k ] * b[ k ][ j ];
            }
       }
      /* loop to print resultant matrix */
      printf(“ \n Resultant matrix is \n”);
      for( i=0; i<m; i++)
       {
          for( j=0; j<l; j++)
```

```
                    printf(“%6d”, c[ i ][ j ]);
                printf(“ \n ”);
              }
             getch ( ) ;
          }
```

Explanation:

When this program is executed, the user has to first enter the order (min) of A

matrix and its values and its values.

The innermost loop

```
for (k=0; k<n; k++)
c[ i ][ j ] = c[ i ][ j ] + a[ i ][ k ] * b[ k ][ j ];
```

is used to multiply row elements of A matrix with respective column elements of B matrix and add the result to get an element for C matrix.

This is repeated in the outer loops to get the other elements in the resultant matrix.

Output:

```
Enter order of A matrix :   2          2
Enter A matrix
        2       -2
        0         4
Enter order of B matrix : 2           2
Enter B matrix
                6       2
                4       -5
Resultant matrix is
                4       14
                16      -20
```

(8) Write a C program to find the trace of a given square matrix of order mxm.

Solution

We know that the trace of a matrix is defined as the sum of the leading diagonal elements.

* Note that trace is possible only for a square matrix.

Ex:

$$A = \begin{bmatrix} 3 & 2 & -1 \\ 4 & 1 & 8 \\ 6 & 4 & 2 \end{bmatrix}$$

Trace of A matrix = $A_{11}+A_{22}+A_{33}=3+1+2=6$

- Row i and column j are equal for a diagonal element.

/* Program to find trace of square matrix * /

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

```
    {
            int  a[10][10], m,i,j, sum;
            clrscr( );
            printf (“\n Enter order of the square matrix :”) ;
            scanf (“%d”, &m);
            /* loop to read values of A matrix * /
            printf (“ \n Enter the matrix \n”);
            for( i=0; i<m;i++)
              for ( j=0; j<m; j++)
                    scanf (“%d”, &a[ i ][ j ]);
            /* loop to find trace of the matrix * /
            sum = 0;
            for ( i=0; i<m; i++)
               sum = sum + a[ i ][ i ];
            printf (“\n trace of the matrix = %d”, sum);
            getch( ) ;   }
```

- When this program is executed, the user has to enter the order m & values of the given matrix.
- A for loop is written to find the sum of the diagonal elements.
- The index variable of loop i is used for row & column subscripts to represent the diagonal elements.

Output :

```
        Enter order of the square matrix 3
        Enter the matrix
                3       2       -1
                4       1       8
                6       4       2
        Trace of the matrix = 6
```

MULTI –DIMENSIONAL ARRAY:

A list of items can be given one variable name using more than two subscripts and such a variable is called Multi – dimensional array.

Three Dimensional Array:

A list of items can be given one variable name using three subscripts and such a variable is called Three – dimensional array.

Declaration of Three-Dimensional Arrays :

Syntax:

<datatype> <array_name>[sizeofno.oftwoDimArray] [sizeofrow] [sizeofcolom];

- The datatype specifies the type of elements that will be contained in the array, such as int, float, or char.

Initializing Three- Dimensional Arrays:

- Like the one-dimensional arrays, three-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

  int table[2][2][3] = {0,0,0,1,1,1,6,6,6,7,7,7};

- This initializes the elements of first two dimensional(matrix) first row to zero‟s and the second row to one‟s and second matrix elements are first row to six‟s and the second row to seven‟s.
- This initialization is done row by row.
- The above statement can be equivalently written as

  int table[2][3] = { {{0,0,0},{1,1,1}}**,** {{0,0,0},{1,1,1}}**}**

we can also initialize a two – dimensional array in the form of a matrix as shown.

```
int table[2][3] = {
                        {
                          {0,0,0},
                          {1,1,1}
                         },
                           {
                        {6,6,6},
                        {7,7,7}
                      }
         } ;
```

STRING MANIPULATIONS IN C

- In C language, an array of characters is known as a string.

```
char st[80];
```

- This statement declares a string array with 80 characters.
- The control character „\0‟ which represents a null character is placed automatically at the end of any string used.

STRING HANDLING FUNCTIONS IN C:

There are four important string Handling functions in C language.

(i) **strlen( )** function
(ii) **strcpy( )** function
(iii) **strcat( )** function
(iv) **strcmp( )** function

(I) strlen( ) Function:

strlen( ) function is used to find the length of a character string.

Ex:

```
int  n;
char st[20] = “Bangalore”;
n = strlen(st);
```

- This will return the length of the string 9 which is assigned to an integer variable n.
- Note that the null charcter „\0‟ available at the end of a string is not counted.

(II) strcpy( ) Function:

strcpy( ) function is used to copy from one string to another string.

Ex :

```
char  city[15];
strcpy(city, “BANGALORE”) ;
```

This will assign the string “BANGALORE” to the character variable city.

* Note that character value like

city = “BANGALORE”; cannot be assigned in C language.

(III) strcat( ) Function:

strcat( ) function is used to join character, Strings. When two character strings are joined, it is referred as concatenation of strings.

Ex:

```
char city[20] = “BANGALORE”;
```

char pin[8] = “-560001”;

strcat $\left(\frac{city}{L}, \frac{pin}{R}\right)$;

- This will join the two strings and store the result in city as “BANGALORE – 560001”.
- Note that the resulting string is always stored in the left side string variable.

(iv) **strcmp( ) Function:**

strcm ( ) function is used to compare two character strings.

- It returns a 0 when two strings are identical. Otherwise it returns a numerical value which is the different in ASCII values of the first mismatching character of the strings being compared.

```
char city[20] = “Madras”;
char town[20] = “Mangalore”;
strcmp(city, town);
```

- This will return an integer value „- 10‟ which is the difference in the ASCII values of the first mismatching letters „D‟ and „N‟

* Note that the integer value obtained as the difference may be assigned to an integer variable as follows:

```
int  n;
n =  strcmp(city, town);
```

READING / WRITING STRINGS:

- We use scanf ( ) function to read strings which do not have any white spaces.
  Ex: **scanf(“ %s ”, city);**
- When this statement is executed, the user has to enter the city name (eg “NEWDELHI”) without any white space (i.e not like “NEW DELHI”).
- The white space in the string will terminate the reading and only “NEW” is assigned to city.
- To read a string with white spaces gets( ) function or a loop can be used as.
  (i) gets(city);
  (ii) do … while loop

```
 i = 0;
do
  {
     Ch = getchar( );
     city[ i ] = ch;
     i++;
  }
while(ch ! = „ \n ‟) ;
 i - - ;          city[ i ] = „\0‟;
```

- The copy or assign a single character to a string variable, the assignment can be written as given below;

```
city[ i ] = „N‟;
```

- When a string contains more than one character, the assignment is written as strcpy (city, “NEW DELHI”);

atoi( ) Function:

atoi( ) function is a C library function which is used to convert a string of digits to the integer value.

```
char st[10] = “24175” ;
int n;
n = atoi(st);
```

This will assign the integer value 24175 to the integer variable n.

(1) Write a C program to count the occurrence of a particular character in the given string.

Solution:

Consider a string “MISSISSIPPI”. Count the appearance of a character, say „S‟.

Program:

```
\* program to count a character in a string * /
# include<stdio.h>
# include<conio.h>
# include<string.h>
main( )
{
        char  st[20], ch;
        int  count, l,i;
        clrscr( );
        printf(“ \n Enter the string:”);
        gets(st);
        printf(“ \n which char to be counted ?”);
        scanf(“ %c”, &ch);
 /* loop to check the occurrence of the character * /
        l = strlen(st);
        count = 0;
        for( i=0; i < l; i++)
           if(st[ i ] = = ch)
           count ++;
        printf(“ \n the character %c occurs %d times”, ch, count);
        getch( ) ;
}
```

- When this program is executed, the user has to enter the string and the character to be counted in the given string.

Output:

```
Enter the string : MISSISSIPPI
```

```
Which char to be counted? S
The character S occurs 4 times.
```

(2) Write a C program to count the number of vowels present in a sentence.

Solution

Consider a sentence "this is a book". Count the appearance of vowels AEIOU in capital or small letters.

Program

```
/* program to count vowels * /
# include<stdio.h>
# include<conio.h>
# include<string.h>
main( )
 {
        char  st[80], ch;
        int  count = 0, i;
        clrscr( );
    /* loop to read a string * /
        printf(" \n Enter the sentence: \n");
        gets(st);
   /* loop to count the vowels in the string * /
        for( i=0; i<strlen(st); i++)
           switch(st [i ])
             {
                case  „A‟:
                case  „E‟:
                case   „I‟:
                case  „O‟:
                case  „U‟:
                case   „a‟:
                case   „e‟:
                case   „i‟:
                case   „o‟:
                case  „u‟:
                                count ++;
                                break;
           }
        printf("\n %d vowels are present in the sentence", count);
        getch( );
   }
```

- When this program is executed, the user has to enter the sentence.
- Note that gets( ) function is used to read the sentence because the stirng has white spaces between the words.
- The vowels are counted using a switch statement in a loop.

- Note that a count++ statement is given only once to execute it for the cases in the switch statement.

Output:

```
Enter the sentence:
This is a book
5 vowels are present in the sentence.
```

(3) Write a C program to test whether a given string is palindrome string. explain the working of program.

Solution: Consider the string „LIRIL" when it is reversed, it reads again as "LIRIL".

Any string of this kind is called a palindrome string.

- Few other palindrome strings are
  DAD, MOM, MADAM, MALAYALAM

```
/* program to check for palindrome stirng */
# include<stdio.h>
# include<conio.h>
# include<string.h>
  main( )
   {
        char st[20], rst[20];
        int i,j;
        clrscr( );
        printf(„\n Enter the string:");
        scanf("%s", st);
/* loop to reverse the string */
        i=0;
        j=strlen(st)-1;
        while( j >= 0 )
         {
           rst[ i ] = st[ j ];
           i++;
           j--;
          }
        rst[ i ] = „\0‟;
        if(strcmp(st,rst)==0)
                printf("\n %s is a palindrome string", st);
        else
                printf("\n %s is not a palindrome string", st);
        getch( );
     }
```

- When this program is executed, the user has to read a string.

```
              0  1  2  3  4  5  6  7  8← j
St =        | H | Y | D | E | R | A | B | A | D |   |
```

- To reverse this string, the original string is copied from the last position & written as

i→ 0 1 2 3 4 5 6 7 8

rst =

D	A	B	A	R	E	D	Y	H	

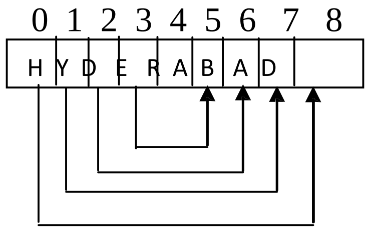
- The two strings are compared using a strcmp ( ) function which will return a 0 when the original string and the reversed string are identical.
- Based on this value the result is printed.

Output:

```
Enter the string : HYDERABAD
Hyderabad is not a palindrome string.
```

Alternative Method:

- This program can also be written to compare the first character of the string with the last, second with the second last and so on up to the middle of the string as shown.



- If all the characters are identical, then the string is a palindrome string.
- Otherwise the loop will be terminated.

(4) Write a C program to compare two strings which are given as input through keyboard and print the alphabetically greater string.

Solution

The alphabetically greater string is the one whose ASCII value of the first letter is greater than that of the second string.

Consider the two character strings.

St1 = "ALPHA"

St2 = "BEETA"

St2 is greater than st1 because the ASCII value of „B‟ in "BETA" is greater than that of „A‟ in „ALPHA‟.

- Note that when the first letters of the two strings are identical, the second letters of the strings are compared.

```
/* PROGRAM TO PRINT ALPHABETICALLY GREATER STRING * /
# include<stdio.h>
# include<conio.h>
# include<stirng.h>
main( )
```

```
{
        char  st1[20], st2[20];
        clrscr( );
        printf(" \n Enter string 1:");
        scanf( " %s ",st1);
        printf(" \n Enter string 2:");
        scanf( " %s ", st2);
        if(strcmp(st1,st2)> 0)
            printf("\n %s is alphabetically greater string", st1);
        else
            printf(" \n %s is alphabetically greater string", st2);
        getch( );
}
```

- When this program is executed, the user has to enter the two strings.
- Note that strcmp( ) function returns a positive value when string 1 is greater & a negative value when string 2 is greater.

Output:

```
Enter string 1 : ACPHA
Enter string 2 : BETA
BETA is alphabetically greater string.
```

(5) Write a C program to read an array of names and to sort them in alphabetical order (dictionary).

Solution:

Consider a list of names stored in a 2-Dimensional character array. Each row can store a name of a maximum length of 10 letters.

Names 0 1 2 3 4 5 6 7 8 9

0	D	E	E	P	A	K			
	S	H	E	R	I	N			
1	S	O	N	I	K	A			
2	A	R	U	N					

- Compare the name in the first row with the Second, then with the third name and so on… till the smallest name (& name starting with letter „A") moves to the first place.

Names

0	A	R	U	N					
	S	H	E	R	I	N			
1	S	O	N	I	K	A			
	D	E	E	P	A	K			

- Now compose the second name with the third then with the fourth name & so on.. till the alphabetically second smallest name none to the second place.

Names

0	A	R	U	N					
	S	H	E	R	I	N			
1	S	O	N	I	K	A			
	D	E	E	P	A	K			

This procedure is repeated till the names are arranges in alphabetical order shown below.

0	A	R	U	N					
	D	E	E	P	A	K			
1	S	H	E	R	I	N			
	S	O	N	I	K	A			

Step involved in arranging names in alphabetical order are given below.

Step 1 : Read n
Step 2 : Loop to read „n" names of the list
Step 3 : Loop to arrange the names by comparison
Step 4: Loop to print the arranged list.
Step 5 : Stop

PROGRAM TO ARRANGE NAMES IN ALPHABETICAL ORDER.

```
# include<stdio.h>
# include<conio.h>
# include<string.h>
main( )
  {
        char  names[50][20], temp[20];
        int n,i,j;
        clrscr( ):
        printf(" \n How many names?");
        scanf("%d", &n);
        printf(" \n Enter the %d names one by one \ n",n);
        for( i=0; i<n; i++)
        scanf("%s", names[ i ]);
/* loop to arrange names in alphabetical order * /
        for( i=0; i<n-1; i++)
          for ( j =i+1; j<n; j++)
                if(strcmp(names[ i ], names[ j ]) > 0)
                  {
                    strcpy( temp, names[ i ]);
                    strcpy(names[ i ], names[ j ]);
                    strcpy(names[ j ],temp);
/* loop to print the alphabetical list of names * /
        printf (“ \n names in alphabetical order”);
        for( i=0; i<n; i++)
           printf(“\n %s”, names[ i ]);
        getch( );
    }
```

- When this program is executed, the user has to first enter the total no of names (n) and then all the names in the list.
- The names are compared, rearranged and printed in alphabetical order.

Output:

```
How many names? 4
Enter the 4 names one by one
        DEEPAK
        SHERIN
        SONIKA
        ARUN

Names in Alphabetical order
        ARUN
        DEEPAK
        SHERIN
        SONIKA
```

(6) Write a C program to convert a line in the lower case text to upper case.

Solution

Consider ASCII values of lower and upper case letters

A – 65 B – 66 …………………… Z – 90

a - 97 b – 98 …………………… z – 122

* Note that the difference between the lower and upper case letters (ie. – 32) is used for conversion.

```
/* PROGRAM TO CONVERT LOWER CASE TEST TO UPPER CASE * /
    # include<stdio.h>
    # include<conio.h>
    # include<string.h>
    main( )
    {
        char  st[80];
        int  i;
        clrscr( );
        printf(“ \ n Enter a sentence : \ n”);
        gets(st);
    /* loop to convert lower case alphabet to upper case * /
        for( i=0; i<strlen(st); i++)
           if(st[ i ] >= „a‟ && st[ i ] <= „z‟)
```

```
                st[ i ] = st[ i ] – 32;
            printf(“ \n the converted upper case strings \n %s”, st);
            getch( );
    }
```

OUTPUT:

```
            Enter a sentence:
            Logical thinking is a must to learn programming
            The converted upper case string is
            LOGICAL THINKING IS A MUST TO LEARN PROGRAMING.
```

(7) Write a C program to count no of lines, words and characters in a given text.

Solution:

A while loop is used to read the text. The character „$‟ is used to terminate the reading of text.

Program

```
# include<stdio.h>
# include<string.h>
# include<conio.h>
main( )
  {
        char  txt[250], ch, st[30];
        int  ins, wds,  chs,  i;
        clrscr( ) ;
        printf(“ \n Enter the text, type $ st end \n \n”);
        i=0;
        while((txt[i++]= getchar( ) ) ! = „$‟);
        i--;
        st[ i ] = „\0‟ ;
        ins = wds = chs = 0;




/ * loop to count lines, words & characters in text * /
        i=0;
        while(txt[ i ]!=‟$‟)
          {
            switch(txt[ i ])
                {
                        case   „,‟:
                        case   „!‟:
                        case   „\t‟:
                        case  „  ‟:
                                {
                                   wds ++;
```

```
                                    chs ++;
                                     break;
                                   }
                         case  „?‟:
                         case  „.‟:
                                 {
                                    wds  ++;
                                    chs   ++;
                                    break;
                                  }
                         default:    chs ++;
                                   break;
                     }
                  i++;
                }
          printf(“\n\n no of char (incl.blanks) = %d”, chs);
          printf(“\n No. of words = %d”, wds);
          printf(“\n No of lines = %d”, lns);
          getch(  ) ;
        }
```

Output:

```
Enter the text, type $ at end
What is a string? How do you initialize it? Explain with example.
With example. $
No of char (inch. Blanks) = 63
No of words = 12
No of lines = 3.
```

<u>Additional String Handling Functions:</u>

Some „C‟ compilers will accept the following string handling functions which are available in header files string.h and ctype.h

<u>Functions:</u>

(i) <u>strupr( )</u>: to convert all alphabets in a string to upper case letters.

Ex:

strupr(“ delhi ”) ⇒ “ DELHI”

(ii) <u>strlwr( )</u>: To convert all alphabets in a string to lower case letters.

Ex:

strlwr(“ CITY ”) ⇒ “city”

(iii) **strrev( ):** To reverse a string
Ex: strrev(“ SACHIN ”) ⇒ “NIHCAS”

(iv) **strncmp( ):** To compare first n characters of two strings.
Ex:
m = strncmp (“ DELHI ”, “ DIDAR ”, 2);
⇒ m = -4

(v) **strcmpi( ):** To compare two strings with case in sensitive (neglecting upper / lower case)
Ex:
m=strcmpi(“ DELHI ”, “ delhi ”); ⇒ m = 0.

(vi) **strncat( ):** To join specific number of letters to another string.
Ex.
```
char  s1[10] = “New”;
char  s2[10] = “Delhi -41”;
strncat(s1,s2,3);
```
⇒ s1 will be “NewDel”.

Operations with Characters:

C language also supports operations with characters and these functions are available in the header file “ctype.h”.

*** Note that the value „1‟ or positive integer will be returned when a condition is true, and value „0‟ will be returned when condition is false.**

- Some „ctype.h‟ functions are listed below.

Function:

(i) **isupper( ) :** To test if a character is a upper case letters.
Ex:
isupper(„A‟) ⇒ 1
isupper(„a‟) ⇒ 0
isupper(„8‟) ⇒ 0

(ii) **islower( ) :** To test if a charcter is a lower case letter.
Ex:
islower(„n‟) ⇒ 1

(iii) **isalpha( ) :** To test if a character is an alphabet.
Ex:
isalpha(„K‟) ⇒ 1
isalpha(„+‟) ⇒ 0

(iv) **isalnum( ) :** To test if a character is an alphabet or number.
Ex:
isalnum(„8‟) ⇒ 1
isalnum(„y‟) ⇒ 1

isalnum(„-‟) ⇒ 0

(v) **isdigit( )** **:** To test if a character is a number.

Ex:

isdigit(„6‟) ⇒ 1

isdigit(„a‟) ⇒ 0

(vi) **isxdigit( )** **:** To test if a character is a Hexa decimal number (0-9, A-F and a-f are Hexa decimal digits)

Ex:

isxdigit(„9‟) ⇒ 1

isxdigit(„A‟) ⇒ 1

isxdigit(„M‟) ⇒ 0

(vii) **tolower( )** **:** To convert an alphabet to lower case letter

Ex:

tolower(„A‟) ⇒ a

(viii) **toupper( )** **:** To convert an alphabet to upper case letter

Ex:

toupper(„a‟) ⇒ A.

POINTERS

INTRODUCTION:

When variables are declared memory is allocated to each variable.

C provides data manipulation with addresses of variables therefore execution time is reduced. Such concept is possible with special data type called pointer. A pointer is a variable which holds the address of another variable or identifier this allows indirect access of data.

DECLARATION:

1. To differentiate ordinary variables from pointer variable, the pointer variable should
   proceeded by called „value at address‟ operator. It returns the value stored at particular address. It is also called an indirection operator(symbol *****).
2. Pointer variables must be declared with its type in the program as ordinary variables.
3. Without declaration of a pointer variable we cannot use in the program.
4. A variable can be declared as a pointer variable and it points to starting byte address of
   any data type.

SYNTAX:

Data type *pointer variable;

The declaration tells the compiler 3 things about variable p

a) The asterisk (*) tells that variable p is a pointer variable
b) P needs a memory location
c) P points to a variable of type data type.

Ex:

```
int   * p, v;

float  *x;
```

In the first declaration v is a variable of type integer and p is a pointer variable.

V stores value p stores address

In the 2nd declaration x is a pointer variable of floating point data type.

2) Pointer variables are initialized by p= &v, it indicates p holds the starting address of

integervariable v.

```
int   v=20,*p;
  p=&v;        /*address of v stored in p*/
        V               p
        20             1000
```

Let us assume that address of v is 1000.
This address is stored in p. Now p holds address of v now *p gives the value stored at the address pointer by p i.e *p=20.

IMPORTANT POINTS:

1. A pointer variable can be assigned to another pointer variable, if both are pointing to the same data type.
2. A pointer variable can be assigned a NULL value.
3. A pointer variable cannot be multiplied or divided by a constant.

   Ex: p*3 or p/3 where p is a pointer variable

4. One pointer variable can be subtracted torn another provided both winters points to Elements on same away.

5. C allows us to add integers to or subtract integers from pointers.

   Ex: p1+4, p2-2, p1-p2
   If both p1, p2 are pointers to same way then p2-p1 gives the number of elements between p1,p2
6. Pointer variable can be intermingled or decremented p++ & p- -
   The values of a variable can be released in to way
   1.by using variable name

2.by using adders.

Disadvantages:

1. Unless pointer is defined and initialized properly use of pointers may lead to disastrous situations.
2. Using pointers sometimes be confusions.

Program:

```
#incude < stdio.h>

main ( )

{

        int x = 10, * p1, *p2, *p3;

        P1=p2=p3=&x;

        printf ("%d",*p1);

        printf ("% d",*p2);

        printf ("%d",*p3);

        printf ("%d", x);

        printf ("%d",*(& x));

}

Output : 10 10 10 10 10
```

<u>**Void Pointers :**</u>

We can declare a pointer to be of any data type void and can assign a void pointer to any other type.

```
Ex : int x:
     void p;
     p= &x;
    *p = 27
printf("%d",*p);
```

It given error because the pointer p is of type void and cannot hold any value. So we have to type cast the pointer variable from one type to other type.

```
int x = 27;
void *p;
p=&x;
printf (" %d", * ((int*)p));
```

Output is 27

The statement (int *)p makes p to become an integer type

```
Ex: #include < stdio.h>
    #include < conio.h>
    main()
       {
              int x = 27;
              void *p;
              p= &x;
              printf (“value is = %d”, * ((int *)p));
       }

       Output : Value is 27
```

Pointer and functions

1. Pointers can be passed as argument to a function definition.
2. Passing pointers to a function is called call by value.
3. In call by reference what ever changes are made to formal arguments of the function definition will affect the actual arguments in calling function.
4. In call by reference the actual arguments must be pointers or references or addresses.
5. Pointer arguments are use full in functions, because they allow accessing the original data in the calling program.

```
Ex1: void swap (int *, int *);
     main ( )
        {
               int a=10, b=20;
               swap(&a, &b);  /* a,b are actual parameters */
               printf (“ %d %d ”, a ,b);
        }
     void swap (int *x , int *y)     /* x,y are formal parameters */
        {
               int t;
               t = *x;
               *x=*y;
               *y=t;
        }
   Output:   20  10
```

Ex 2:

```
# include < stdio.h>
# include < conio.h>
void copy (char *, char *);
```

```
main ( )
 {
         char a[20], b[20];
         printf (“ enter string a:”);
         gets(a);
         printf (“ enter string b:”);
         gets(b);
         printf (“ before copy “ );
         printf (“ a=%s  and b=%s ”, a , b);
         copy(a ,b);
         printf (“after copy “);
         printf (“ a=%s and b=%s”, a , b);
 }
void copy (char *x, char *y)
 {
         int i=0;
         while ((X[i]=Y[i]) != „\0‟)
           i++;
 }
```

Output:

```
enter string a: computer
enter string b: science
before copy
a=computer and b= science
after copy
a=science and b=science
```

Pointer and arrays :

Pointer can be used with array for efficient programming. The name of an array itself indicates the stating address of an array or address of first element of an array. That means array name is the pointer to starting address or first elements of the array. If A is an array then address of first element can be expressed as &A[0] or A.The compiler defines array name as a constant pointer to the first element.

Ex:
```
#include<stdio.h>
#include< conio.h>
void disp (int *, int);
main( )
   {
         int i, a[10],n;
         printf (“ enter the size of an array:”);
         scanf (“%d”, &n);
         printf (“ enter the array elements:”);
         for (i=0; i<n; i++)
            scanf (“%d”, &a[i]);
         disp (a,n);
```

```
            printf(“ array elements after sorting ” );
            for(i=0;i<n;i++)
              printf (“%d”, a[i]);
    }

  void disp(int a[ ], int n)
    {
            int i,j, temp;
            for (i=0; i<n; i++)
              for (j=0; j<n; j++)
                {
                   if (a[j] > a[i])
                     {
                        temp = a[ j ];
                        a[ j ] = a[ i ];
                        a[ i ] = temp;
                      }
                }
    }
```

Output :

```
    enter the size of an array: 5
    enter the array elements: 20 30 10 50 40
    array elements after sorting: 10 20 30 40 50
```

<u>Pointer and Character array</u>

Pointers are very useful in accessing character arrays. The character strings can be assigned with a character pointer.

Ex:

```
        char array[ ] = “Love India”;        array version
        char *p= “Love India”;              pointer version
```

Here p points to the first character in a string.

Ex1:

```
#include<stdio.h>
#include<conio.h>
main()
 {
        int i;
        char *p= “ Love India”;
        clrscr();
        while (*p! = „\0‟)
          {
                printf (“ %c “, *p);
                p++;
          }
}
Output: Love India
Ex2:
# include <stdio.h>
# include <conio.h>
int slen(char *);
main ( )
  {
          char name[50];
          int k;
          printf (“ enter string: “);
          gets (name);
          k = slen (name);
          printf (“ the string length is %d ” , k);
  }
int slen (char *s)
  {
          int n;
          for (n=0;  (*s) != „\0‟; n++)
               s++;
          return (n) ;
  }
Output:   enter string: College
          the string length is 7
```

Pointer Arithmetic:

An integer operand can be used with a pointer to move it to a point / refer to some other address in the memory.

consider an int type ptr as follows

```
                                      65494
Int * mptr ;             mptr  [          ]
```

Assume that it is allotted the memory address 65494 increment value by 1 as follows .

```
mptr ++;               or              ++ mptr;
mptr = mptr +1;        or              mptr + =1 ;
```

++ and - - operators are used to increment, decrement a ptr and commonly used to move the ptr to next location. Now the pointer will refer to the next location in the memory with address 64596.

C language automatically adds the size of int type (2 bytes) to move the ptr to next memory location.

mptr = mtpr + 1

= 645.94 + 1 * sizeof (int)
= 65494 + 1 * 2

Similarly an integer can be added or subtract to move the pointer to any location in RAM. But the resultant address is dependent on the size of data type of **ptr**.

The step in which the ptr is increased or reduced is called scale factor.Scale factor is nothing but the size of data type used in a computer.

We know that in a pc,
The size of float = 4
char = 1
double = 8 and so on

Ex: float *xp; (assume its address = 63498)

xp = xp + 5;

Will more the ptr to the address 63518

- ⇨ 63498 + 5 * size of (float0
- ⇨ 63498 + 5*4
- ⇨ 63498+20
- ⇨ 63518

Rules for pointer operation:

The following rules apply when performing operations on pointer variables

1. A pointer variable can be assigned to address of another variable.
2. A pointer variable can be assigned the values of another pointer variables.
3. A pointer variable can be initialized with NULL or 0 value.
4. A pointer variable can be prefixed or postfixed with increment and decrement operator.
5. An integer value may be added or subtracted from a pointer variable.
6. When 2 pointers points to the same array, one pointer variable can be subtracted from another.
7. when two pointers points to the objects of save data types, they can be compared using relational .
8. A pointer variable cannot be multiple by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address.

UNIT –V : STRUCTURES & UNIONS

Data in the array is of the same composition in native a fan as type is concerned. In real life we need to have different data types for ex. To maintain the employee information. We should have information such as name, age, qualification, salary etc. here to maintain the information of employee dissimilar data types required. Name & qualification are char data type age is integer, & salary is float. All these data types cannot be expressed in a single away. One may think to declare different always for each data type. Hence, always cannot be useful her for tackling such mixed data types, a special feature is provided by C, known as Structure.

Structure is a collection of heterogeneous type of data i.e. different types of data. The various individual components, in a structure can be accessed is processed separately. A structure is a collection of variables referenced under a name, providing a convenient means of keeping related information. A structure declaration forms a template that may be used to create structure objects.

Difference between structure is arrays :-

	Structure	Arrays
1.	Collection of heterogeneous types of data i.e. different types of data.	1. Collection of homogeneous types of data i.e. same types of data
2.	A structure element component of structure has a unique name.	2. An away it is referred by its position i.e. index.

The point on which array & structures can be similar is that both array & structure must be definite number of components.

Features of Structures:-

To copy elements of one array to another array of same data type elements are copied one by one. It is not possible to copy elements at a time. Where as in structure it is possible to copy the contents of all structure elements of different data types to another structure var of its type using assignment (=) operator. It is possible because structure elements are stored in successive memory locations. Nesting of structures is possible.

It is also possible to create structure pointers. We can also create a pointer. Pointing to structure elements.

For this we require „ → ‟ operator.

➔ Declaration of structure :-

```
Struct    Struct_type

{

type val 1; // members of structures

type val 2;

};
```

After defining structure we can create variables as given

Struct struct_type v1, v2, v3.

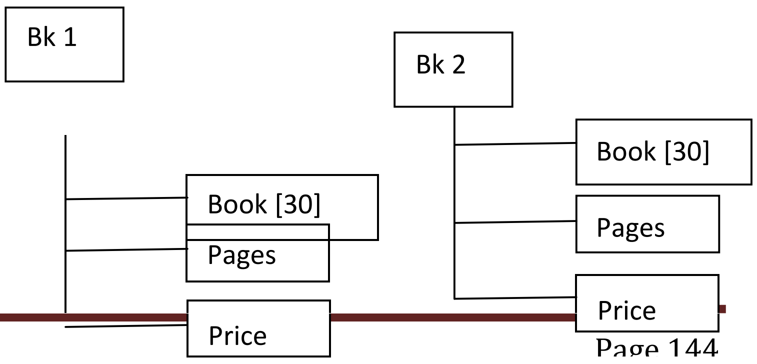
The declaration defines the structure but this process does not allocate memory. The memory allocation takes only when var.r declared

Example:-

Struct book

{

Char book [30];



```
        int pages;
        float price;
};
Strict book bk1, bk2;
```

Initialization:-

Struct book bk1= {“siri”, 500, 38500};

All the members of structure are related to variable bk1 structure – var. member i.e. bk1.book

The period (.) sign is used to access the struct members.

- ⇨ WAP to define & initialize a structure & its variables

```
 main ( )
{
        Strict book
        {
                char book [30];
                int pages;
                float price;
         };
        struct book bk1 = { “c++”, 300, 285};
        clrscr ( );
        printf (“ \n Book name : %s “ , bk1.book);
        printf (“\n No of pages  :%d “,bk1.pages);
        printf (“\n book price     :%f “,bk1.price);
}
```

Out put:

Book name : C ++

No of pages : 300

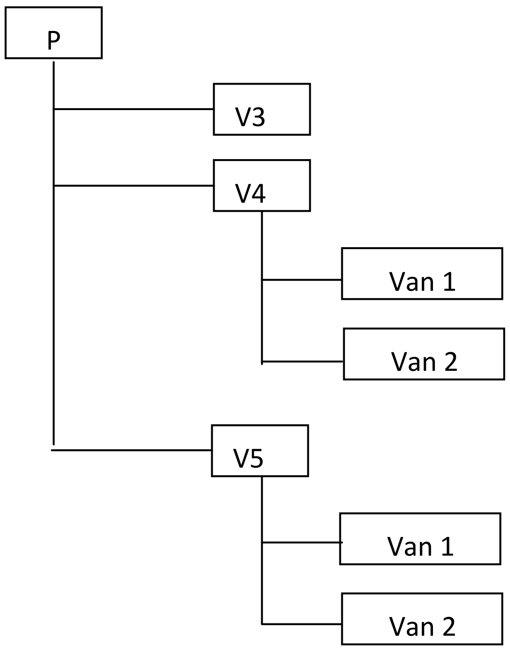
Book Price : 285

Structure within structure (nested):-

We can take any data type for declaring structure members like int, float, char etc. In the same way wee can also take object of one structure as member in another structure.

The syntax as follows:-

```
struct s1

{

        Type var1;

        Type var2;

};

struct s2

{

        Type v3;

        Strict s1       v4;

        Strict s2       v5;

};

struct  s2      p;
```



=> WAP to enter the records of no of students & then display them using nested structure

```
# include <stdio.n> # define max 300
Struct data
{
        int  day;
        int  month;
        int  year;
};
Struct student
{
        char name [30];
        long int rollno;
        struct data dob;
};
struct  student bio [max];
void main ()
{
        int  i, n;
        printf (“ How many student data required:”);
        scanf (“%d”,&n);
        for(i=0;i<=n-1; i++)
        {
                printf (“Record. No :  % d\n”, i+1);
                printf (“Student name :”);  sf(“ %s”,bio[i].name);
                printf (“Roll no :”) ; sf (“ %d “, bio [i].rollno);
                pf (“date of birth”); pf (“\n Day”);
                printf (“%d”, &bio [i].dob.day);
                printf (“ month:”); sf (“%d”, &bio[i].dob.month)
                printf (“year:”) ; sf (“% d”, & bio[i].dob.year).
         }
printf (“ The entered contents are :”);
for (i=0;i <=n-1;i++)
{       printf (“\n Record No : % d”, i+1).
        printf (“\n Student name : % s”, bio [i].name);
        printf (“\n Roll no : % d “, bio [i] . rollno);
        printf (“\n Date of birth “);
        printf (“\n day : % d”,bio[i].dob.day);
        printf (“\n month : % d”, bio [i].dob. month);
        printf (“\n year : % d”, bio [i].dob.year);
  }
}
```

Array of structures:-

A structure is simple to define if there are only one or two element, but incase there are too many objects needed for a structure, for ex. A structure designed to show the data of each student of the class, then in that case, the away will be introduced. A structure declaration using array to define object is given below.

```
        Stuct student
        {
                Char name [15];
                Int rollno;
                Char sex;
        };
        Student data[50];
```

Initializing array of structures:-

It is to be noted that only static or external variable can be initialized.

Ex: -

```
struct employee
```

```
                {
                        int enum;
                        Char sex;
                        Int sal;
                };
Employee data [3] = {   {  146, „m‟,  1000 },
                          {  200, „f‟,    5000},
                          {  250,  „m‟,10000}   };
```

If incase, the structure of object or any element of struct are not initialized, the compiler will automatically assigned zero to the fields of that particular record.

Ex: employee data [3] = { { 146,‟m‟} ,{ 200, „f‟ },{250 ,‟m‟}};

Compiler will assign:-

Data [0] sal=0; data [1].sal=0; data [2]. Sal=0;

=>WAP to initialize the array of structure members & display

```
void main ()
{
        int i;
        Struct student
        {
                long int rollno;
                char sex;
                float height;
                float weight;
        };
   struct student data [3] = {  {121,‟m‟,5.7,59.8},
                                {122,‟f‟,6.0,65.2},
                                {123,‟m‟, 6.0, 7.5} };
   clrscr ();
   printf ("the initialized contents are:");
   For ( i=0; i< =2; i++)
  {
        printf ("%d/n   ** Record is  \n ", data [i].rollno);
        printf ("% c\n ",  data [i] .sex);
        printf ("%f\n",  data [i].height);
        printf ("%f\n",  data [i]. weight);
   }
}
```

Arrays with in the structures:-

Arrays can be used within the structure along with the member declaration. In other words, a member of a structure can be an array type also.

```
struct emp
{       char name [30] ;// 30 characters array .
        char dept [30];
        int sal;
        char add[50];
};
```

Structures &pointers:-

So far we have seen that pointer is a variable that hold the memory address of other var. of basic data type such as integer, float, char etc. A pointer can also be used in a program to hold the address of heterogeneous type of var .i.e. structure var. pointer along with structure are used to make linked lists, binary trees etc.

Ex:-

```
         strtuct data
          {
                char n[10]
              ---
              ---
          };
        struct data * ptr;
```

*ptr is a ptr var. Which holds the address of the structure named data? The ptr is declared same as any object of a struct. Now, this ptr struct var can be accessed & processed by one of the following given two methods.

1) Any field of the structure can be accessed as:

(*ptr name). Field name =var;

Ex:-

```
        void main ()
         {
                struct data
                        {   int a ; float b;  char c;  };
                struct data * ptr;
                (*ptr) . a =100;
                (*ptr) . b =20.9;
                (*ptr).  C ="m";
         }
```

The pointer to struct is expressed using a dashed line followed by (>) sign**.**

ptr name – >field name = var;

=> wap to use ptr s in structures using 2 methods

```
 void main ()
     {
                struct data
                        { int a; float b; char c ; };
                struct data * ptr;
                ptr ──►a  = 100;
                ptr ──►b  = 20.9;
                ptr ──►c =  „m" ;
                -----
                -----
      };
```

Uses of structures:-

Structures are useful in database management i.e. to main to data about employees, but use of structures much beyond data base management**.** They can be used for.

1. Changing the size of cursor.
2. Clearing the eonents of the screen.
3. Drawing any graphical shape on the screen.
4. Placing the cursor at appropriate position on the screen.
5. Receiving a key from the keyboard.
6. Checking the memory size of the computer finding out the list of equipment attached to the computer.
7. Formatting a floppy.
8. Hiding a file from directly.
9. Display the directory of a disk.
10. Sending the o/p to printer.
11. Interacting with the mouse.

Terms within structures:-

ptr : can be used as a member of the structure.

ex :

```
 void main ()
     {
               struct  student
                         {
                                 int rno, *ptr;
                          };
                          struct  student s;
                          s.ptr = * s.rno;
                          *s.ptr = 10;
                          printf (”rno = % d/n”, s.rno);
     }
```

Structures & functionctions:-

Structures may be passed as an argument to a functionction in diff ways. We can pass individual members, who structures or struct ptr. To the function, similarly, a function. Can return either a struct. Member or a whole structure van or a ptr. To structure.

1). Passing struct members as arguments.

2). Passing struct var. as argument.

3). Passing ptr. To struct as argument.

4). Returning a struct var. from function.

5). Returning a ptr. To struct from function.

6). Passing away of struct as argument.

Note :-

1). All the change made in the away of struct. Inside the called function. Will be visible in the calling function .

2). If the size of a struct is very large then it is not efficient to pass the whole struct. To function.

3) It is better to sent the address of the structure which will improve the execution speed.

4) Passing individual members to a function. Become cumbersome when there are many members & the relationship b/w the members is also last so we can pass the whole structure as an argument

5) As any other van. Structure .van is return the retuned value can be assigned to a structure of the appropriate time.

6) As we pass away to a function away of Structure. Can also be passed to a function where each ele. of away is of Structure type.

Union:-

Union is a data type in „C" which allows an **ruelay** of more than none Var in the same memory **arc** i.e. using the memory space for different van at same place.

Syntax:-

```
                    union     uni-name

                              {

                                        type     Var 1;

                                        type     Var 2;

                               }< union Var> ;
```

All var inside an union share storage space. The compiler will allocate sufficient storage for the union var to accommodate the largest element in the union .Other element of the union use the same space (as it is using same space) it is different from structures.

Declining var & pointer to unions:-

```
union name

      {

           char   name [40];

           char   id [20];

       }    id      *ptr 2;
```

Difference b/w structures & unions:

```
structure

    {

         char     name [25]

         Int      idno;

         float    sal;

     } emp;

union

     {

         char     name [25]
```

```
        Int     idno;
        float   sal;
     } desc;
void main()
    {
                printf (“ size of struct is %d” size of ( emp));
                printf (“size of union is %d”, size of (desc));
     }
```

<u>Operations on union members</u>:

Only one member of a union can be accessed at a time i.e. because at every instance only one of the union var. will have a meaningful value, only that var is read.

<u>Operations an unions:-</u>

Unions have all the features provided y a Structure which are a conveyance of memory sharing properties of a union. This is evident by the following operations union which are legal.

1. Union var. can be assigned to another union var
2. Union Var can be passed to a function as parameters.
3. The address of the union var. can be extracted by using address of operator.
4. A function can accept & retune a union of a ptr to a union

<u>Scope of a Union</u>:

```
void main ( )
    {
                union
                        {
                                int i;
                                char c;
                                float f;
                        };
        printf { " i = % d", i);
}
```

O/p:-

/* compiler dependent* /

Fields in structure:

To allocate memory in bits. Bit fields are treated as unsigned integers

Syntax:-

```
        structure <tag>
                {
                        unsigned <var> : value ;/*rep of bit fields */
                };
```

Ex:

```
        structure with_bits
              {
                        unsigned  first :5;
                        unsigned second :9;
              };
```

Total 14, but it goes to 16 as we can‟t have 14 bit memory

```
void main ( )
   {
                Union
                {
```

```
                int i ;
        };
        i = 0;/*second =0 */
        b. first =9; /* second =0*/
}
```

Strictest & union assignment question:

1. Difference between an array & structure.
2. Write the syntax of a structure & explain in with an example.
3. WAP to instate nested structures.
4. Difference between the two member access operators: &,→.
5. Difference between a structure & union.
6. What is the scope of the member of a union.

Programs:

1. WAP to pass structure elements to function print ( ) & print the elements.
2. WAP to declare pointer to structure & display the contents of the structure.
3. WAP to find size of union in C? How data is stored using union ?.

Objective Questions:

1. Members of a structure can be accessed using
   a. dot (.) operator.
   b. → operator.
   c. both
   d. none.

2. Total amount of memory required to store a structure variable is:
   a. Sum of size of all the member .
   b. Same as that required by the largest member.
   c. Sum of size of all the members + the paddira bytes that may be provided by the compiler
   d. None.
3. Total amount of memory required to store a union viable is:
   a. Sum of size of all the member
   b. Same as that required by the largest member
   c. Sum of size of all the members + the paddira bytes that may be provided by the compiler
   d. None.

4. In union, addresses of the voiles is
   a. different
   b. same
   c. depends on the compiler
   d. cannot say

5. syntax of a union is similar to
   a. file
   b. away
   c. structure
   d. none

<u>FILES</u>

A file is a place on disk where group of related data are stored. C supports a number of functions that have the ability to perform basic file operations, which include:

- Naming a file
- Opening the file
- Reading data from the file
- Writing data to the file
- Closing the file

There are two distinct ways to perform file operations in C. The first one is known as the low level I/O and uses UNIX system calls. The second method is referred to as the high level I/O operation and uses functions in C's standard I/O library.

FILE TYPES:

DOS treats files in two different ways viz., as binary or text files. Almost all UNIX Systems do not make any distinction between the two. If a file is specified as the binary type, the file I/O functions do not interpret the contents of the file when they read from or write to the· file. But, if the file is specified as the text type the I/O functions interpret the contents of the file. The basic differences in these two types of files are:

I. ASCII 0xla is considered an end-of-file character by the file I/O function when reading from a text file and they assume that the end of the file has been reached.
II. In case of DOS a new file is stored as the sequence 0*0d 0*0a on the disk in case of text files. UNIX stores \n as 0* 0a both on disk and in memory.

FILE Operations:

1. Opening a File:

C communicates with files using a new data type called a file pointer. This type is defined within stdio. h, and written as FILE *. A file pointer called output_file is declared in a statement like e

FILE *output_file;

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include the filename, data structure, purpose.

The general format of the function used for opening a file is
FILE *fp;

fp = fopen("filename", "mode");

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening the file. The mode does this job.

r open the file for read only.
w open the file for writing only.
a open the file for appending data to it.
r+ open an existing file for update (reading and writing)
w+ create a new file for update. If it already exists, it will be overwritten.
a+ open for append; open for update at the end of the file

Consider the following statements:

FILE *pl, *p2;
pI =fopen("data" ,"r");
p2=fopen("results", "w");

In these statements the pI and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If the data file does not exist error will occur.

2. Closing the File:

The input output library supports the function to close a file; it is in the following format.
f close(file -'pointer);

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer.
Observe the following program ..

```
FILE *p 1 *p2;
Pl = fopen ("Input","w");
p2 = fopen ("Output" ,"r");
……
……
fclose(p1); fclose(p2)
```

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

Reading a character from a file and writing a character into a file (getc () and putc ()' **functions)**

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fpl); similarly getc function is used to read a character from a file that has been open in read mode. c=getc(fp2).

The program shown below displays use of a file operations. The data enter through the keyboard and the program- writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
Sample program for using getc and putc functions*/.:
#include< stdio.h >
void mainO
{

        FILE *fl;
        printf("Data input output");
        fl =fopen("Input","w"); /*Open the file Input*/
        while«c=getcharO)!=EOF) /*get a character from key board*/
        putc( c,fl); /*write a character to input* /
        fclose(fl); /*close the file input* / printf("Data output");
        fl =fopen("INPUT" ,"r"); /*Reopen the file input* /
        while« c=getc(fl ))!=EOF)
        printf("%c",c );
        fclose(fl );
}
```

4. Reading a integer from a file and writing a integer into a file (getw() and putw() functions)

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of getw and putw are:

```
putw(integer,fp );
getw(fp);
```

```
/*Example program for using getw and putw functions * / #include< stdio.h >
void main( )
{
FILE *fl, *f2, *f3;
int number,i;
printf("Contents of the data filenn");
fl =fopen("DA T A","W");
for(i=l;i< 30;i++)
{
printf("enter the number");
scanf("%d" ,&number);
if(number= =-l)
break;
putw (number,fl );
}
fclose(fl);
fl =fopen("DA T A","r");
f2=fopen("ODD", "w");
f3 =fopen ("EVEN", "w");
while((number=getw(fl))!=EOF)/* Read from data file*/
{ if(number%2=0)
putw(number,f3);/*Write to even file*/
else
putw(number,f2);/*write to odd file*/
}
fclose(fl  );
fclose(f2);
fclose(f3);

f2=fopen("O DD" ,"r"); f3
=fopen("EVEN", "r");
printf("nnContents of the odd filenn");
while(number=getw(f2))! = EO F)
printf("%d" ,number);
printf("nnContents of the even file");
while(number=getw(f3))! = EPF)
printf("%d" ,number);
fclose(f2);
fclose(f3);
}
```

5. fscanf() and fprintf() functions:

The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of theses functions is a file pointer which specifies the file to be used. The general form of fprintf is

fprintf(fp,"control string", list);

Where fp id a file pointer associated with a file that has been opened for writing. The control string is fiJe output specifications list may include variable, constant and string.

fprintf(fl ,%s%d%f",name,age,7 .5);

Here name is an array variable of type char and age is an int variable
The general format of fscanf is

fscanf(fp,"controlstring" ,list);

This statement would cause the reading of items in the control string.

Example:
fscanf(fL,"5 s%d" ,item, & quantity");

Like scanf, fscanf also returns the number of items that are successfully read.

```
I*Program to handle mixed data types*/ #include< stdio.h >
main()
{
        FILE *fp;
        int num,qty,i;
        float price,value;
        char item[10],
        flemupe[10];
        printf("enter the filename");
        scanf("%s" ,filename);
        fp=fopen( filename, "w");
        printf("Input inventory data");
        printf("Item namem number price quantity");
        for (i=l;i< =3;i++)
        {
                fscanf( stdin, "%s%d %f%d" ,item,&number ,&price,&quality);
                fprintf( fp, "%s%d%fl,lod" ,itemnumber, price, quality );
        }
        fclose (fp);
        fprintf( stdout, "nn");
        fp=fopen( filename, "r");
        printf ("Item name number price quantity value");
        for(i=l;i< =3;i++)
        {
        fscanf( fp, "%s%d %f1l,lod" ,item, &number ,&prince, &quality);
        value=price*quantity;
        fprintf("stdout, "%s%d%f1l,lod %dn" ,item, number ,price, quantity , val ue);
    } fclose(fp );
}
```

6. fgetc () and fputc() functions:

fgetc() function is similar to getc() function. It also reads a character and increases the file pointer position. If any error or end of the file is reached it returned EOF.

fputc() function writes the character to the file shown by the file pointer. It also increases the file pointer position.

Sample program to write text to a file using fputc() and read the text from the same file using fgetc()

```
#include<stdio.h>
void main()
{
        FILE *fp; char c;
        fp=fopen("lines. txt", "w");
        while( ( c=getcharO 1=' *.') fputc( c,fp);
        fclose(fp);
        fp=fopen("lines. txt" ,"r");
        while(( c=fgetc(fp ))!=EOF) printf("%c",c );
        fclose(fp);
}
```

7. fgets() and fputs() functions:

fgets() function reads string from a file pointed by file pointer. It also copies the string to a memory location referred by an array.
fputs() function is useful when we want to write a string into the opened file .
.·

Sample program to write string to a file using fputs() and read the string from the same file using fgets()

```
#include<stdio .h> void main()
{
        FILE *fp;
        Char file [ 12],text[ 50];
        fp = fopen("line. txt" ,"w");
        printf("enter text here ");
        scanf("%s" , te~t); fputs(text,fp );
        fclose( fp );
        fp = fopen("line. txt", "r");
        if(fgets(text,50,fp )!=NULL)
        while(text[i] !='\O')
        {
                putchar(text[i]);
                 i++;
        }
        fclose(fp );
        getch();
}
```

8. fread() and fwrite () functions

You can regard an array, or an array of structured variables, as simply a block of memory of a particular size. The input/output routines provide you with routines that you can call to drop a chunk of memory onto a disk file with a single function call. However, there is one thing you must remember. If you want to store blocks of memory in this way the file must be opened as a binary file. What you are doing is putting a piece of the program memory out on the disk. You do not ~ow how this memory is organised, and will never want to look at this, so you must open the file as a binary file. If you want to print or read text you use the **fprintf or scanf** functions.

The function **fwrite** sends a block of memory out to the specified file. To do this it needs to know three things; where the memory starts, how much memory to send, and the file. to send the memory to. The location ofthe memory is just a simple pointer, the destination is a pointer to a **FILE** which has been opened previously, the amount of memory to send is given in two parts, the size of each chunk, and the number of chunks. This might seem rather long wjnded, but is actually rather sensible. Consider:

```
typedef struct {
char name [30] ;
char address [60] ;
int account ;
int balance ;
int overdraft ;
} customer;
```

```
customer Hull_Branch [100] ;
FILE * Hull_File;

fwrite ( Hull_Branch, sizeof ( customer), 1 00, Hull_File) ;
```

The first parameter to **fwrite** is the pointer to the base of the array of our customers. The second parameter is the size of each block to save. We can use **sizeof** to find out the size of the structure. The third parameter is the number of customer records to store, in this case 100 and the -final parameter is the file which we have opened. Note that if we change the number of items in the customer structure this code will still work, because the amount of data saved changes as well.

The opposite **o**f **fwrite is fread.** This works in exactly the same way, but fetches data from the file and puts it into memory:

```
fread ( Hull_Branch, sizeof ( customer), 1 00, Hull_Data) ;
```

If you want to check that things worked, these functions return the number of items that they transferred successfully:

```
if( fread (Hull_Branch, sizeof( customer), 100, Hull_Data) < 100) {
printf ( "Not all data has been read!\n\n" ) ; }
}
```

Sample program using fread() and fwrite() functions

```
#incl ude<stdio.h>
struct student
{
char name[20];
int age;
 }stud[5];
void main()
 {
        FILE *fp;
        int i,j=0,n;
        char str[15];
        fp=fopen("line. txt" ,"rb");
        printf("enter the number of records");
        scanf("%d",n);
        for(i=O;i<n;i++ )
        {
                scanf("%s",stud[i] .name);
                scanf("%d",stud[i] .age);
        }
}
```

```
whileG<n)
{
fwrite( &stud,size( stud), 1 ,fp);
j++;
}
}
fclose(fp );
fp=fopen ("line. txt", "r");
j=0;
while(j<n)
{
fread( &stud,sizeof( stud), 1 ,fp);
printf("%s %d" ,stud[j] .name,stud[j] .age);
} fclose(fp ); }
```

9. Random access files:

The file I/O routines all work in the same way; unless the user takes explicit steps to change the file position indicator, files will be read and written sequentially. A read followed by a write followed by a read (if the file was opened in a mode to permit that) will cause the second read to start immediately following the end of the data just written. (Remember that stdio insists on the user inserting a buffer-flushing operation between each element of a read-write-read cycle.) To control this, the Random Access functions allow control over the implied read/write position in the file. The file position indicator is moved without the need for a read or a write, and indicates the byte to be the subject of the next operation on the file.

Three types of function exist which allow the file position indicator to be. examined or changed. Their declarations and description follow.

```
#include <stdio.h>

/* return file position indicator * / .
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);

/* set file position indicator to zero * / void rewind (FILE *stream);
/* set file position indicator */
int fseek(FILE *stream, long offset,
int ptmame); int fsetpos(FILE *stream, const fpos_t *pos);
```

ftell returns the current value (measured in characters) of the file position indicator if stream refers to a binary file. For a text file, a 'magic' number is returned, which may only be used on a subsequent call to fseek to reposition to the current file position indicator. On failure, -1 L is returned and ermo is set.

rewind sets the current file position indicator to the start of the file indicated by stream. The file's error indicator is reset by a call of rewind. No value is returned.

Fseek allows the file position indicator for stream to be set to an arbitrary value (for binary files), or for text files, only to a position obtained from ftell, as follows:

- In the general case the file position indicator is set to offset bytes (characters) from a point in the file determined by the value of ptrname. Offset may be negative. The values of ptmame may be SEEK_SET, which sets the file position indicator relative to the beginning of the file, SEEK_CUR, which sets the file position indicator relative to its current value, and SEEK_END, which sets the file position indicator relative to the end of the file. The latter is not necessarily guaranteed to work properly on binary streams.
- For text files, offset must either be zero or a value returned from a previous call to ftell for the same stream, and the value ofptrname must be. SEEK_SET.
- Fseek clears the end of file indicator for the given stream and erases the memory of any ungetc. It works for both input and output.
- Zero is returned for success, non-zero for a forbidden request.

Note that.for ftell and fseek it must be possible to encode the value of the file position indicator into a long. This may not work for very long files, so the Standard introduces fgetpos and fsetpos which have been specified in a way that removes the problem. fgetpos stores the current file position indicator for stream in the object pointed to by pos. The value stored is 'magic' and only used to return to the specified position for the same stream using fsetpos. fsetpos works as described above, also clearing the stream's end-of-file indicator and forgetting the effects of any ungetc operations. For both functions, on suc,cess, zero is returned; on failure, non-zero is returned and errno is set.

```
[#include <stdio.h>
;
int main()
{
FILE * f;
f= fopen("myfile.txt", "w"); fputs("Hello World", f); fseek(f, 6,
SEEK_SET); fputs(" India", f);
fclose(f);
return 0;
}
```

Now the file consist of following data:

Hello India
Sample program to print the current position of the file pointer in the file using ftell() function:

```
        #include<stdio.h>    .
void main()
 {
FILE *fp;
char ch; fp=fopenG'lines.txt" ,"r"); fseek( fp,21 ,SEEK_SET); ch=fgetc(fp );
clrscr();
while(!feof(fp ))
{
        printf("%c" ,ch); printf("%d",ftell(fp )); ch=fgetc(fp );
}
rewind(fp ); while(! feof( fp))
{
        printf("%c" ,ch); printf("%d" ,ftell(fp)); ch=fgetc(fp );
}
fclose(fp );
}
```

10 Error handling
It is possible that an en-or may occur during I/O operations on a file. Typical en-or situations include:

1. Trying to read beyond the end of file mark.
2. Device overflow
3. Trying to use a file that has not been opened.
4. Opening a file with an invalid file name.
5. Attempting to write to a write protected file.
6. Trying to perform an operation on a file when the file is opened for another type of operation. he standard I/O functions maintain two indicators with each open stream to show the end-of file and en-or status of the stream. These can be interrogated and set by the following functions: .

.

```
#include <stdio.h>

void clearerr(FILE *stream);

int feof(FILE *stream); int

ferror(FILE *stream); void

perror(const char *s);
```

clearer clears the error and EOF indicators for the stream.
feof: returns non-zero if the stream's EOF indicator is set, zero otherwise.
Ferror: returns non-zero if the stream's error indicator is set, zero otherwise.
Perror: prints a single-line error message on the program's standard output, prefixed by the string pointed to by s, with a colon and a space appended. The error message is determined by the value of ermo and is intended to give some explanation of the condition causing the error.

For example, this program produces the error message shown:

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

main()
{

        fclose(stdout);
        if(fgetc(stdout) >= 0)
        {
                fprintf(stderr, "What - no error!\n");
                exit(EXIT _FAILURE);
}
ferror("fgetc");
exit(EXIT _SUCCESS);
}
/* Result */
fgetc: Bad file number
```

Assignment questions and programs:

1. What is the difference between end of a file and end of a string?
2. Distinguish between text mode and binary mode operation of a file?
3. What is the use of fseekO? Explain its syntax?
4. How does an append mode differs from a write mode?
5. Compare between printf and fprintf functions?
6. What is the significance of EOF?
7. What are the common uses of rewind and ftell functions?
8. Describe the use and limitations of the functions getc and putc?
9. What is the difference between the statements rewind(fp) and fseek(fp,OL,O)?
10. Write a program to reposition the file to its lOth character?
11. Write a program to find the size of the file?
12. Write a program to write contents of one file in reverse into another file?
13. Write a program to read to interchange the contents of two files?
14. Write a program to combine contents of two files in a third file?
15. Write a program to read the contents of three files and find the largest file?
16. Write a program to show how the rewind function works?
17. Write a program to read the last few characters of the file using fseek function?

Bits:

1. Which one of the following function is used for sorting information in a disk file [d]
a) Scanf() b) fscanf () c)printf () d)fprintf()

2. The EOF is equivalent to [c]
a) 2 b) 1 c) -1 d) 0

3. Which of the following function is used to set the record pointer at the beginning of the file
[c]
a) Putc() b) getw() c) rewind() d) gets()

4. Which of the following mode opeqs a binary file in write mode [d]
a) W+ b)Y wb+ c) w d) wb

5. Which of the following functions used to write an entire block of data to a file [d]
a) Fputs b). fgets c) fscanf d) fwrite

6. The fscanf() function reads data from [a]
a) File b) keyboard c) both a and b d) none

7. When fopen() fails to open the file it returns [a]
a) Null b) -1 c) 1 d) none

8. The rewind function takes number of arguments [a]
a) 1 b)2 c)3 d)4

9. fseek(fptr,0,0) is equivalent to [b]
a) ftell b) rewind c) fput d) none

9. feof function checks for [c]
a) fiJe opening error b) data error c) end of file d) file closing error

10. The contents of the file are safe if it is opened in _ mode [a]
a) a b) r c) a+b d) none

11. The general format of feof function [d]
a) Feof(fp," control string") b)feof(fp) c) feof(fp,list) d) none

12. Which of the following is used to store data permanently [d]
a) Arrays b) structures c) pointers d) files

13. The file opened in w+ mode can be [a]
a) Read/write b) only read c) only write d) none

14. The fseek function needs _ arguments [b]
a) 2 b) 3 c) 1 d) none

15. Which of the standard file pointer is used for standard auxillary device [b]
a) Stdpm b)stdaux c) stdio d) stdconio

16. Which of the following function is used to read each member of the structure in the file[c]
a) Gets b) puts c) fscanf d) fprintf

17. The contents of the file is lost if it is opened in __ mode [b]
a) a b)w c)w+ d)a+

18. The function that is not used for random access to file [d]
a) Ftell b) rewind c) fseek d)fprintf

19. The C library that contains the prototype statement for the file operation [b]
a) Proto.h b) file.h c) stdio.h d) stdlib.h

Write a program to read last few characters of the file using fseek () Statement.

```
# include < stdio.h>
# include < conio.h>
Void main ()
{
        File * fp;
        Int n, ch;
        Clrscr ();
        Fp=fopen (“text.txt”, “r”),
        Printf (“/n contents of file \n”);
        while ((ch=fgetc(fp))!=EOF)
        Printf (“%c”, ch);
        Printf (“/n How many characters including spaces would you like to skip?:”);
        while ((ch = fgetc(fg))
                printf (“%c”ch);
        fclose(fp);
}
```

```
o/p:   How many characters including spaces would you like to skip? : 5
Lat characters of the file
WORLD
```