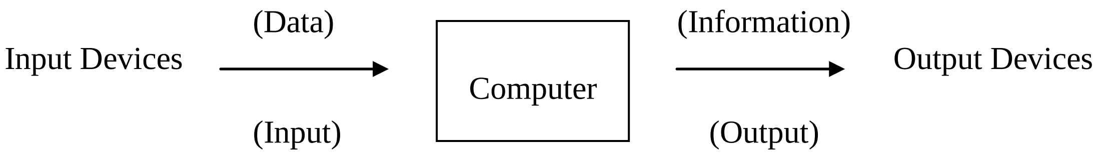
Complete C & Data Structure Notes

UNIT-I: Overview of Computers and Programming – Electronic computers then and now, Computer hardware, Computer Software, Algorithm, Flowcharts, Software Development Method, Applying the software development method.

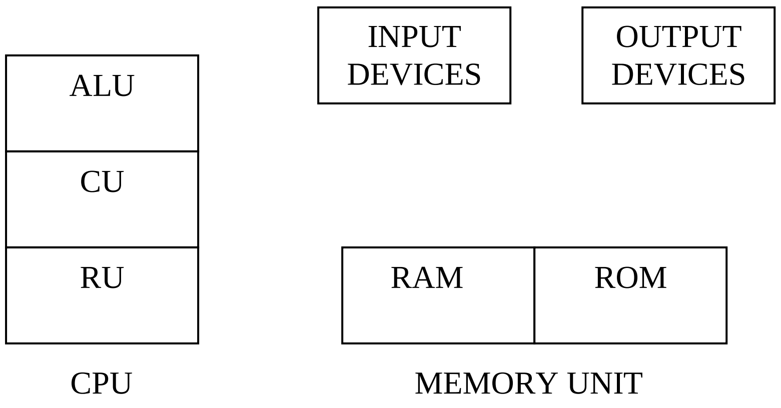
COMPUTER Computer is an electronic device that takes data as input from input devices, process the instructions, and produces information as output on output devices.



FUNCTIONAL PARTS OF THE COMPUTER

The main functional parts of the computer are:

1. Input Devices
2. Output Devices
3. Central Processing Unit
4. Memory Unit



Input Devices:

Input devices are used to submit data to the computer for processing the instructions.

Examples: Keyboard, Mouse, Touch pad, Scanner etc.,

Output Devices:

Output devices are used to display information after processing the instructions by the computer.

Examples: Monitor, Speaker, Printer etc.,

Central Processing Unit:

Central processing unit (CPU) is main heart of the computer. Since, entire processing instructions are carried out by the CPU. CPU contains three important parts. Those are

Arithmetic and Logical Unit (ALU)
Control Unit (CU)
Register Unit (RU)

Entire arithmetic and logical calculations are performed inside the arithmetic and logical unit. Control unit is responsible for to follow up all the signals carried out by the computer. CPU's current instructions and data values are stored temporarily inside a high-speed memory location called register unit.

Memory Unit:

Memory unit is used to store the data. Memory unit contains an ordered sequence of storage locations called memory cells and each memory cell has a unique address that indicates relative position in memory.

Address	Contents
0	-27.2
1	543
2	X
.	
.	
.	
.	
99	75.62

The data stored in a memory cell are called the contents of the cell. A memory cell is actually a grouping of smaller units called **bytes.** Each byte is formed with the combinations of **8 bits.** Each bit is a binary digit either **0** or **1.**

Main memory: Main memory stores programs, data and results. Most common types of main memory (**primary memory)** are: RAM (Random Access Memory) and ROM (Read-only-Memory).

RAM offers temporary storage of programs and data. It allows both read and write operations. RAM is a **volatile** memory. Since, every thing in RAM will be lost when the computer is switched off.

ROM stores programs or data permanently. It allows only read operation. ROM is a **non-volatile** memory. Since, the data stored there do not disappear when the computer is switched off.

RAM is very expensive in cost and has limited storage capacity. So those large amounts of programs are inefficient to store in RAM. For storing huge amount of data, it is better to select secondary storages devices.

Secondary storage devices are less expensive in cost and have large storage capacity. Information stored in secondary storage devices are organized in terms of **files**.

Examples for secondary storage devices are:

Hard disks
Floppy disks
Zip disks
Compact disks (CD)
Digital video disks (DVD) etc.,

Primary storage devices and secondary storage devices are available in different storage capacities like Bytes, Kilobytes, Megabytes, Gigabytes and Terabytes.

TERM	ABBREVIATION	EQUIVALENT TO
Byte	B	8 bits
Kilobyte	KB	1,024 (2^{10}) bytes
Megabyte	MB	1,048,576 (2^{20}) bytes
Gigabyte	GB	1,073,741,824 (2^{30}) bytes
Terabyte	TB	1,099,511,627,776 (2^{40}) bytes

The program must first be transferred from secondary storage devices to main memory before it can be executed. Programmer submits input data from input devices to process the instructions. Those values are stored in the computer's main memory, where they can be accessed and manipulated by the central processing unit. The result of this manipulation are then stored back in main memory. Finally, the information in main memory can be displayed through an output device.

COMPONENTS OF COMPUTER

Elements or components of a computer system fall into two major categories: Hardware and Software.

COMPUTER HARDWARE

Hardware is physical parts of the computers. The parts are possible to touch and visible.

Examples: Keyboard, Mouse, Monitor, Speakers, RAM etch.,

COMPUTER SOFTWARE

Software is a collection of programs. A program contains set of instructions to initiate the computer to perform some action. Main components of computer software are: Operating system and Application software.

Operating System:

The collection of computer programs that control the interaction of the user and the computer hardware is called the **operating system (OS).**

Examples: DOS, Windows, Unix etc.,

Main responsibilities of the operating system are:

- Communicating with the computer user
- Managing allocation of memory, processor time, and other resources
- Collecting input from input devices
- Conveying the output on output devices
- Accessing data from secondary storage devices
- Writing data to secondary storage devices.

Application Software:

The collection of programs used to solve the given problem statement is called the **application software.** Programs are designed based on the computer languages.

Examples: Pascal, Fortran, Cobol, C, C++, Java etc.,

Generally computer languages are classified into three types as:

- Machine languages
- Assembly languages
- High-level languages

Machine languages *are formed with the combination of* ***machine codes*** *which are binary numbers either* ***0*** *or* ***1****.*

Assembly languages *are formed with the combination of* ***mnemonic codes,*** *which contains simple English words like ADD, SUB, MUL etc.,*

High-level languages *are formed with the combination of simple* ***English sentences****.*

Most of the users are interested to design programming language in high-level language. But a computer can understood only binary language which contains binary number either 0 or 1. So, that a mediator is required to convert the given programming language into machine codes and vice-versa. Such mediators are translators.

TRANSLATORS

A translator is a program that converts the given programming language from one type to machine codes and vice-verse. Different types of translators are:

- Assembler
- Compiler
- Interpreter

Assembler is a translator used to convert the assembl7y language program into machine code and vice versa.

Compiler and Interpreter are translators used to convert the high-level language program into machine code and vice versa. The main different between both of them is compilers converts the entire program into machine code and displays error if occurred. Where as interpreter converts line by line of the program into machine code and displays errors immediately, if occurred.

TYPES OF COMPUTERS

At the initial stage of computer invention, **vacuum tubes** are the basic electronic component used for processing the instructions. In modern technology, the entire circuitry of a computer processor can be package in a single electronic component called a **microprocessor** chip. According to the size and performance, modern computers are classified into different types as,

Personal Computers	:	Used by a single person
Mainframes	:	Used in ATMs, Banking networks, Airlines etc.,
Super Computers	:	Used in Research laboratories, Weather forecasting etc.,

COMPUTER NETWORKS

Network is collection of systems that are communicated with each other. Networks are classified into two types as:

- Local Area Network (LAN)
- Wide Area Network (WAN)

LAN is collection of computers that linked with each other. It is limited to a small area like with in the organization, building etc., In LAN technology systems can share information and resources such as printers, scanners etc.,

A network that links many individual computers and local area networks over a large geographic area is called a **WAN.**

Ex: Internet, Corporate companies etc.,

***

ALGORITHM

An algorithm is a step-by-step procedure of solving the given problem statement. Algorithms are designed by using *pseudo code.* Pseudo code is a language independent code. All algorithms must satisfy the following characteristics.

- Input : Zero or more quantities are externally supplied
- Output : At least one quantity is produced
- Definiteness : Each instruction is in clear format
- Finiteness : If we trace out the instructions, the algorithm must terminates after a finite sequence of steps
- Effectiveness : Every instruction must be in basic format.

Basic rules followed while designing algorithms are:

- START and STOP statements are used for beginning and ending of the algorithm
- READ and WRITE statements are used for input and output statements
- ← Symbol is used to assign values to the variables.

DESIGN ALGORITHMS FOR THE FOLLOWING PROBLEM STATEMENTS

1. Addition of given two numbers.
2. Addition, Subtraction, Multiplication and Division of given two numbers.
3. Average of given three numbers.
4. Swapping of given two numbers.
5. Calculate simple interest (SI=(PTR)/100).
6. Gross Salary of an Employee Where HRA = 1500 and DA = 75% of Basic Pay (GS=BPARY+HRA+DA).
7. Conversion of Fahrenheit Temperature into Celsius Temperature (C=(F-32)/1.8).
8. Calculate Area and Circumference of a Circle (Area=πr^2 and Circumference=2∏r).

1. Addition of given two numbers

```
Step 1: START
Step 2: READ x, y
Step 3: sum ← x + y
Step 4: WRITE sum
Step 5: STOP
```

2. Addition, Subtraction, Multiplication and Division of given two numbers

```
Step 1: START
Step 2: READ x, y
Step 3:  Sum ← x + y
         Sub ← x – y
         Mul ← x * y
         Div ← x / y
Step 4: WRITE Sum, Sub, Mul, Div
Step 5: STOP
```

3. Average of given three numbers

```
Step 1: START
Step 2: READ x, y, z
Step 3: sum ← x + y + z
Step 4: avg ← sum / 3
Step 5: WRITE avg
Step 6: STOP
```

4. Swapping of given two numbers

```
Step 1: START
Step 2: READ x, y
Step 3:  Temp ← x
         x ← y
         y ← Temp
Step 4: WRITE x, y
Step 5: STOP
```

5. Calculate simple interest (SI=(PTR)/100)

```
Step 1: START
Step 2: READ P, T, R
Step 3:          SI ← (P*T*R)/100
Step 4: WRITE SI
Step 5: STOP
```

6. Gross Salary of an Employee Where HRA = 1500 and DA = 75% of Basic Pay (GS=BPARY+HRA+DA)

```
Step 1: START
Step 2: READ BPAY
Step 3:          HRA ← 1500
         DA ← (BPAY*75)/100
Step 4: GS ← BPAY + HRA + DA
Step 4: WRITE GS
Step 5: STOP
```

7. Conversion of Fahrenheit Temperature into Celsius Temperature (C=(F-32)/1.8)

```
Step 1: START
Step 2: READ F
Step 3: C ← (F-32)/1.8
Step 4:          WRITE C
Step 5: STOP
```

8. Calculate Area and Circumference of a Circle (Area=πr^2 and Circumference=2∏r)

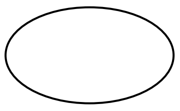
```
Step 1: START
Step 2: READ r
Step 3: Area ← ∏ * r * r
        Circumference ← 2 * ∏ * r
Step 4: WRITE Area, Circumference
Step 5: STOP
```

FLOWCHART

Pictorial or Graphical representation of an algorithm is called a flowchart. Flowcharts are designed by using some specific symbols. The most import symbols used for designing flowcharts are:

START / STOP STATEMENTS: The symbol used for to show START / STOP statements is “Oval”.

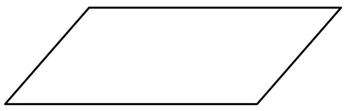
Symbol :



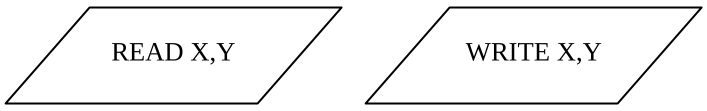
Example :

INPUT / OUTPUT STATEMENTS: The symbol used to represent input statements and output statements is “Parallelogram”.

Symbol :

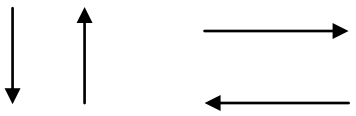


Example :



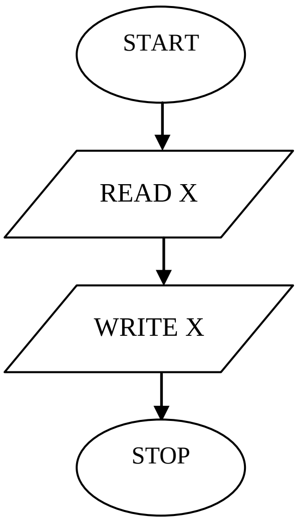
FLOW LINES: The symbol used to represent data flow from one place to another place is “Arrow”.

Symbol :



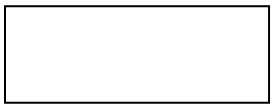
Arrow symbol actually connects every two symbols in the flowchart.

Example :



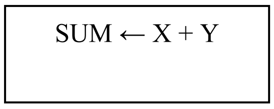
PROCESS STATEMENTS: The symbol used to represent processing instructions is "Rectangle".

Symbol :



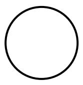
Assignment statements and calculation statements are placed inside the rectangle symbol.

Example :



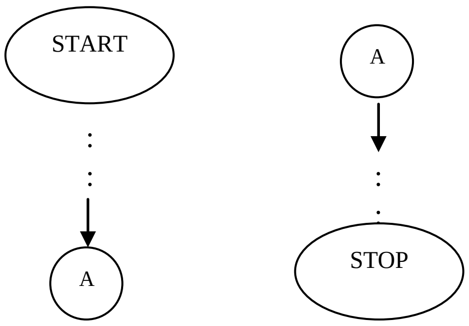
CONNECTOR SYMBOL: The symbol used to connect two parts of the program flow is "Circle".

Symbol :

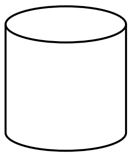
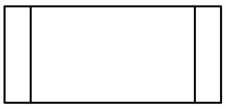
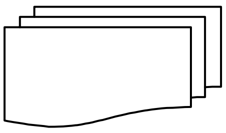
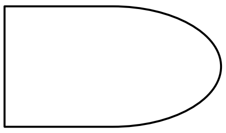
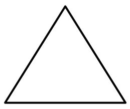
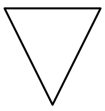
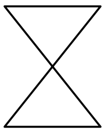


When we reach the end of a column or a page, but total chart is not finished. In this case, at the bottom of flow we use a connector to show that the flow continues at the top of the next column or page.

Example :



Additional symbols that used for designing flowcharts are:

SYMBOL	DESCRIPTION
	DATA BASE
	SUB ROUTINE
	MULTIDOCUMENTS
	IDEL OR WAITING STATE
	EXTRACTS INDIVIDUAL SETS OF DATA ITEMS
	MERGE
	MERGE AND EXTRACT ACTIONS

PROGRAM / SOFTWARE DEVELOPMENT STEPS

Software development steps are falls into 6 phases. Those are:

1. Analysis
2. Algorithm & Flowchart
3. Program Design
4. Compilation
5. Program Execution
6. Testing & Validation

1. Analysis: In first phase of the software development for a given problem statement, the problem statement must be analyzed to determine the input and output requirements.

2. Algorithm & Flowchart: A solution must be conceived and must be represented in terms of step-by-step procedure called algorithm. Then convert the algorithm into flowchart. This help us for proper way of solving the given problem statement.

3. Program Design: The flowchart and algorithm developed in the previous step is converted into actual programs by selecting any programming languages like C, C++ etc.,

4. Compilation: The process of converting the program from one language to machine language is called as compilation. Syntax errors are found at the time of compilation process. These errors are occurred due to the usage of wrong syntaxes for the statements.

Example: Sum = X + Y

There is a syntax error. Since, each and every statement in 'C' language must end with a semicolon.

5. Program Execution: It program execution phase, it may occur two types of errors.

Run-Time Errors: These errors may occur during the execution of the programs even though the program is successfully compiled without syntax errors. The most common types of run time errors are:

Example: Divide-By-Zero, Array-Out-Of-Bounds etc.,

Logical Errors: These errors may occur due to incorrect usage of the instructions in the program. These errors are neither detected during compilation or execution nor cause any stoppage to the program execution. They only produce incorrect outputs.

Logical errors are to be detected by analyzing the outputs for different possible inputs that can be applied to the program.

6. Testing & Validation: Once the program is successfully compiled and in execution phase, it must be tested and then validated with different legal input values.

With the completion of all phases, the program must be successfully produces correct results.

<u>EXAMPLE:</u>

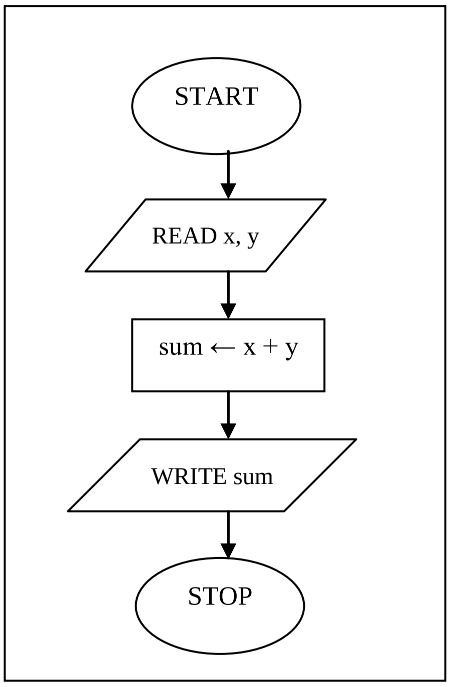
Problem Statement: *Addition of given two numbers*

Phase 1: Analysis

Input :	Read two numbers as x and y
Output :	Addition Value

Phase 2: Algorithm & Flowchart

```
1. Addition of given two
numbers

Step 1: START
Step 2: READ x, y
Step 3: sum ← x + y
Step 4: WRITE sum
Step 5: STOP
```



Phase 3: Program Design

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

main()
{
        int x,y,sum;
        clrscr();
        pritnf("\nEnter Two Numbers:");
        scanf("%d%d",&x,&y);
        sum = x+y;
        printf("\nTotal:%d",sum);
}
```

Phase 4: Compilation

Compile the program by pressing either F_9 OR ALT F_9.

Phase 5: Program Execution

Execute the program with CTRL F_9.

Phase 6: Testing & Validation

```
Enter Two Numbers:  10      20

Total:              30
```

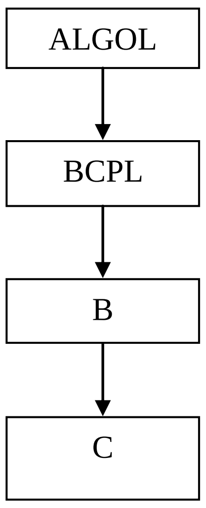
Total phases successfully completed.

***

UNIT-II

Introduction to C language – C Language elements, Variable Declarations and Data Types, Executable Statements, General Form of a C Language, Expressions, Precedence and Associativity, Expression Evaluation, Operators and Expressions, Type Conversions, Decision Statements – If and Switch Statements, Loop Control Statements – while, for, do-while Statements, Nested for Loops, Other Related Statements – break, continue, goto.

INTRODUCTION TO C LANGUAGE



ALGOL programming language is developed in the early 1960 s.

- In 1967, Martin Richards developed a language called Basic Combined Programming Language (BCPL).
- In 1970, Ken Thompson developed a simple language called B. B language was used to develop the first version UNIX operating system.
- In 1972, Dennis Ritchie developed C language.

C is a programming language developed by Dennis Ritchie at AT&T's Bell Laboratories of USA in 1972.

CHARACTERISTICS OF C LANGUAGE

1. C is structured programming language.

2. C is a middle-level language.

Depending on the efficiency and performance, programming languages are classified into two types as:

- **Problem Oriented (or) High Level Languages:** These languages have been designed to improve the program efficiency while designing programs.

  Examples: FORTRAN, COBOL, PASCAL, C, C++ etc.,

- **Machine Oriented (or) Low Level Languages:** These languages have been designed to improve the machine efficiency while designing programs.

  Examples: Assembly language and Machine language.

C is a middle level language. Since, it was designed to improve both program efficiency and machine efficiency.

3. C is a case-sensitive language.

In C language, both lower case and upper case characters are different.

4. C supports the concept of dynamic memory allocation.

5. C is a robust language. It contains rich set of built-in functions and operators that are used to design complex programs.

C CHARACTER SET

Character set of C language contains Alphabets, Digits and Special Symbols.

Alphabets	:	Upper case characters A, B, - - - - -, Z Lower case characters a, b, - - - - , z
Digits	:	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special Symbols	:	~ ‘ ! @ # ^ & * ( ) [ ] { } + - / % “ , : ; < > etc.,

CONSTANTS AND VARIABLES

A constant is a quantity that doesn’t change during the program execution.
A variable is a quantity that does change during the program execution.

Example: $3X + Y = 20$

Here, 3, 20 are constants and X, Y are variables.

KEYWORDS

The words which are predefined by the system compiler are called keywords. Keywords are also known as ”Reserved Words”. 32 keywords are available in C language.

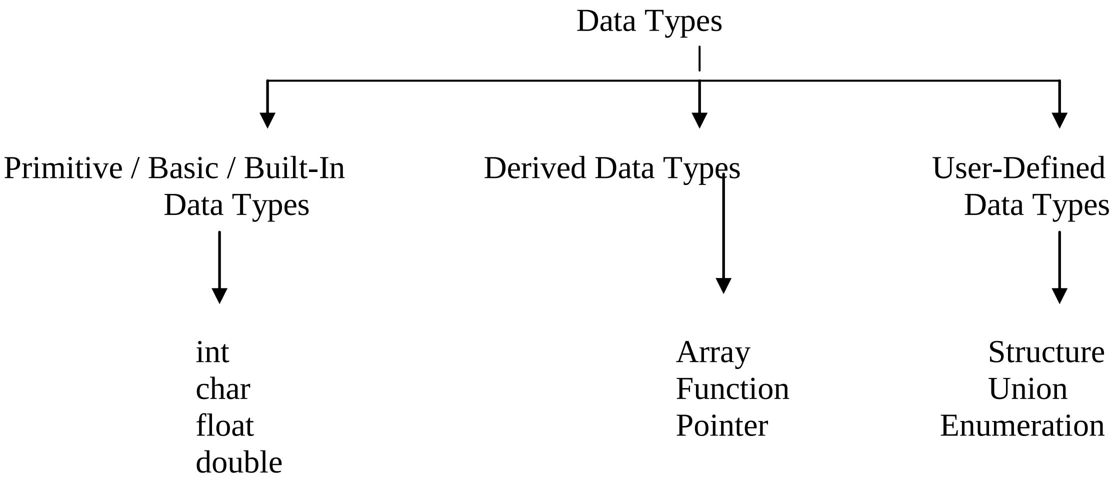
auto	double	if	static	break	else
int	struct	case	char	const	continue
default	do	enum	extern	float	far
for	goto	long	near	register	return
short	signed	switch	typedef	union	unsigned
void	while				

IDENTIFIER

An identifier is a name given to the variable, constant, array, structure etc.,

DATA TYPES

The type of the value stored in a variable is called it data type. Data types of C language can be classified into different types as:



Primitive / Basic / Built-In Data Types:

For storing character, integer and real values primitive data types are used. The following table shows different basic data types for storing integers, floating numbers, characters etc., and their memory space allocation inside the computer system.

Complex data types can build from the combination of primitive data types.

DATA TYPE	SIZE (BYTES)	RANGE
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

VARIABLES

Variable is a quantity that does change during program execution.

Declaration of Variables: All variables must be declared before they are using in the program. Declaration of a variable directs the compiler to allocate memory for the variables. The declaration statement begins with data type followed by the name of the variables.

Syntax: **DataType VariableName;**
Example: **int X;**
float p;

Multiple variables of the same data types can be declared in a single statement by separating comma operator as:

Syntax: DataType VariableName1, VariableName2, - - - , VariableNamen;
Example: **int x,y,z;**

Rules for Constructing Variable Names:

1. The first character in the variable name must be an alphabet.
2. A variable name is any combination of 1 to 8 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 40 characters.
3. No comma or blank spaces are allowed within a variable name.
4. No special symbols other than underscore can be used in a variable name.
5. Keywords can't used as variable names.

Initialization of Variables: Assigning a value to the variable at the time of its declaration is called initialization. The general format for initializing variables is:

Syntax: **DataType VariableName = Value;**
Example: **int x = 40;**

Here, the right hand side value is assigned to the left hand side variable.

C-TOKENS

In a C program the smallest individual units are known as C tokens. C language has six types of tokens as:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators
6. Special Symbols

1. Keywords: The words which are predefined by the system compiler are called keywords. Keywords are also known as "Reserved Words". 32 keywords are available in C language.

2. Identifiers: An identifier is a name given to the variable, constant, array, structure etc.,

3. Constants: A constant is a quantity that doesn't change during the program execution. C language supports several types of constants as:

Constants	:	Numerical Constants	:	Integer Constants
				Real Constants
		Character Constants	:	Character Constants
				String Constants

Integer Constants: A integer constant refers to the sequence of digits. There are three types of integers namely, Decimal Integers, Octal Integers and Hexa-Decimal Integers.

Decimal Integers consist of a set of digits 0 through 9.

Example: 23 -67 +678

Octal Integers consist of a set of digits 0 through 7 with a leading 0.

Example: 075 0123

Hexa-Decimal Integers consist of a set of digits 0 through 9, A to F (10 to 15) with a leading 0X.

Example: 0X79 0XA76E

Rules for Constructing Integer Constants:

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It could be either positive or negative. Default sign is positive.
- No comma or blank spaces are allowed within an integer constant.

Real Constants: Real constant is a quantity containing fractional parts. Real constants often called as Floating Point constants. Real constants could be written in two forms as:

Decimal Format: In decimal format the whole number is followed by a decimal point and the fraction part.

Example: 21.7896 -1045.2341

Exponential Format: The exponential format of real constant is as follows.

Example: 2179e-2

Character Constants: Character constant contains a single character enclosed within a pair of single quotation marks.

Example: 'A' '9' '#'

Rules for Constructing Character Constants:

- A character constant is a single alphabet, digit or a special symbol.
- The maximum length of a character constant can be only 1 character.

String Constants: A string constant is a sequence of characters enclosed in double quotation marks. The characters may be letters, digits, special symbols and blank spaces.

Example: "PBR VITS" "23456" "786&$34"

ESCAPE SEQUENCE CHARACTERS

C language supports some special back slash character constants which are known as escape sequence characters that are used to format he output display. Some of them are:

CONSTANT	MEANING
'\a'	Bell Sound Character
'\n'	New Line Character
'\f'	Form Feed Character
'\r'	Carriage Return Character
'\t'	Horizontal Tab Character
'\v'	Vertical Tab Character
'\0'	Null Character

COMMENTS

The lines beginning with /* and ending with */ are known as comments. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements.

Example: /* SAMPLE C LANGUAGE PROGRAM */

LIBRARY FUNCTIONS

C language is accomplished by a collection of library functions that includes a number of input and output functions.

Functions can be accessed any where with in the program simply by writing the function name followed by a list of arguments enclosed in parenthesis. Some functions do not require any arguments, then place empty parenthesis.

HEADER FILES

C includes a collection of header files that provides necessary information in support of the various library functions. These header files are entered in the program via #include statement at the beginning of the program.

Syntax: **#include<HeaderFileName>**
Or
#include “HeaderFileName”

Example: **#include<stdio.h>**
#include”stdio.h”

#include<conio.h>

stdio.h (Standard Input Ourput Header File) is a header file that provides input and output library functions.

clrscr(): It is a library function that is used to clear the screen contents.

conio.h (Console Input Output Header File) provides necessary information for clrscr() function.

READING DATA FROM KEYBOARD

scanf() is an input statement. scanf() library function is used to provide values to the variables as input data through the keyboard. The general format of scanf() function is:

Syntax: **scanf(“Control String”,&varname1, &varname2, . . , &varnamen);**

The control string consists of the format of data being received. Control string is formed with the combination of % symbol followed by the conversion characters of different data types. Control strings for different data types are:

%d	-	int
%c	-	char
%lf	-	double
%f	-	float
%u	-	unsigned int
%ld	-	long int
%o	-	octal
%x	-	hexa decimal

The scanf() statement requires ‘&’ operator called address operator. The role of the address operator is to indicate the memory location of the variable.

Note: Commas, blank spaces or any other special symbol characters are not allowed in between the one conversion character to other.

DISPLAY DATA ON MONITOR

printf() is an output statement. printf() library fuction is used to display any data on the monitor. The general format of printf() function is:

Syntax: **printf("Control String",varname1, varname2, . . , varnamen);**

Note: In printf() library functions, it is possible to place any commas, blank spaces and output format characters like escape sequence characters in the between the conversion characters.

STRUCTURE OF A C PROGRAM

The general format of a c program is:

```
Header Files

Function Prototypes

Global Variable Declarations

main()
{

        Local Variable Declarations

        - - - - -
        - - - - -        /*     PROGRAMMING LOGIC */
        - - - - -

}
```

- Variables declared inside the function are called local variables.
- Variables declared outside the function are called global variables.
- The main() is a special function used by the C system to tell the compiler that where the program execution starts. Every C program must have exactly one main function.
  - Left curly brace '{' and Right curly brace '}' indicates opening and ending function implementation.
  - All the statements between these two braces form the function body.

```
/* WRITE A C PROGRAM TO PRINT NAME OF YOUR COLLEGE */

#include<stdio.h>
#inlcude<conio.h>
main()
{
        clrscr();
        printf("PBR VITS");
}
```

SAVE	:	F2
FILENAME	:	demo.c
COMPILE	:	F9 (OR) ALT F9
RUN	:	CTRL F9
RESULT VIEW	:	ALT F5

OPERATORS AND EXPRESSIONS

An expression is a combination of operators and operands. An operator is a symbol that tells the compiler to perform certain mathematical and logical manipulations. An operand is a data item on which the particular operation takes place.

Example: **x + y = 10;**

Here,

x, y and 10 are operands ; + and = are operators

In C language, operators can be classified into different categories. Those are:

1. Arithmetic Operators:

C language provides arithmetic operators as +, -, *, /, and % operators. Any expression that forms with the combination of operands and arithmetic operators termed as arithmetic expression.

OPERATOR	MEANING
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

- Arithmetic operators are binary operators. Since, they required two operands.
- Integer division truncates any fractional part.
- While performing division,
  - If both operands are integers, result is also an integer value.
  - If either operand is float, result is also a floating point value.
- Modulo operator (%) can’t be applied on floating point numbers.

```
/* EXAMPLE PROGRAM FOR ARITHMETIC OPERATORS */

#include<stdio.h>
#include<conio.h>
main()
{
        int x,y,sum,sub,mul,div,rem;
        clrscr();
        printf("\nEnter Two Numbers:");
        scanf("%d%d",&x,&y);
        sum=x+y;
        sub=x-y;
        mul=x*y;
        div=x/y;
        rem=x%y;
        printf("\nAddition:%d",sum);
        printf("\nSubtraction:%d",sub);
        printf("\nMultiplication:%d",mul);
        printf("\nDivision:%d",div);
        printf("\nRemainder:%d",rem);
}
```

2. Assignment Operator:

Assignment operator is used to assign the result of an expression to any variables. C language provides assignment operator as =.

Syntax: **VariableName = Expression;**

Here, The right hand side value is assigned to the left hand side variable.

Example: **x = 10;**
x = x+25;

3. Relational Operators:

An expression contains relational operators such as <, >, <=, >=, == and != termed as a relational expression. These operators are used to compare the given two operand values. The result of a relational expression is either 1 or 0. Where 1 stands for TRUE and 0 stands for FALSE.

OPERATOR	MEANING
<	Is Less Than
<=	Is Less Than Or Equal To
>	Is Greater Than
>=	Is Greater Than Or Equal To
==	Is Equal To
!=	Is Not Equal To

```
/* EXAMPLE PROGRAM FOR RELATIONAL OPERATORS */

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

main()
{
        clrscr();
        printf("\nResult 1:%d",14>78);
        printf("\nResult 2:%d",14<78);
        printf("\nResult 3:%d",25<=50);
        printf("\nResult 4:%d",25>=50);
        printf("\nResult 5:%d",100==100);
        printf("\nResult 6:%d",100!=100);
}
```

4. Logical Operators:

An expression that forms with the combination of two or more relational expressions termed as a logical expression or a compound relational expression. C language provides three logical operators as:

OPERATOR	MEANING
&&	Logical AND
||	Logical OR
!	Logical NOT

Syntax: **Expression1 Operator Expression2**

Logical expressions are also produces the result values as either 1 or 0, depending on truth tables that provided by the expressions.

Exp1	Exp2	Exp1 && Exp2	Exp1 || Exp2	! Exp1
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

```
/* EXAMPLE PROGRAM FOR LOGICAL OPERATORS */

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

main()
{
        clrscr();
        printf("\nResult 1:%d",(14>78)&&(24<78));
        printf("\nResult 2:%d",(14>78)||(24<78));
        printf("\nResult 3:%d",!45);
}
```

5. Increment and Decrement Operators:

++ and -- are called increment and decrement operators. These operators are unary operands and required only one operand.

Increment Operator: Increment operator ++ adds 1 to the given operand. Depending on the placement of operator with the operand, increment operator can be sub classified into two types as: pre-increment and post-increment.

If ++ operator placed before the operand, it is termed as the pre-increment. Here, first compiler adds 1 to the operand and then result is assigned to the variable.

```
Example:    X = 7;
            Y = ++X;
            Y = 8
```

If ++ operator placed after the operand, it is termed as the post-increment. Here, first assign the value to the variable and then adds 1 to the operand.

```
Example:    X = 7;
            Y = X++;
            Y = 7
```

Decrement Operator: Decrement operator -- subtracts 1 from the given operand. Depending on the placement of operator with the operand, decrement operator can be sub classified into two types as: pre-decrement and post-decrement.

If -- operator placed before the operand, it is termed as the pre-decrement. Here, first compiler subtracts 1 from the operand and then result is assigned to the variable.

```
Example:    X = 7;
            Y = --X;
            Y = 6
```

If -- operator placed after the operand, it is termed as the post-decrement. Here, first assign the value to the variable and then subtracts 1 from the operand.

```
Example:    X = 7;
            Y = X++;
            Y = 7
```

6. Conditional Operator:

A ternary operator pair “?:” is available in C language to form a conditional expression.

Syntax: **Expression1 ? Expression2 : Expression3 ;**

Here,

- First Expression1 is evaluated. It produces either TRUE of FALSE.
- If Expression1 is TRUE, then Expression2 is evaluated and becomes the result of the total expression.
- If Expression1 is FALSE, then Expression3 is evaluated and becomes the result of the total expression.

i.e., either Expression2 or Expression3 is evaluated depending upon the out come of Expression1.

```
/* EXAMPLE PROGRAM FOR CONDITIONAL OPERATOR */

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

main()
{
        int A,B,Max;
        clrscr();
        printf("\nEnter Two Numbers:");
        scanf("%d%d",&A,&B);
        Max=(A>B)?A:B;
        printf("\nMaximum Number:%d",Max);
}
```

7. Bit-Wise Operators:

C language supports special operators known as bit-wise operators for manipulating of data at bit level. These operators can operate only on integer quantities such as int, char, short int, long int etc.,

OPERATOR	MEANING
&	Bit-Wise AND
|	Bit-Wise OR
^	Bit-Wise Exclusive OR
<<	Left Shift
>>	Right Shift
~	One’s Complement

Bit-Wise AND, Bit-Wise OR and Bit-Wise Exclusive OR follows the following bit comparison tables.

Bit1	Bit2	Bit1 & Bit2	Bit1 | Bit2	Bit1 ^ Bit2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Here,

- Bit-Wise AND compares the corresponding bits of the operands and produces 1 when both bits are 1; 0 otherwise.
- Bit-Wise OR compares the corresponding bits of the operands and produces 0 when both bits are 0; 1 otherwise.
- Bit-Wise Exclusive OR compares the corresponding bits of the operands and produces 0 when both bits are same; 1 otherwise.

For the above operations consider the following number conversion system for octal and hexa-decimal numbers.

NUMBER	OCTAL	HEXADECIMAL
0	000	0000
1	001	0001
2	010	0010
3	011	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111
8		1000
9		1001
A - 10		1010
B – 11		1011
C – 12		1100
D – 13		1101
E – 14		1110
F - 15		1111

Example:

1.	X	: 011	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1	
	Y	: 027	0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1	(Octal)
	X&Y	:	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	: 1
	X|Y	:	0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1	: 37
	X^Y	:	0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0	: 36

```
2.    X      : 0X7B     0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1
      Y      : 0X129    0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 1    (Hexadecimal)

      X&Y    :          0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1         : 29
      X|Y    :          0 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1         : 17B
      X^Y    :          0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 0         : 152
```

<u>One's Complement (or) Bit Negation</u> operator is an unary operator that complements the bits of the given operand. i.e., Bit 0 converted into Bit 1 and Bit 1 converted into Bit 0.

Example:

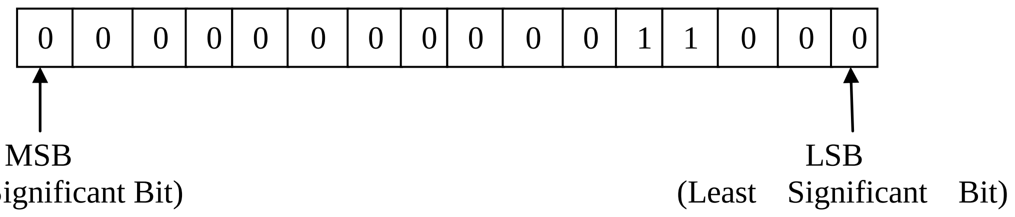
```
X      : 0X7B     0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1
~X     :          1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0        :        FF84
```

<u>Shift Operators:</u> Left shift operator (<<) and right shift operator (>>) are known as shift operators. They performs left and right shifts of their operand bits by the number of bit positions specified by the argument which must be integer and positive number.

Left Shift Operator Syntax:

VariableName << NoOfBitPositions;

Example: Let X = 24
X << 1;

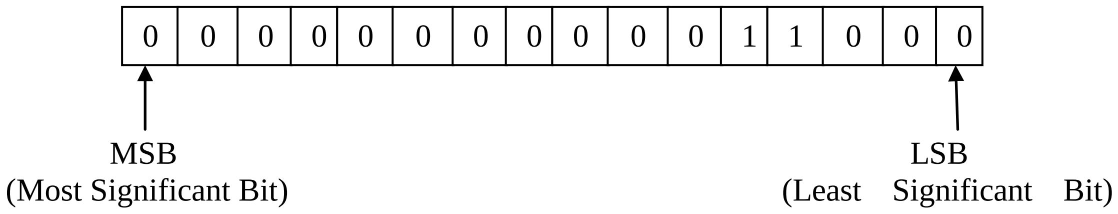


- ➢ While performing left shift operations, there is a loss of data at MSB side.
- ➢ The vacated positions at LSB side are filled with zeros.
- ➢ For each shift value by the number of bits, the result value is equivalent to multiplication by 2.

Right Shift Operator Syntax:

VariableName >> NoOfBitPositions;

Example: Let X = 24
X >> 1;



- While performing right shift operations, there is a loss of data at LSB side.
- The vacated positions at MSB side are filled with zeros.
- For each shift value by the number of bits, the result value is equivalent to division by 2.

```
/* EXAMPLE PROGRAM FOR SHIFT OPERATORS */

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

main()
{
        int x;
        clrscr();
        x=24;
        printf("\nLeft Shift Result:%d",x<<2);
        x=24;
        printf("\nRight Shift Result:%d",x>>2);
}
```

8. Special Operators:

C supports some special operators such as comma operator, sizeof operator, pointer operators (& and *) and member selection operators(. And ->).

a) Comma Operator: The comma operator is used to separate the operand from one to another.

Example: **int x,y;**

<u>*b) sizeof Operator:*</u> sizeof operator is a compile time operator, used with an operand. The operand may by a variable, a constant or a data type. sizeof operator returns the number of bytes occupied by the operand.

Syntax: **sizeof(Operand);**

```
/* EXAMPLE PROGRAM FOR SIZEOF OPERATOR */

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

main()
{
        int k;
        char p;
        float z;
        double t;
        clrscr();
        printf("\nSIZE OF K:%d Bytes",sizeof(k));
        printf("\nSIZE OF P:%d Bytes",sizeof(p));
        printf("\nSIZE OF Z:%d Bytes",sizeof(z));
        printf("\nSIZE OF T:%d Bytes",sizeof(t));
        printf("\nSIZE OF INT:%d Bytes",sizeof(int));
        printf("\nSIZE OF CHAR:%d Bytes",sizeof(char));
        printf("\nSIZE OF FLOAT:%d Bytes",sizeof(float));
        printf("\nSIZE OF DOUBLE:%d Bytes",sizeof(double));
        printf("\nSIZE OF INT VALUE:%d Bytes",sizeof(100));
        printf("\nSIZE OF CHAR VALUE:%d Bytes",sizeof('A'));
        printf("\nSIZE OF FLOAT VALUE:%d Bytes",sizeof(23.45f));
        printf("\nSIZE OF DOUBLE VALUE:%d Bytes",sizeof(456.678));
}
```

9. Additional Operators:

<u>a) Unary Minus Operator:</u> Unary minus operator changes the sign of the given operand. i.e., ‘+’ sign changed to ‘-’ and ‘-‘ sign changed to ‘+’ sign.

<u>b) Arithmetic Assignment Operators:</u>

C language supports arithmetic assignment operators as +=, -=. *=, /= and %=.

Syntax:

Exp1 += Exp2	***Equivalent To***	***Exp1 = Exp1 + Exp2***
Exp1 -= Exp2	***Equivalent To***	***Exp1 = Exp1 - Exp2***
Exp1 *= Exp2	***Equivalent To***	***Exp1 = Exp1 * Exp2***
Exp1 /= Exp2	***Equivalent To***	***Exp1 = Exp1 / Exp2***
Exp1 %= Exp2	***Equivalent To***	***Exp1 = Exp1 % Exp2***

Example:

a += 2	↔	a = a+2
a -= 5	↔	a = a-5
a *= 3	↔	a = a*3
a /= 2	↔	a = a/2
a %= 4	↔	a = a%4

OPERATOR PRECEDENCE & ASSOCIATIVITY

If more than one operator is available in a single statement, order of evaluation depends on precedence of operators. Precedence refers to rank of operators that follow order of evaluation. Highest precedence operator is evaluated first before the lowest precedence operator.

If two or more operators have same precedence, then they follow associativity. Associativity refers to the order of direction. i.e., from right to left or left to right.

RANK	OPERATOR	ASSOCIATIVITY
1	( ) [ ] -> . ++ (POSTFIX) – (POSTFIX)	L to R
2	++ (PREFIX) – (PREFIX) ! ~ sizeof unary minus &(address) *	R to L
3	*/ %	L to R
4	+ -	L to R
5	<< >>	L to R
6	< <= > >=	L to R
7	== !=	L to R
8	&	L to R
9	^	L to R
10	|	L to R
11	&&	L to R
12	||	L to R
13	?:	R to L
14	= += -= *= /= %= >>= <<= &= ^= |=	R to L
15	, (comma operator)	L to R

Example:

1. Y = 2 + 3 * 4
   Y = 2 + 12
   Y = 14

2. Y = X++ + ++Y + ++X Let X=7 and Y=19
Y = 7 + ++Y + ++X
Y = 7 + ++Y + 9
Y = 7 + 20 + 9
Y = 27 + 9
Y = 36

3. Y = (12 + 14) * 2
Y = 26 * 2
Y = 52

TYPE CONVERSIONS

The process of converting data item from one data type to another data type is called type casting. Type casting can be classified into two types. Those are:

1. Implicit Type Casting
2. Explicit Type Casting

1. Implicit Type Casting: If the operands are of different data types, the lower data type is automatically converted into the higher data type by the compiler before the operation proceeds. Such type casting is called implicit type casting. Implicit type casting is also known as automatic type conversion.

In implicit type casting, the result is in higher data type and there is no loss of data.

Example: int j,p; float f; double d,r;

r = (p * j) + (f / j) – (f + d);

$$\begin{tikzcd}
\text{int} \arrow[dr] & & \text{int} \arrow[dl] & \text{float} \arrow[dr] & & \text{int} \arrow[dl] & \text{float} \arrow[dr] & & \text{double} \arrow[dl] \\
& \text{int} \arrow[drr] & & & \text{float} \arrow[dl] & & & \text{double} \arrow[ddl] & \\
& & & \text{float} \arrow[drrr] & & & & & \\
& & & & & & \text{Double} & &
\end{tikzcd}$$

2. Explicit Type Casting: Higher data type data items can also be possible to convert into lower data types. Such a type casting is called explicit type casting. The general format of explicit type casting is:

Syntax: **(TargetDataType) Expression;**

Where,

Expression may be a constant, variable, or any expression.

Example:

```
int x=14, y=3;
float z;


z = x/y;                    z = (float) x/y;

z = 14/3;                   z = 14.00/3;

z = 4;                      z = 4.666667
```

STATEMENTS

A statement is a syntactic construction that performs some action when the program is executed. In C language, statements are classified into three types as:

1. Simple (or) Expression Statements
2. Compound (or) Block Statements
3. Control Statements

1. Simple (or) Expression Statements

A simple statement is a single statement that ended with a semicolon.

Example: int x,y;

2. Compound (or) Block Statements

Any sequence of simple statements can be grouped together and enclosed within a pair of braces termed as compound (or) block statements.

Example:

```
{
        int x=4, y=2, z;
        z=x+y;
        printf("\nResult :%d",z);
}
```

3. Control Statements

Generally, C program is a set of statements which are normally executed in sequential order from top to bottom. But sometimes it is necessary to change the order of executing statements based on certain conditions, and repeat a group of statements until certain conditions. Such statements are called control statements.

In C language, control statements are classified into three categories as:

a) Selection (or) Decision Control Statements
b) Loop (or) Iterative Control Statements
c) Branch (or) Jump Control Statements

a) Selection (or) Decision Control Statements

Selection control statements are used to skip one or more statements depending on the outcome of the logical test. C language provides decision control statements as:

i) if statement
ii) if-else statement
iii) Nested if-else statement
iv) else-if ladder
v) switch statement

i) if statement: The general format of a simple if statement is:

Syntax:

```
if(condition)
{
        Block Statements
}
Statements-X;
```

Here,

- First condition is evaluated. It produces either TRUE or FALSE.
- If the condition outcome is TRUE, then Block Statements are executed by the compiler. After executing the block statements, control reaches to Statements-X.
- If the condition outcome is FALSE, then Block Statements are skipped by the compiler and control reaches to Statements-X.
- Block Statements may be either simple or compound statements. If the statements are a simple statement, pair of braces is optional.

```
/* PROGRAM TO READ THE VALUE OF X AND PRINT Y AS Y=1 FOR X>0; Y=0 FOR
X=0 AND Y=-1 FOR X<0 */

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

main()
{
        int x,y;
        clrscr();
        printf("\nEnter x value:");
        scanf("%d",&x);
        if(x>0)
        y=1;
        if(x==0)
        y=0;
        if(x<0)
        y=-1;
        printf("\nY value is:%d",y);
}
```

ii) if-else statement: The general format of an if-else statement is:

```
Syntax:     if(condition)
            {
                    Block-I Statements
            }
            else
            {
                    Block-II Statements
            }
            Statements-X;
```

Here,

- First condition is evaluated. It produces either TRUE or FALSE.
- If the condition outcome is TRUE, then Block-I Statements are executed by the compiler. After executing the block-I statements, control reaches to Statements-X.
- If the condition outcome is FALSE, then Block-II Statements are executed by the compiler. After executing the block-II statements, control reaches to Statements-X.

```
/* PROGRAM TO CHECK WHETHER A GIVEN NUMBER IS EVEN OR ODD */

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

main()
{
        int x;
        clrscr();
        printf("\nEnter one value:");
        scanf("%d",&x);
        if(x%2==0)
        printf("\n%d is Even Number",x);
        else
        printf("\n%d is Odd Number",x);
}
```

Write a program to read three subject marks of a student and print the status of the student as either pass or fail.

Write a program to print gross salary of an employee based on i) If the basic salary is less than Rs:1500 then HRA=10% of basic salary and DA=75% of basic salary ii) If the basic salary is either equal or greater than Rs:1500 then HRA=Rs:500 and DA=90% of basic salary.

<u>iii) Nested if-else statement</u>: An if-else statement is embedded within another if-else statement such representation is called as nested if-else statement. The general format of nested if-else statement is:

Syntax:

```
if(condition1)
{
        if(condition2)
        {
                Block-I Statements
        }
        else
        {
                Block-II Statements
        }
}
else
{
        Block-III Statements
}
Statements-X;
```

Here,

- First condition1 is evaluated. It produces either TRUE or FALSE.
- If the condition1 outcome is TRUE, then control enters into condition2 section. Condition2 produces either TRUE or FALSE.
  - If Condition2 is TRUE, then Block-I Statements are executed by the compiler. After executing the block-I statements, control reaches to Statements-X.
  - If Condition2 is FALSE, then Block-II Statements are executed by the compiler. After executing the block-II statements, control reaches to Statements-X.
- If the condition1 outcome is FALSE, then Block-III Statements are executed by the compiler. After executing the block-III statements, control reaches to Statements-X.

```
/* PROGRAM TO PRINT LARGEST NUMBER FROM THE GIVEN THREE NUMBERS */

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

main()
{
        int x,y,z;
        clrscr();
        printf("\nEnter Three Numbers:");
        scanf("%d%d%d",&x,&y,&z);
        if(x>=y)
        {
                if(x>=z)
                        printf("\nMaximum Number:%d",x);
                else
                        printf("\nMaximum Number:%d",z);
        }
        else
        {
                if(z>=y)
                        printf("\nMaximum Number:%d",z);
                else
                        printf("\nMaximum Number:%d",y);
        }
}
```

iv) else-if Ladder: Another way to put if-else statements together with multi path decision is else-if ladder. The general form of else-if ladder is:

Syntax:

```
if(condition1)
{
        Block-I Statements
}
else if(condition2)
{
        Block-II Statements
}
.
.
.
else if(conditionn)
{
        Block-n Statements
}
else
{
        ElseBlock Statements
}
Statements-X
```

Here,

- First condition1 is evaluated. It produces either TRUE or FALSE.
- If the condition1 outcome is TRUE, then Block-I Statements are executed. After executing Block-I Statements control reaches to Statements-X.
- If the condition1 outcome is FALSE, then control reaches to condition2 and is evaluated. If condition2 is TRUE, then Block-II Statements are executed. After executing Block-II Statements control reaches to Statements-X and so on.

```
/* PROGRAM TO PRINT LARGEST NUMBER FROM THE GIVEN THREE NUMBERS */

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

main()
{
        int x,y,z;
        clrscr();
        printf("\nEnter Three Numbers:");
        scanf("%d%d%d",&x,&y,&z);
        if(x>=y&&x>=z)
                printf("\nMaximum Number:%d",x);
        else if(y>=z)
                printf("\nMaximum Number:%d",y);
        else
                printf("\nMaximum Number:%d",z);
}
```

Write a program to print electric bill paid by the customer based on

Consumption Units	Rate of Charge
0 – 100	Rs: 1.75
101 – 200	Rs: 2.25
201 – 300	Rs: 3.75
Above 300	Rs: 5.00

Write a program to print status of the student according to the following rules:

Average Marks	Grade
0 to 39	FAIL
40 to 49	THIRD DIVISION
50 to 59	SECOND DIVISION
60 to 79	FIRST DIVISION
80 to 100	HONOUR

v) switch Statement: switch statement is a multi-way decision that allows to place different block statements and execution depending on the result of the expression value. The general format of switch statement is:

```
Syntax:     switch(Expression)
            {
                    case value1:   Block-I Statements
                                   break;
                    case value2:   Block-II Statements
                                   break;
                    .
                    .
                    .
                    case valuen:   Block-n Statements
                                   break;
                    default:       DefaultBlock Statements
            }
```

Here,

- First Expression is evaluated and produces an integer value.
- Now, the expression value will be compared with case values value1, value2, ---, valuen. If any case value coincide with the expression value then that particular block statements are executed until break statement is encountered.
- break is a branch control statement used to transfer the control out of the loop.
- Case values value1, value2, …. are either integer constants (or) character constants.
- If the expression value doesn’t match with any case value then default block statements will be executed.
- Default block is optional block.
- Generally switch statements are used for creating menu programs.

```
/* CREATE A MENU PROGRAM TO SELECT A CHOICE AND PRINT MESSAGE AS 1
FOR RED 2 FOR GREEN 3 FOR BLUE */

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

main()
{
        int ch;
        clrscr();
        printf("\n1:RED\n2:GREEN\n3:BLUE");
        printf("\nEnter Your Choice:");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:printf("\nRED SELECTED");
                        break;
                case 2:printf("\nGREEN SELECTED");
                        break;
                case 3:printf("\nBLUE SELECTED");
                        break;
                default:printf("\nINVALID SELECTION");
        }
}
```

/* LP: PROGRAM TO PERFORM ARITHMETIC OPERATIONS USING MENU FORM */

```
#include<stdio.h>
#include<conio.h>
main()
{
  int x,y,ch,sum,sub,mul,div,rem;
  clrscr();
  printf("\nEnter two Numbers:");
  scanf("%d%d",&x,&y);
  while(1)
  {
    printf("\n1:ADDITION\n2:SUBTRACTION\n3:MULTIPLICATION\n");
    printf("4:DIVISION\n5:REMAINDER\n6:EXIT");
    printf("\nEnter Ur Choice:");
    scanf("%d",&ch);
    switch(ch)
    {
         case 1:sum=x+y;
              printf("\nADDITION RESULT:%d",sum);
              break;
         case 2:sub=x-y;
              printf("\nSUBTRACTION RESULT:%d",sub);
              break;
         case 3:mul=x*y;
              printf("\nMULTIPLICATION RESULT:%d",mul);
              break;
         case 4:div=x/y;
              printf("\nDIVISION RESULT:%d",div);
              break;
         case 5:rem=x%y;
                   printf("\nREMAINDER RESULT:%d",rem);
                   break;
         case 6:exit();
         default:printf("\nINVALID CHOICE");
    }
  }
}
```

One of the most important use of switch statement is if same block statements are necessary to executed for more than one case value, then follows the syntax as:

```
Syntax:        switch(Expression)
               {
                    case value1:
                    case value2:
                              :
                              :
                    case valuen: Block Statements;
                                   break;
                    :
                    default: defaultBlockStatements
               }
```

```
/* PROGRAM TO CHECK WHETHER A GIVEN CHARACTER IS VOWEL OR NOT */

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

main()
{
        char ch;
        clrscr();
        printf("\nEnter A Character:");
        scanf("%c",&ch);
        switch(ch)
        {
                case 'a':
                case 'A':
                case 'e':
                case 'E':
                case 'i':
                case 'I':
                case 'o':
                case 'O':
                case 'u':
                case 'U':printf("\nVOWEL");
                        break;
                default:printf("\nCONSONANT");
        }
}
```

CONDITIONAL EXPRESSION

An expression formed with the combination of operands and ternary operator pair ?: termed as a conditional expression.

Conditional expression is an alternative representation for if-else statement.

```
/* EXAMPLE PROGRAM FOR CONDITIONAL OPERATOR */

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

main()
{
        int A,B,Max;
        clrscr();
        printf("\nEnter Two Numbers:");
        scanf("%d%d",&A,&B);
        Max=(A>B)?A:B;
        printf("\nMaximum Number:%d",Max);
}
```

```
/* ALTERNATE PROGRAM WITH IF-ELSE STATEMENT */

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

main()
{
        int A,B,Max;
        clrscr();
        printf("\nEnter Two Numbers:");
        scanf("%d%d",&A,&B);
        if(A>B)
           Max=A;
        else
           Max=B;
        printf("\nMaximum Number:%d",Max);
}
```

b) Loop (or) Iterative Control Statements

Repetitive execution of one or more statements is called iteration, commonly known as loop. Loop control statements are used for repetitive execution of statements depends upon the outcome of the logical test. C language provides three loop control statements as:

i) while statement
ii) do-while statement
iii) for statement

i) while statement:

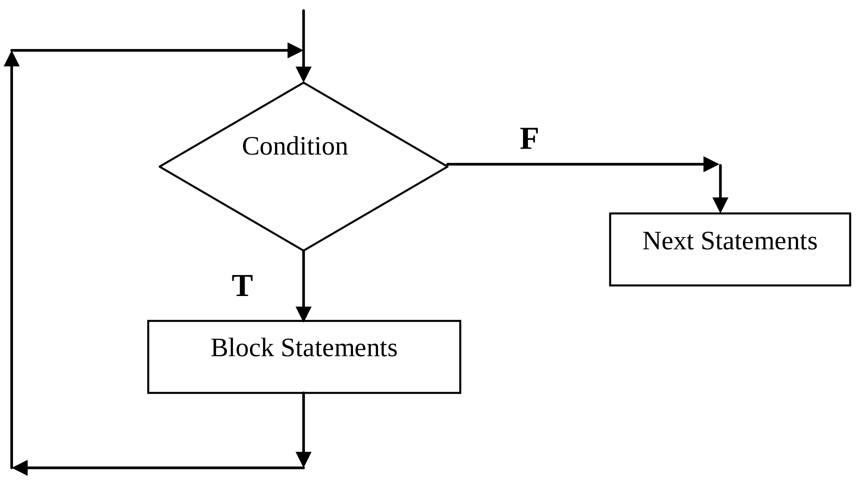
The general format of a while statement is:

```
Syntax:     while(condition)
            {
                    Block Statements
            }
            Next Statements
```

Here,

- First the condition is evaluated. It produces either TRUE or FALSE.
- If the condition is TRUE, then Block Statements will be executed by the compiler. After executing the statements, once again control reaches to condition section. Again condition is evaluated.
- The process is repeated until the condition becomes FALSE.
- When the condition reaches to FALSE, then the control is transferred out of the loop.

Flowchart:



```
/* PROGRAM TO PRINT FIRST N NATURAL NUMBERS */

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

main()
{
        int i,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        printf("\nNatural Numbers Are:");
        i=1;
        while(i<=n)
        {
                printf("   %d",i);
                i=i+1;
        }
}
```

```
/* PROGRAM TO PRINT SUM OF FIRST N NATURAL NUMBERS */

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

main()
{
        int i,n,sum;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        i=1;
        sum=0;
```

```
	while(i<=n)
	{
		sum=sum+i;
		i=i+1;
	}
	printf("\nResult is:%d",sum);
}
```

/* LP: PROGRAM TO PRINT SUM OF INDIVIDUAL DIGITS OF A GIVEN NUMBER */

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

main()
{
	int n,sum,k;
	clrscr();
	printf("\nEnter A Number:");
	scanf("%d",&n);
	sum=0;
	while(n>0)
	{
		k=n%10;
		sum=sum+k;
		n=n/10;
	}
	printf("\nSum of Individual Digits is:%d",sum);
}
```

/* PROGRAM TO PRINT REVERSE OF A GIVEN NUMBER */

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

main()
{
	int n,rev,k;
	clrscr();
	printf("\nEnter A Number:");
	scanf("%d",&n);
	rev=0;
	while(n>0)
	{
		k=n%10;
		rev=rev*10+k;
		n=n/10;
	}
	printf("\nReverse Number is:%d",rev);
}
```

```
/* PROGRAM TO PRINT FACTORIAL OF A GIVEN NUMBER */

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

main()
{
        int i,n,fact;
        clrscr();
        printf("\nEnter A Number:");
        scanf("%d",&n);
        i=1;
        fact=1;
        while(i<=n)
        {
                fact=fact*i;
                i=i+1;
        }
        printf("\nFactorial Value:%d",fact);
}
```

/* LP: PROGRAM TO PRINT THE FIRST N FIBONACCI SEQUENCE NUMBERS */

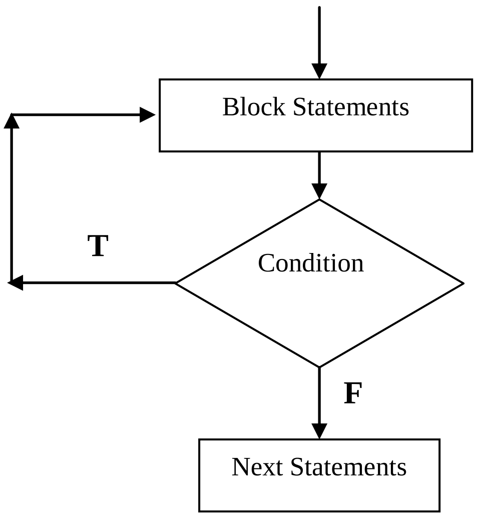
```
#include<stdio.h>
#include<conio.h>

main()
{
        int i,n,f1,f2,f3;
        clrscr();
        printf("\nEnter A Number:");
        scanf("%d",&n);
        f1=0;
        f2=1;
        printf("\nFibonacci Sequence is:");
        if(n==2)
        printf("%d      %d",f1,f2);
        else
        {
                printf("%d      %d",f1,f2);
                i=3;
                while(i<=n)
                {
                        f3=f1+f2;
                        printf(" %d",f3);
                        f1=f2;
                        f2=f3;
                        i=i+1;
                }
        }
}
```

ii) do-while statement: The general format of a do-while statement is:

Syntax:

```
do
{
        -   - -
        -   - -     Block Statements
        -   - -
}
while(condition);
Next Statements
```

Flowchart:



Here,

- ➢ First the control executes Block statements and then enters into condition section.
- ➢ Condition is evaluated and produces either true or false.
- ➢ If the condition is true, then once again control enters into block statement and executed and soon. This procedure is repeated as long as the condition becomes true.
- ➢ If the condition reaches to false, then the control comes out of the loop and reaches to next statements available after the loop.

Note: The main difference between while and do-while statements is do-while statement executed the block statements at least once even the condition becomes false.

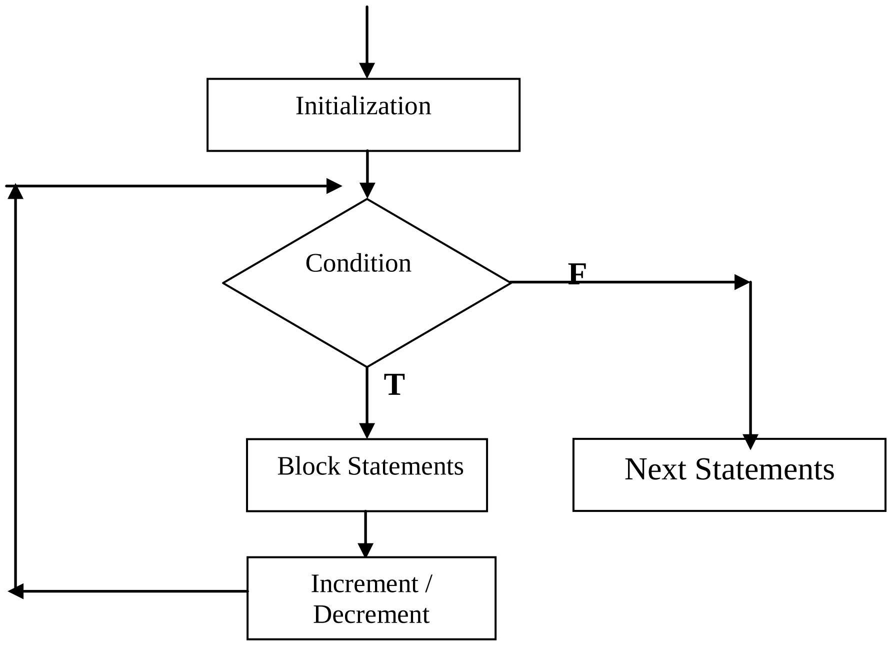
```
/* PROGRAM TO PRINT FIRST N NATURAL NUMBER USING DO-WHILE */

void main()
{
        int i=1,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        printf("\nNATURAL NUMBERS ARE:");
        do
        {
                printf("%5d",i);
                i=i+1;
        }
        while(i<=n);
}
```

<u>iii) for statement:</u> The general format of a for statement is:

Syntax:

```
for(Initialization ; Condition ; Increment/Decrement)
{
                -   - -
                -   - - Block Statements
                -   - -
}
Next Statements
```

Flowchart:



Here,

- First the control reaches to Initialization section. Initialization starts with assigning value to the variable and executes only once at the start of the loop. Then control enters into Condition section.
- In Condition section, if the outcome of the Condition is true, then control enters into Block Statements and is to be executed. After executing the statements control reaches to Increment/Decrement section.
- After incrementing or decrementing the variable in Increment/Decrement section, again control reaches to Condition section.

- This procedure is repeated as long as the condition becomes true. If the Condition becomes false, then control comes out of the loop and the control reaches to Next statements available after the loop.

```
/* PROGRAM TO PRINT FIRST N NATURAL NUMBER USING DO-WHILE */

void main()
{
        int i,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        printf("\nNATURAL NUMBERS ARE:");
        for(i=1;i<=n;i++)
        printf("%5d",i);
}
```

```
/* PROGRAM TO READ A LIST OF NUMBERS AND PRINT EVENSUM, EVENCOUNT,
ODDSUM AND ODDCOUNT */

void main()
{
        int i,n,x,ecount=0,ocount=0,esum=0,osum=0;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("\nEnter Number %d:",i);
                scanf("%d",&x);
                if(x%2==0)
                {
                        ecount=ecount+1;
                        esum=esum+x;
                }
                else
                {
                        ocount=ocount+1;
                        osum=osum+x;
                }
        }
        printf("\nEVEN COUNT = %d",ecount);
        printf("\nODD COUNT  = %d",ocount);
        printf("\nEVEN SUM   = %d",esum);
        printf("\nODD SUM    = %d",osum);
}
```

/* LP: PROGRAM TO PRINT DISTANCE TRAVELLED BY A VEHICLE IN REGULAR INVERVALS OF TIME */

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i,st,et,tt;
        float u,a,dis;
        clrscr();
        printf("\nEnter Starting time:");
        scanf("%d",&st);
        printf("\nEnter Ending time:");
        scanf("%d",&et);
        printf("\nEnter Time interval:");
        scanf("%d",&tt);
        for(i=st;i<=et;i+=tt)
        {
                printf("\nEnter velocity:");
                scanf("%f",&u);
                printf("\nEnter acceleration:");
                scanf("%f",&a);
                dis=(u*i)+(0.5*a*i*i);
                 printf("\nDISTANCE TRAVELLED IN %d TIME IS:%f",i,dis);
        }
}
```

Features of for loop:

1. More than one variable can be initialized at a time in the for statement. In such situations the variables are separated with comma operator. Similarly, the Increment/Decrement section may also have more than part.

   Ex:
   ```
   for(i = 1, j = n ; j >= 1 ; i++ , j--)
   {
           printf("\n");
           printf("%d \t %d",i,j);
   }
   ```

2. In condition section, it may have any compound relations.

3. An important feature of for loop is that one or more sections (Initialization and Increment/Decrement) can be omitted. In such situations, initialization has been done before the for statement and the control variable is incremented or decremented inside the loop.

NESTED LOOPS

Loop control statements can be placed within another loop control statement. Such representation is known as nested loops. In these situations, inner loop is completely embedded within outer loop.

```
Example:     for( i=1; i<=n; i++)                  outer loop
             {
                     ----
                     for( j=1; j<=i; j++)          inner loop
                     {
                             ---
                     }
                     ----
             }
```

Write C programs to generate the following formats

```
1) 1            2) 1          3) 3 2 1      4) *
   1 2             2 2           2 1           * *
   1 2 3           3 3 3         1             * * *
```

/* LP: PROGRAM TO CONSTRUCT PYRAMID OF NUMBERS */

```
#include<stdio.h>
#include<conio.h>
main()
{
         int i,j,k,n;
        clrscr();
        printf("\nEnter how many Rows:");
        scanf("%d",&n);
         printf("\nRESULT IS:");
         for(i=1;i<=n;i++)
        {
                printf("\n");
                for(j=20-i;j>=1;j--)
                printf(" ");
                 for(k=1;k<=i;k++)
                printf(" %d",i);
         }
}
```

/* LP: PROGRAM TO GENERATE PASCAL'S TRIANGLE */

```
#include<stdio.h>
#include<conio.h>
main()
{
        int k,n=1,q=0,x,n;
        clrscr();
        printf("\nEnter how many Rows:");
        scanf("%d",&p);
        printf("\nRESULT IS:\n");
        while(q<p)
        {
                for(k=20-q;k>=1;k--)
                printf(" ");
                 for(x=0;x<=q;x++)
                {
                        if(x==0||q==0)
                                n=1;
                        else
                                 n=(n*(q-x+1))/x;
                        printf(" %d",n);
                }
                printf("\n");
                 ++q;
         }
}
```

Write a program to check whether a given number is strong number or not.

Given number = Sum of factorial of individual digits

145 = 1!+4!+5! = 145

Write a program to print all prime numbers in between 1 and n.

Write a program to print all Armstrong numbers in between 1 and n.

Write a program to print all strong numbers in between 1 and n.

<u>c) Branch (or) Jump control statements:</u> Branch control statements are used to transfer the control from one place to another place. C language provides three branch control statements. Those are

i) break statement ii) continue statement iii) goto statement

<u>i) break statement:</u> The break statement is used in loop control statements such as while, do-while, for and switch statements to terminate the execution of the loop or switch statement. The general format of break statement is:

Syntax: break;

When the keyword break is encountered inside any C loop, control automatically skip entire loop and passes to the statements available after the loop.

```
/* Example program for break statement */

main()
{
        int i,x,sum=0;
        clrscr();
        for(i=1;i<=5;i++)
        {
                printf("\nEnter Number %d:",i);
                scanf("%d",&x);
                if(x<0)
                break;
                sum=sum+x;
        }
        printf("\nTotal:%d",sum);
}
```

/* LP: PROGRAM TO GENERATE PRIME NUMBERS BETWEEN 1 AND N */

```
#include<stdio.h>
#include<conio.h>
main()
{
  int n,i,p,j,count;
  clrscr();
  printf("\nEnter the Range Value:");
  scanf("%d",&n);
  printf("\nPRIME NUMBERS BETWEEN 1 AND %d ARE:",n);
  for(i=1;i<=n;i++)
  {
    p=i;
    count=1;
    for(j=2;j<p;j++)
    {
         if(p%j==0)
         {
         count=0;
         break;
         }
    }
    if(count==1)
    printf("\t%d",p);
  }
}
```

ii) continue statement: The continue statement is used in while, do-while and for statements to terminate an iteration. The general format of continue statement is:

Syntax: **continue;**

When the keyword continue is encountered inside any C loop, compiler skips the remaining statements available after the continue statement and control reaches to next iteration of the loop.

```
/* Example program for continue statement */

main()
{
        int i,x,sum=0;
        clrscr();
        for(i=1;i<=5;i++)
        {
                printf("\nEnter Number %d:",i);
                scanf("%d",&x);
                if(x<0)
                continue;
                sum=sum+x;
        }
        printf("\nTotal:%d",sum);
}
```

<u>iii) goto statement:</u> The goto statement is used to alter the normal sequence of program execution by transferring the control to some other part of the program. In its general form, the goto statement can be written as:

Syntax: goto Label;

Where,
Label is an identifier used to specify the target statements to which control will be necessary to transfer. The target statements must be labeled and the label must be followed by a colon as:

Syntax: Label : Statements

Each label statement with in the program must have a unique name.
Depending on passing control goto statements can be classified as forward jump and backward jump.

Forward jump Syntax:

```
_ _ _
goto Label;
_ _ _
_ _ _
Label : _ _ _
_ _ _
```

Backward jump Syntax:

```
        Label : _ _ _
        _  _ _
        _ _  _
        goto Label;
        _ _ _
        - - -
```

```
/* Example program for FORWARD JUMP */

main()
{
        int i,x,sum=0;
        clrscr();
```

```
/* Example program for BACKWARD JUMP */

main()
{
        int i,x,sum;
        clrscr();
        for(i=1;i<=5;i++)
        {
                printf("\nEnter Number %d:",i);
                scanf("%d",&x);
                if(x<0)
                goto tp;
                sum=sum+x;
        }
        tp:
        printf("\nTotal:%d",sum);
}
```

```
        tp:
        sum=0;
        for(i=1;i<=5;i++)
        {
    printf("\nEnter Number %d:",i);
    scanf("%d",&x);
        if(x<0)
        goto tp;
        sum=sum+x;
        }
        printf("\nTotal :%d",sum);
}
```

INPUT-OUTPUT STATEMENTS

C provides several functions to support input and output statements. The input/output functions are classified into two categories as:

1. Non-Formatted I/O statements
2. Formatted I/O statements

1. Non-Formatted Input/Output statements: Non-Formatted I/O statements can be carried by standard input / output library functions.

Non-Formatted Input Statements: Non-Formatted input statements are **getch(), getchar()** and **getche()** functions. All these functions are used to read a single character from the console units.

a) getchar(): The general format of a getchar() functions is:

Syntax: VariableName = getchar();

Where, VariableName is a valid identifier that is of char data type.

When this statement is encountered, the computer waits until a key is pressed. When the key is pressed after giving the character, function accepts the character value with echoing on the screen, and then assigns the character to the variable on its left hand side.

```
/* EXAMPLE PROGRAM FOR getchar() FUNCTION */

main()
{
        char ch;
        clrscr();
        printf("\nEnter a Character:");
        ch=getchar();
        printf("\nResult is:%c",ch);
}
```

```
Enter a Character:      T
Result is:              T
```

b) getch(): The general format of a getch() function is:

Syntax: VariableName = getch();

Where,

VariableName is a valid identifier that is of char data type.

When this statement is encountered, the input function reads a single character from the keyboard without echoing on the screen, and immediately assigns the character value to its left hand side without pressing any key.

```
/* EXAMPLE PROGRAM FOR getch() FUNCTION */

main()
{
        char ch;
        clrscr();
        printf("\nEnter a Character:");
        ch=getch();
        printf("\nResult is:%c",ch);
}
```

```
Enter a Character:
Result is:              T
```

c) getche(): The general format of a getche() function is:

Syntax: VariableName = getche();

Where,

VariableName is a valid identifier that is of char data type.

When this statement is encountered, the input function reads a single character from the keyboard with echoing on the screen, and immediately assigns the character value to its left hand side without pressing any key.

```
/* EXAMPLE PROGRAM FOR getche() FUNCTION */

main()
{
        char ch;
        clrscr();
        printf("\nEnter a Character:");
        ch=getche();
        printf("\nResult is:%c",ch);
}
```

```
Enter a Character:      T
Result is:              T
```

<u>Non-Formatted Output Statements:</u> Non-Formatted output statements are **putch() and putchar()**functions. All these functions are used to display a single character on the console units.

<u>a) putch():</u> The general format of a putch() functions is:

Syntax: putch(VariableName);

Where,
VariableName is a valid identifier that is of char data type.

<u>b) putchar():</u> The general format of a putchar() functions is:

Syntax: putchar(VariableName);

Where,
VariableName is a valid identifier that is of char data type.

```
/*        Example Program */

main()
{
          char ch;
          clrscr();
          printf("\nEnter a Character:");
          ch=getchar();
          printf("\nResult 1:");
          putch(ch);
          printf("\nResult 2:");
          putchar(ch);
}
```

<u>2. Formatted Input/Output Statements</u>

<u>a) Formatted Input Statement:</u> scanf() library function is used to read data from the console input device. The general format of a scanf() function is:

Syntax: scanf("Control String", &arg1, &arg2, ------ , &argn);
Example: scanf("%s%d",name, &age);

<u>b) Formatted Output Statement:</u> printf() library function is used to display the information on the console output device. The general format of a printf() function is:

Syntax: printf("Control String", arg1,arg2, ------ ,argn);

Control String consists of three types of items as:
i. Characters that will be printed on the screen as they appear.
ii. Format specifications that define the output format for display of each item.
iii. Escape sequence characters such as \n, \t, \a etc.,

Format Specifications:

I. Output of integer numbers:

i. The format specification for printing an integer number is:

%Wd

W specifies minimum field width for the output.

```
Ex:    printf("%d",7492);          7492
```

ii. If the number size is greater than the specified field width size, then the compiler automatically extends the field width to fit the value.

```
Ex:    printf("%2d",7492);         7492
```

iii. If the number size is less than the specified field width size, then the value is right justified with leading blank spaces.

```
Ex:    printf("%6d",7492);           7492
```

iv. It is possible to add zero's at blank spaces by placing a '0' before the field width specifier.

```
Ex:    printf("%06d",7492);        007492
```

v. "%#o" format specifier is used to add a leading '0' to the octal number printing.

```
Ex:    int p=071;
       printf("%o",p);             71
       printf("\n%#o",p);          071
```

vi. "%#x" format specifier is used to add a leading '0x' to the hexa decimal number printing.

```
Ex:    int p=0X71;
       printf("%x",p);             71
       printf("\n%#x",p);          0x71
```

II. Output of real numbers:

The format specification to print the decimal notation of a floating point number is:

%W.Pf

Where,

W specifies the minimum number of positions that are used for the display of the value.

P indicates the number of digits to be displayed after the decimal point.

While printing the value is rounded to P decimal places, and printed as right-justified.

```
Ex:    printf("%6.2f",7491.2579);          7491.26
       printf("\n%6.2f",5678.3);           5678.30
```

The format specification to print an exponential form of a floating point number is:

%W.Pe

Ex: printf("%6.2e",7491.2579); 7.49e+03

III. Output of character:

The general format specification to print a character is:

%Wc

The character will be displayed right-justified in the field width of W columns.

Ex: printf("%3c",'K');

IV. Output of Strings:

The general format specification to print a string is:

%Ws

The string will be displayed right-justified in the field width of W columns.

Ex: printf("%10s","KAVALI");

V. Mixed data: Different data type values can also be printed within a single printf() statement.

Ex: printf("%s%d%.2f","RAVI",23,3456.789);

***

UNIT – III

Functions – Library Functions, Top-Down Design and Structure Charts, Functions with and without Arguments, Communications Among Functions, Scope, Storage Classes – Auto, Register, Static, Extern, Scope rules, Type Qualifiers, Recursion – Recursive Functions, Preprocessor Commands.
Arrays – Declaring and Referencing Arrays, Array Subscripts, Using For Loops for Sequential Access, Using Array Elements as Function Arguments, Arrays Arguments, Multidimensional Arrays.

FUNCTIONS: A function is a self-contained program segment that carries out some specific, well-defined task. C program consists of one or more functions. One of these functions must be called by main() function. Program execution will always begin by carrying out the instructions in the main() function. Additional functions will be subordinate to the main() function.

If a program contains multiple functions, their definitions may appear in any order and they must be independent of one another i.e., one function definition can't be embedded within another.

```
Ex:     main()                          main()
        {                               {
            --                              --
            --                              --
            Statement-x                     call set
            --                              --
            --                              --
            Statement-x                     call set
            --                              --
        }                               }
                                        set()
                                        {
                                            --
                                        }
```

Advantages:

1. It avoids redundant repeated code.
2. It makes program significantly easier to understand and maintain by breaking them up into easily manageable parts.
3. main program can consist of a series of function calls rather than countless lines of code. It can be executed any number of times.
4. Well-defined and written functions may be reused in multiple programs.
5. Another main advantage is that different programmers working on one large project can divide the workload by writing different functions.

FUNCTION TYPES

C functions are classified into two types as: Library functions and User-defined functions.

<u>a) Library functions:</u> Functions that are predefined by the compiler are known as library functions.

Ex: scanf(), printf(), getch(), getche(), etc.,

For all the library functions, definitions and necessary information is available in header files. Some of the important header files are:

stdio.h	-	Standard Input and Output header file
conio.h	-	Console Input and Output header file
stdlib.h	-	Standard library header file
ctype.h	-	Character type header file
math.h	-	Mathematical header file

<u>ctype.h - Character type header file</u>

This header file provides several library functions used to character testing and conversions.

<u>Functions:</u>

1. isalnum(): Syntax: isalnum(ch)

Where, ch is a character type variable.

Function determines whether the given argument is alpha numeric or not. If the character is alpha numeric, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

2. isalpha(): Syntax: isalpha(ch)

Function determines whether the given argument is alphabet or not. If the character is alphabet, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

3. isdigit(): Syntax: isdigit(ch)

Function determines whether the given argument is digit or not. If the character is digit, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

4. isspace(): Syntax: isspace(ch)

Function determines whether the given argument is space or not. If the character is space, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

5. islower(): Syntax: islower(ch)

Function determines whether the given argument is in lowercase or not. If it is in lowercase, then returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

6. isupper(): Syntax: isupper(ch)

Function determines whether the given argument is in uppercase or not. If it is in uppercase, then returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

7. tolower(): Syntax: tolower(ch)

Function converts the given argument from upper case to lower case.

8. toupper(): Syntax: toupper(ch)

Function converts the given argument from lower case to upper case.

```
/* PROGRAM TO COUNT NUMBER OF ALPHABETS, DIGITS, WORDS AND
SPECIAL SYMBOLS IN A GIVEN LINE */

#include<ctype.h>
void main()
{
        char ch;
        int a=0,d=0,w=1,s=0;
        clrscr();
        printf("\nEnter a Line:");
        ch=getchar();
        while(ch!='\n')
        {
                if(isalpha(ch))
                        a=a+1;
                else if(isdigit(ch))
                        d=d+1;
                else if(isspace(ch))
                        w=w+1;
                else
                        s=s+1;
                ch=getchar();
        }
        printf("\nALPHABETS:%d",a);
        printf("\nDIGITS:%d",d);
        printf("\nWORDS:%d",w);
        printf("\nSPECIAL SYMBOLS:%d",s);
}
```

```
/* PROGRAM TO CONVERT LOWER CASE TO UPPER CASE AND VICE-VERSA */

#include<ctype.h>
void main()
{
        char ch;
        clrscr();
        printf("\nEnter a Line:");
        ch=getchar();
        while(ch!='\n')
        {
                if(islower(ch))
                  putchar(toupper(ch));
                else
                  putchar(tolower(ch));
                ch=getchar();
        }
}
```

math.h - Mathematical header file

This header file provides several library functions used for mathematical operations.

1. sin(): Syntax: sin(d)

Where, d is a double type argument.
Function receives a double type argument and returns the sine value as double.

Ex: printf("%lf", sin(45));

2. cos(): Syntax: cos(d)

Function receives a double type argument and returns the cosine value as double.

Ex: printf("%lf", cos(45));

3. tan(): Syntax: tan(d)

Function receives a double type argument and returns the tangent value as double.

Ex: printf("%lf", tan(45));

4. log(): Syntax: log(d)

Function receives a double type argument and returns the natural logarithm (base e) value as double.

Ex: printf("%lf", log(10));

5. log10(): Syntax: log10(d)

Function receives a double type argument and returns the logarithm (base 10) value as double.

Ex: printf("%lf", log10(10));

6. sqrt(): Syntax: sqrt(d)

Function receives a double type argument and returns the square root value as double.

Ex: printf("%lf", sqrt(81));

7. pow(): Syntax: pow(d1, d2)

Function receives two double type arguments and returns the d1 raised to the d2 power value as double.

Ex: printf("%lf", pow(3,4));

8. exp(): Syntax: exp(d)

Function receives a double type argument and returns the e to the power d value as double.

Ex: printf("%lf", exp(4));

9. ceil(): Syntax: ceil(d)

Function receives a double type argument and returns the value rounded upto the next higher integer as double.

Ex: printf("%lf", ceil(3.85));

10. floor(): Syntax: floor(d)

Function receives a double type argument and returns the value rounded down to the next lower integer as double.

Ex: printf(“%lf”, floor(3.85));

/* LP: Write a C program to find the roots of a quadratic equation */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
  float a,b,c,r1,r2,dis;
  clrscr();
  printf("\nEnter a,b and c values:");
  scanf("%f%f%f",&a,&b,&c);
  dis=(b*b)-(4*a*c);
  if(dis==0)
  {
    printf("\nROOTS ARE REAL AND EQUAL");
    r1=-b/(2*a);
    r2=-b/(2*a);
    printf("\nROOT 1:%f",r1);
    printf("\nROOT 2:%f",r2);
  }
  else if(dis>0)
  {
    printf("\nROOTS ARE REAL AND DIFFRENT");
    r1=(-b+sqrt(dis))/(2*a);
    r2=(-b-sqrt(dis))/(2*a);
    printf("\nROOT 1:%f",r1);
    printf("\nROOT 2:%f",r2);
  }
  else
    printf("\nROOTS ARE IMAGINARY");
}
```

/* LP: PROGRAM TO EVALUATE THE EXPRESSION 1+X+X^2+X^3+----+X^n */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
         int x,n,i;
        double sum,term;
        clrscr();
        T:
                printf("\nEnter X value:");
                scanf("%d",&x);
                printf("\nEnter how many terms:");
                scanf("%d",&n);
                if(n<0)
                {
                        printf("\nILLEGAL DATA");
                        printf("\nENTER ANOTHER VALUES");
                        goto T;
                }
        sum=1;
        for(i=1;i<n;i++)
        {
                term=pow(x,i);
                sum=sum+term;
        }
        printf("\nX VALUE:%d",x);
        printf("\nN VALUE:%d",n);
        printf("\nRESULT OF THE EXPRESSION:%lf",sum);
}
```

/* LP: PROGRAM TO EVALUATE THE EXPRESSION 1-X^2/2!+X^4/4!- ----+X^n */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
          int n,i,j,count,x;
          double sum,fact,term;
          clrscr();
          printf("\nEnter X value:");
          scanf("%d",&x);
          printf("\nEnter how many terms:");
          scanf("%d",&n);
          sum=1;
          count=1;
          for(i=2;i<=2*n-2;i+=2)
         {
                count=count+1;
                fact=1;
                for(j=1;j<=i;j++)
                        fact=fact*j;
                term=pow(x,i)/fact;
                if(count%2==0)
                        sum=sum-term;
                else
                        sum=sum+term;
        }
        printf("\nRESULT OF THE EXPRESSION:%lf",sum);
}
```

Write programs to evaluate the following expressions

a) X+X^2/2!+X^4/4!- ---- n terms

b) 1+X+X^2/2!+X^4/4!- ---- n terms

c) X+X^3/3!+X^5/5!- ---- n terms

d) -X+X^2!-X^3- ---- n terms

b) User-Defined functions: User-defined functions are defined by the user according to their requirements. While defining functions, function places are classified into three parts. Those are:

1. Function prototype
2. Function call
3. Function definition

1. Function prototype: Function prototype specifies the declaration statement of the function that intimates to the compiler about function information before executed it.

Function prototype includes function name, list of arguments and type of the data returned. The general format of a function prototype is:

Syntax: ReturnType FunctionName (datatype arg1, - - - - - - - , datatype argn);

Where,

- FunctionName is the name given to the function, which allows the same rules as valid identifier.
- ReturnType specifies the type of the data returned from the function to calling function.
- datatype arg1, - - - - are the arguments used to perform the function implementation. Argument names arg1, -- , argn are optional.

Example: int Fact(int); (or) int Fact(int x);

2. Function call: The function call statement invokes the function by passing arguments. The arguments which passed at the function call are called "actual arguments". The general format of a function call is:

Syntax: VariableName = FunctionName(var1, var2, --- varn);

Where, var1,----, varn are actual values passed to the function definition.

Note: The function name, type, number and order of arguments listed in the function call statement must match with that of the function prototype and function definition.

Example: p = Fact(5);

3. Funtion body (or) definition: Function definition contains implement code of the function. The general format of a function definition is:

Syntax: ReturnType FunctionName (datatype arg1, - - - - - - - , datatype argn)

```
{
        Local Variable Declarations
        --
        --      (Implementation Code)
        --
        return value;
}
```

Where,

- arg1, --- , argn arguments are known as “formal arguments”. List of values passed from the function call are temporarily stored in these formal parameters.
- Variables declared in the function definitions are called local variables.
- After completing implementation of the function, it should return a value to the calling function with a **return** statement.
- The syntax of a return statement is:

  Syntax: **return value;**

  If the function should not return any value, then simply placed as:

  Syntax: **return;**

  Or omit the statement. If we omit the statement, it is automatically invokes by the compiler default.

Example:
```
int Fact(int k)
{
        int i,f=1;
        for(i=1;i<=k;i++)
        f=f*i;
        return f;
}
```

Now, the total representation can be placed as:

```
- - -
Function proto type
-  - -
main()
{
        - -
        Function call                 Calling Function
        - -
}
Function definition
{
        - -                           Called Function
        - -
}
```

```
/* PROGRAM TO FIND FACTORIAL OF A GIVEN NUMBER USING FUNCTIONS */

int Fact(int);
main()
{
        int p,n;
        clrscr();
        printf("\nEnter a Value:");
        scanf("%d",&n);
        p=Fact(n);
        printf("\nFactorial Value:%d",p);
}
```

```
int Fact(int k)
{
        int i,f=1;
        for(i=1;i<=k;i++)
        f=f*i;
        return f;
}
```

FUNCTION CATEGORIES

Depending on whether arguments are passed or not and function returns any value or not, functions are classified into different categories as:

Case 1: Functions with no arguments and no return value
Case 2: Functions with arguments and no return value
Case 3: Functions with arguments and with return value

Case 1: When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return any value, the calling function does not receive any data from the called function.
Note: If the function does not return any value, return type must be **void.**

Case 2: In this case, calling function sends data to the called function. But, called function does not return any value after completing function implementation.

Case 3: In this case, calling function sends data to the called function and called function also returns some value to the function call after completing the function implementation.

Write a program to calculate np_r value n! / (n – r)!
Write a program to calculate nc_r value n! / r! * (n – r)!
Write a program to check a whether a given number is prime or not. If the number is prime return 1 otherwise return 0.

NESTED FUNCTIONS

Functions can also be included within another function. Such representation is called nested functions.

Syntax:

```
main()          Fun1()          Fun2()
{               {               {
  --              - -             - -
  --              Fun2();         - -
  Fun1();         - -           }
  - -           }
}
```

RECURSION

A function calls it self is called recursion and the function is called recursive function. The main advantage of recursion is to avoid the length of the code.

Ex:
```
main()
{
    --
        main();
    --
}
```

Recursion can be classified into two types as:

1. Direct recursion
2. Indirect recursion

1. Direct recursion: A method calls itself is called direct recursion.

Ex:
```
void sum()
{
        --
        sum();
        --
}
```

2. Indirect recursion: A method calls another method, which initiates to call the initial method is called indirect recursion.

Ex:
```
void sum()
{
        ---
        call();
        ---
}
void call()
{
        --
        sum();
}
```

Note: While using recursion, control may falls into infinite loop. To avoid this, when we write recursive functions, we must place a return statement with an if statement some where to force the control without calling recursive call.

So that in recursion concept, two types of conditions must be placed. Those are **base condition** and **recursive condition. Base condition** avoids the recursive call and **recursive condition** calls the recursive call.

```
/* PROGRAM TO PRINT FACTORIAL OF A GIVEN NUMBER USING RECURSION */

int Fact(int);
main()
{
	int p,n;
	clrscr();
	printf("\nEnter a Value:");
	scanf("%d",&n);
	p=Fact(n);
	printf("\nFactorial Value:%d",p);
}
int Fact(int k)
{
	if(k==0)
		return 1;
	else
		return k*Fact(k-1);
}
```

```
/* PROGRAM TO PRINT FIBONACCI SEQUENCE USING RECURSION */

int Fib(int);
main()
{
	int i,n;
	clrscr();
	printf("\nEnter How Many Values:");
	scanf("%d",&n);
	printf("\nFibonacci Sequence is:");
	for(i=1;i<=n;i++)
	printf("%5d",Fib(i));
}
int Fib(int k)
{
	if(k==0||k==1)
		return k;
	else
		return Fib(k-1)+Fib(k-2);
}
```

/* LP: PROGRAM TO PRINT FACTORIAL OF A GIVEN NUMBER USING RECURSIVE AND NON-RECURSIVE FUNCTIONS */

```
#include<stdio.h>
#include<conio.h>
int RFact(int);
int NRFact(int);
main()
{
	int n,k,p;
	 clrscr();
	printf("\nEnter a Number to find factorial:");
	scanf("%d",&n);
```

```
        k=RFact(n);
        printf("\nFACTORIAL VALUE IN RECURSIVE MANNER:%d",k);
        p=NRFact(n);
        printf("\nFACTORIAL VALUE IN NON-RECURSIVE MANNER:%d",p);
}

int RFact(int s)
{
        if(s==1)
                return 1;
        else
                return s*RFact(s-1);
}

int NRFact(int m)
{
        int i,f=1;
        for(i=1;i<=m;i++)
                f=f*i;
        return f;
}
```

/* LP: PROGRAM TO PRINT GCD OF GIVEN TWO NUMBERS USING RECURSIVE AND NON-RECURSIVE FUNCTIONS */

```
#include<stdio.h>
#include<conio.h>
int RGcd(int,int);
int NRGcd(int,int);
main()
{
        int x,y,k,p;
        clrscr();
        printf("\nEnter two Numbers:");
        scanf("%d%d",&x,&y);
        k=RGcd(x,y);
        printf("\nGCD VALUE IN RECURSIVE MANNER:%d",k);
        p=NRGcd(x,y);
        printf("\nGCD VALUE IN NON-RECURSIVE MANNER:%d",p);
}

int RGcd(int m,int n)
{
        if(n==0)
                return m;
        else
                return RGcd(n,m%n);
}

int NRGcd(int m,int n)
{
        int s;
        while(n>0)
        {
                s=m%n;
                m=n;
                n=s;
        }
         return m;
}
```

LOCAL VARIABLE Vs GLOBAL VARIABLE

Variables declared inside the function definition are called local variables. Local variables can be used by only that particular function where it was defined.

Variables declared outside the function definition are called global variables. Global variables can be used by any function at any time.

```
/* EXAMPLE PROGRAM FOR LOCAL VARIABLES */

void change();
void main()
{
        int x=10;
        clrscr();
        printf("\nValue 1:%d",x);
        change();
        printf("\nValue 2:%d",x);
}
void change()
{
        x=x+10;
        printf("\nValue 3:%d",x);
        x=x+10;
}
```

The above program produces compilation error. Since, x is a local variable for main() function and is not possible to use in change() function.

```
/* EXAMPLE PROGRAM FOR GLOBAL VARIABLE */

void change();
int x=10;
void main()
{
        clrscr();
        printf("\nValue 1:%d",x);
        change();
        printf("\nValue 2:%d",x);
}
void change()
{
        x=x+10;
        printf("\nValue 3:%d",x);
        x=x+10;
}
```

SCOPE AND LIFE TIME OF VARIABLES

Scope of a variable determines what parts of the program a variable is actually available for use. Life-time defines the period up to which a variable can be alive without destroying.

Scope and life-time of a variable can be specified by using storage classes. Storage classes provide access permission for variables.

C language provides four storage class specifiers that can be used with data type in the declaration statement of a variable. The storage classes are:

- Automatic Storage class
- External Storage class
- Static Storage class
- Register Storage class

1. Automatic Storage class:

Variables declared inside the function are precedes with the keyword '**auto**' and termed as automatic variables.

Example: auto int x;

For automatic variables memory is allocated inside the memory unit. The default value of the variable will be garbage value.

Automatic variables are created when the function is called and destroyed automatically when the function is exited. Therefore, automatic variables are local to the function in which they are declared. So that automatic variables are also known as **local** or **internal** variables.

Scope of the automatic variable is within the block or function where it is defined and the life time is until the end of the block or function.

Note: Variables declared inside the function or block without a storage class specification, by default compiler assigns '**auto'** keyword and treated as automatic variables.

```
/* EXAMPLE PROGRAM FOR AUTOMATIC VARIABLES */

void Fun();
void main()
{
        auto int x=100;
        clrscr();
        printf("\nValue 1:%d",x);
        Fun();
}
void Fun()
{
        int y=20;
        printf("\nValue 2:%d",y);
}
```

2. External Storage class:

External storage class variables are declared outside the function with a keyword '**extern**'.

Example: extern int x;

For external variables memory is allocated inside the memory unit. The default value is initialized to zero. An external variable is also known as a global variable.

Scope of the external variable is entire file and other files also and the life time is until the program execution comes to end (global).

Note: Variables declared outside the function without a storage class specification, by default compiler assigns '**extern**' keyword and treated as external variables.

```
/* EXAMPLE PROGRAM FOR EXTERNAL VARIABLES */

void Fun();
extern int x=100;
int y=200;
void main()
{
        clrscr();
        printf("\nValue 1:%d",x);
        Fun();
}
void Fun()
{
        printf("\nValue 2:%d",y);
}
```

3. Static Storage class:

Static variables are declared with the keyword '**static**' either within the function or outside the function.

Example: static int x;

For external variables memory is allocated inside the memory unit. The default value of the variable will be zero. A static variable is initialized only once when the program is compiled. Depending on the place of declaration, static variables may be classified as **Internal static variables** and **External static variables.**

Internal static variables are those which are declared inside the function. The scope of internal static variable is up to the end of the function in which they are defined and life time is throughout the remainder of the program.

```
/* EXAMPLE PROGRAM FOR INTERNAL STATIC VARIABLES */

void Fun();
void main()
{
        int i;
        clrscr();
        for(i=1;i<=3;i++)
        Fun();
}
```

```
void Fun()
{
        static int x=10;
        x=x+5;
        printf("\nValue :%d",x);
}
```

External static variables are those which are declared outside of all functions. These variables are available to all functions. The scope of external static variable is only in that file and the life time is global.

```
/* EXAMPLE PROGRAM FOR EXTERNAL STATIC VARIABLES */

void Fun();
static int x=10;
void main()
{
        int i;
        clrscr();
        for(i=1;i<=3;i++)
        {
        printf("\nValue :%d",x);
        Fun();
        }
}
void Fun()
{
        x=x+5;
}
```

The main advantage of static variables to retain updated values between the function calls.

4. Register Storage class:

User can also be possible to tell the compiler that a variable should be kept in register unit, instead of keeping in memory unit by using the keyword '**register**'. Such variables are called register variables.

Example: **register int x;**

The default value of the variable will be garbage value. The main advantage of register variable is that accessing is very fast when compared to memory unit.
Scope of the register variable is within the block or function where it is defined and the life time is until end of the block or function.

```
/* EXAMPLE PROGRAM FOR REGISTER VARIABLES */

void main()
{
        register int x;
        clrscr();
        x=10;
        printf("\nValue :%d",x);
}
```

SYMBOLIC CONSTANTS

Symbolic constant is a constant representation that does not change during program execution. #define statement is used to define symbolic constants.

Syntax: **#define symbolicname constantvalue**
Example: **#define PI 3.14159**

Rules:

i. Symbolic names have the same form as a valid identifier.
ii. No blank space between # and define.
iii. #define statements must not end with a semicolon.
iv. Symbolic names are not declared with data types.

```
/* EXAMPLE PROGRAM FOR SYMBOLIC CONSTANTS */

#define PI 3.14159
void main()
{
        float R,Area;
        clrscr();
        printf("\nEnter Radius of the Circle:");
        scanf("%f",&R);
        Area=PI*R*R;
        printf("\nAREA OF THE CIRCLE IS:%.2f",Area);
}
```

TYPE QUALIFIERS

C provides three type qualifiers **const, volatile** and **restrict.**
const and **volatile** qualifiers can be applied to any variables, but **restrict** qualifiers may only applied to pointer.

Constant variable

A variable value can be made unchanged during program execution by declaring it as a constant. For this the keyword **const** is placed before the declaration.

Syntax: **const datatype variablename;**
Example: **const int x;**

Volatile variable

Variables that can be changed at any time by external programs or the same program are called as volatile variables. For this the keyword **volatile** is placed before the declaration.

Syntax: **volatile datatype variablename;**
Example: **volatile int x;**

C PREPROCESSOR COMMANDS (DIRECTIVES)

Preprocessor is a program that processes the source code before it passes through the compiler. Preprocessing is done under the control of preprocessor command lines or directives. All preprocessor directives are begin with the symbol **‘#’** and do not require a semicolon at the end. Preprocessor directives are placed in the source program before the main () function.

C preprocessor directives can be divided into three categories as:

1. Macro substitution directives
2. File inclusion directives
3. Conditional compilation directives

1. Macro substitution directives:

a) The way of representing a symbolic constant statement is known as macro substitution.

Syntax: **#define MacroName MacroValue**
Example: **#define PI 3.14159**

In this representation, the identifier in the program replaced with the macro value. This type of macro’s are called simple macros.

b) Macro can also have arguments called argumented macro substitution.

Consider the following example.

```
#define area(x) 3.14159*x*x
void main()
{
        float R=3.97,A;
        clrscr();
        A=area(R);
        printf("\nAREA OF THE CIRCLE IS:%.2f",A);
}
```

In this program, wherever the preprocessor found area(x), it expands it into the statement as 3.14159*x*x. After the code has passed through the preprocessor, compiler works on this as:

```
void main()
{
        float R=3.97,A;
        clrscr();
        A=3.14159*3.97*3.97;
        printf("\nAREA OF THE CIRCLE IS:%.2f",A);
}
```

c) Simple programming code with multiple lines can also be replaced by using macro substitution. In such cases, a ‘\’ character is used at the end of each line while defining the macro.

```
/* EXAMPLE PROGRAM */

#define Max(x,y) if(x>y) \
                        printf("\n%d is Maximum",x); \
                else \
                        printf("\n%d is Maximum",y);

void main()
{
        int a=35,b=79;
        clrscr();
        Max(a,b);
}
```

2. *File inclusion directives:*

File inclusion directives causes' one file to be included in another file. The file may contain functions or macro definitions. The preprocessor command for a file inclusion is:

Syntax: **#include "FileName"**
(or)
#include <FileName>

For the first syntax, the command would look for the specified file name in the specified list of directories only.
For the second syntax, the command would look for the specified file name in the current directory as well as the specified list of directories.

Example: **#include "math.h"** **#include <math.h>**

3. *Conditional compilation directives:*

Conditional compilation allows the user to control the compilation process by including or excluding statements. The following commands are used in conditional compilation.

COMMAND	MEANING
#if Expression	When Expression is TRUE, then the code that follows is included for compilation
#endif	Terminates the conditional command
#else	Specifies alternative code in two way decision
#elif Expression	Specifies alternative code in multi way decision
#ifdef MacroName Expression	Abbreviation for #if defined macro
#error	Reporting errors by the preprocessor

The main advantages with conditional compilation are:

- It provides a compile-time parameterization facility.
- Decisions can be made at compile time rather than at run-time.

```
/* EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION */

#define vax 1
#define sun 0

void main()
{
        clrscr();
        #if vax
                printf("\nTRUE");
        #else
                printf("\nFALSE");
        #endif
}
```

```
/*  PROGRAM TO COUNT NUMBER OF ALPHABETS, DIGITS, WORDS AND LINES OF A GIVEN TEXT */

void main()
{
  int nl,nw,na,nd;
  char ch;
  clrscr();
  na=nd=0;
  nl=nw=1;
  printf("\nEnter a Text:\n");
  while((ch=getchar())!='#')
  {
    if((ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z'))
         na=na+1;
    if(ch>='0'&&ch<='9')
         nd=nd+1;
    if(ch==' '||ch=='\t'||ch=='\n')
         nw=nw+1;
    if(ch=='\n')
         nl=nl+1;
  }
  printf("\nNUMBER OF ALPHABETS:%d",na);
  printf("\nNUMBER OF DIGITS:%d",nd);
  printf("\nNUMBER OF WORDS     :%d",nw);
  printf("\nNUMBER OF LINES     :%d",nl);
}
```

```
/* LP:    PROGRAM TO COUNT NUMBER OF LINES, WORDS AND CHARACTERS IN A GIVEN TEXT */

void main()
{
  int nl,nw,nc;
  char ch;
  clrscr();
  nc=0;
  nl=nw=1;
  printf("\nEnter a Text:\n");
  while((ch=getchar())!='#')
  {
    if(ch==' ')
          nw=nw+1;
    else if(ch=='\n')
    {
          nl=nl+1;
          nw=nw+1;
    }
    else
          nc=nc+1;
  }
  printf("\nNUMBER OF CHARACTERS:%d",nc);
  printf("\nNUMBER OF WORDS     :%d",nw);
  printf("\nNUMBER OF LINES     :%d",nl);
}
```

***

ARRAYS

An array is a collection of homogeneous data elements that are stored in successive memory locations. An array is referred by specifying the array name followed by one or more subscripts, with each subscript is enclosed in square brackets. The number of subscripts determines the dimensionality of the array. Depending on the number of subscripts used, arrays can be classified into different types as:

1. Single (or) One dimensional arrays
2. Double (or) Two dimensional arrays
3. Multi dimensional arrays

1. Single (or) One dimensional arrays:

The general format of a one dimensional array is:

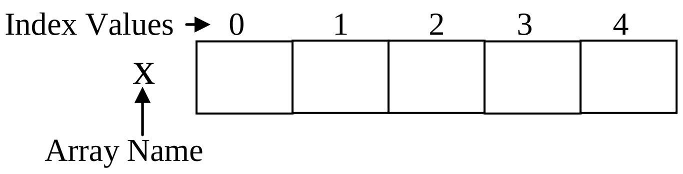
Syntax: **datatype ArrayName[size];**

Where,

- datatype specifies the type of the element that will be stored in the array
- ArrayName specifies the name of the array that follows rules as a valid identifier
- size indicates the maximum number of elements that can be stored inside the array.

Example: **int x[5];**

For this, the memory allocation will be:



Note: In C language, array index starts from 0th location.

Let ‘m’ is the size of an array, the one dimensional array can be defined as – “One dimensional array is a collection of m homogeneous data elements that are stored in m memory locations”.

With the above example,

1. Each memory location size is 2 bytes. Since, the data type is int. Compiler allocates a total of 10 bytes.
2. All memory locations share a common name as ‘x’.
3. An individual element of the array is accessed with index as:
   0th index element as x[0], 1st index element as x[1] and soon.

```
/* EXAMPLE PROGRAM TO READ A ONE DIMENSIONAL ARRAY AND PRINT IT */

void main()
{
        int n,i,x[10];
        clrscr();
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
           printf("\nEnter Element %d:",i+1);
           scanf("%d",&x[i]);
        }
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
        printf("%5d",x[i]);
}
```

1. Write a program to read an array and print it in reverse order.
2. Write a program to read an array and print sum of the array elements.
3. Write a program to read a list of values and print even and odd sum of the elements.

/* LP: PROGRAM TO FIND BOTH THE LARGEST AND SMALLEST NUMBER IN A LIST OF INTEGERS */

```
#include<stdio.h>
#include<conio.h>
main()
{
  int n,i,x[10],max,min;
  clrscr();
  printf("\nEnter how many Numbers:");
  scanf("%d",&n);
  printf("\nEnter %d Numbers:",n);
  for(i=1;i<=n;i++)
    scanf("%d",&x[i]);
  max=x[1];
  min=x[1];
  for(i=1;i<=n;i++)
  {
    if(x[i]>max)
         max=x[i];
    if(x[i]<min)
         min=x[i];
  }
  printf("\nLARGEST NUMBER:%d",max);
  printf("\nSMALLEST NUMBER:%d",min);
}
```

4. Write a program to check whether a given number is present or not in a list of values.

5. Write a program to sort the given elements of an array.

```
                for(i=0;i<=n-2;i++)
                {
                        for(j=i+1;j<=n-1;j++)
                        {
                                if(x[i]>x[j])
                                {
                                        temp=x[i];
                                        x[i]=x[j];
                                        x[j]=temp;
                                }
                        }
                }
```

6. Write a program to copy the contents of one array into another array?

```
        for(i=0,j=0;i<n;i++,j++)
        y[j]=x[i];
```

7. Write a program to insert an element into an array.

```
void main()
{
        int n,i,item,pos,x[10];
        clrscr();
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
           scanf("%d",&x[i]);
        printf("\nEnter an Element to Insert:");
        scanf("%d",&item);
        printf("\nEnter Poistion to Insert:");
        scanf("%d",&pos);
        for(i=n-1;i>=pos-1;i--)
        x[i+1]=x[i];
        x[pos-1]=item;
        printf("\nArray Elements Are:");
        for(i=0;i<=n;i++)
        printf("%5d",x[i]);
}
```

8. Write a program to delete an element from the array.

```
        printf("\nEnter Poistion to Delete:");
        scanf("%d",&pos);
        for(i=pos-1;i<=n-2;i++)
        x[i]=x[i+1];
        printf("\nArray Elements Are:");
        for(i=0;i<n-1;i++)
        printf("%5d",x[i]);
```

9. Write a program to convert the given decimal number into binary number.

```
void main()
{
        int n,i,j,x[10];
        clrscr();
        printf("\nEnter a Decimal Number:");
        scanf("%d",&n);
        i=0;
        while(n>0)
        {
                x[i]=n%2;
                i++;
                n=n/2;
        }
        printf("\nBINARY EQUIVALENT IS:");
        for(j=i-1;j>=0;j--)
        printf("%5d",x[j]);
}
```

10. Write a program to convert the given decimal number into octal number.
11. Write a program to convert the given decimal number into hexadecimal number.

/* LP: PROGRAM TO FIND 2'S COMPLEMENT OF A BINARY NUMBER */

```
void main()
{
  int n,i,j,k,p,x[10],y[10];
  clrscr();
  printf("\nEnter how many digits:");
  scanf("%d",&n);
  printf("Enter %d binary numbers(0 or 1):",n);
  for(i=0;i<n;i++)
  scanf("%d",&x[i]);
  for(i=n-1;i>=0;i--)
  {
    k=x[i];
    j=i;
    if(k==0)
         y[j]=k;
    else
    {
         y[j]=k;
```

```
          for(p=j-1;p>=0;p--)
          {
            if(x[p]==0)
              y[p]=1;
            else
              y[p]=0;
          }
          goto T;
    }
  }

  T:      printf("\n2'S COMPLEMENT OF GIVEN NUMBER IS:");
          for(i=0;i<n;i++)
          printf("%5d",y[i]);
}
```

INITIALIZATION OF SINGLE DIMENSIONAL ARRAYS

1) We can initialize the elements of the array in the same way as the ordinary variables when they are declared. The general form of initializing the one dimensional array is:

   Syntax: **datatype ArrayName[size] = {List of Values};**

   Here, List of Values is separated by comma operator.

   Example: **int k[5] = {11,22,33,44,55};** **/* 11 22 33 44 55 */**

2) While initializing elements, size may be omitted. In such cases, the compiler allocates sufficient memory for all initialized elements.

   Example: **int k[ ] = {11,22,33,44,55};** **/* 11 22 33 44 55 */**

3) While initializing elements by specifying size, even one element is initialized, by default remaining elements are initialized with ‘0’s by the compiler.

   Example: **int k[ ] = {11,22};** **/* 11 22 0 0 0 */**

2. Double (or) Two dimensional arrays:

The general format of a double dimensional array is:

Syntax: **datatype ArrayName[size1][size2];**

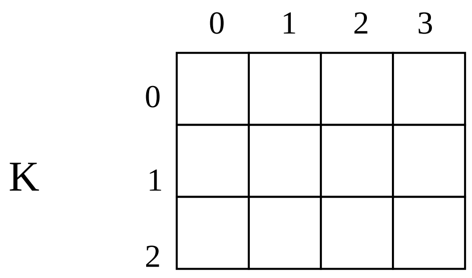
Where,

size1 specifies row size i.e., number of rows
size2 specifies column size i.e., number of columns.

Example: **int k[3][4];**

For double dimensional arrays, memory is allocated in terms of table format that contains collection of rows and columns. So, that double dimensional arrays are useful for matrix representation of given elements.

For the above example, memory allocation will be:



Let ‘m’ is the row size and ‘n’ is the column size, then a double dimensional array can be defined as – “Double dimensional array is a collection of m x n homogeneous data elements that are stored in m x n successive memory locations”.

```
/* PROGRAM TO READ A DOUBLE DIMENSIONAL ARRAY AND PRINT IT */

void main()
{
        int m,n,i,j,x[10][10];
        clrscr();
        printf("\nEnter how many rows:");
        scanf("%d",&m);
        printf("\nEnter how many columns:");
        scanf("%d",&n);
        printf("\nEnter Array Elements:");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                scanf("%d",&x[i][j]);
        }
        printf("\nArray Elements Are:");
        for(i=0;i<m;i++)
        {
                printf("\n");
                for(j=0;j<n;j++)
                printf("%5d",x[i][j]);
        }
}
```

1. Write a program to print every row sum of a given matrix.

```
        for(i=0;i<m;i++)
        {
                rsum=0;
                for(j=0;j<n;j++)
                rsum=rsum+x[i][j];
                printf("\nROW %d SUM:%d",i,rsum);
        }
```

2. Write a program to print every column sum of a given matrix.

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

```
		for(j=0;j<m;j++)
		csum=csum+x[j][i];
		printf("\nCOLUMN %d SUM:%d",i,csum);
	}
```

3. Write a program to print diagonal sum of a given matrix.

```
	psum=0;
	for(i=0;i<m;i++)
	{
		for(j=0;j<n;j++)
		{
			if(i==j)
			psum=psum+x[i][j];
		}
	}
	printf("\nDIAGONAL SUM IS:%d",psum);
	}
```

4. Write a program to print transpose of a given matrix.

```
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		y[i][j]=x[j][i];
	}
```

5. Write a program to print addition of given two matrices.

```
	void main()
	{
	int m,n,p,q,i,j,x[10][10],y[10][10],z[10][10];
	clrscr();
	printf("\nEnter row size of Matrix 1:");
	scanf("%d",&m);
	printf("\nEnter column size of Matrix 1:");
	scanf("%d",&n);
	printf("\nEnter row size of Matrix 2:");
	scanf("%d",&p);
	printf("\nEnter column size of Matrix 2:");
	scanf("%d",&q);
	if(m==p && n==q)
	{
		printf("\nEnter Matrix 1 Elements:");
		for(i=0;i<m;i++)
		{
			for(j=0;j<n;j++)
			scanf("%d",&x[i][j]);
		}
		printf("\nEnter Matrix 2 Elements:");
		for(i=0;i<p;i++)
		{
			for(j=0;j<q;j++)
			scanf("%d",&y[i][j]);
		}
		for(i=0;i<m;i++)
		{
			for(j=0;j<n;j++)
			z[i][j]=x[i][j]+y[i][j];
		}
		printf("\nAddition Matrix Elements Are:");
		for(i=0;i<m;i++)
		{
			printf("\n");
			for(j=0;j<n;j++)
			printf("%5d",z[i][j]);
		}
	}
	else
		printf("\nMATRIX ADDITON NOT POSSIBLE");
}
```

6. Write a program to print multiplication of given two matrices.

```
	void main()
	{
	int m,n,p,q,i,j,k,x[10][10],y[10][10],z[10][10];
	clrscr();
	printf("\nEnter row size of Matrix 1:");
	scanf("%d",&m);
	printf("\nEnter column size of Matrix 1:");
	scanf("%d",&n);
	printf("\nEnter row size of Matrix 2:");
	scanf("%d",&p);
	printf("\nEnter column size of Matrix 2:");
	scanf("%d",&q);
	if(n==p)
	{
		printf("\nEnter Matrix 1 Elements:");
		for(i=0;i<m;i++)
		{
			for(j=0;j<n;j++)
			scanf("%d",&x[i][j]);
		}
		printf("\nEnter Matrix 2 Elements:");
		for(i=0;i<p;i++)
		{
			for(j=0;j<q;j++)
			scanf("%d",&y[i][j]);
		}
		for(i=0;i<m;i++)
		{
			for(j=0;j<q;j++)
			{
				for(k=0;k<n;k++)
				z[i][j]+=x[i][k]+y[k][j];
			}
		}
		printf("\nMultiplication Matrix Elements Are:");
```

```
                for(i=0;i<m;i++)
                {
                        printf("\n");
                        for(j=0;j<q;j++)
                        printf("%5d",z[i][j]);
                }
        }
        else
                printf("\nMATRIX MULTIPLICATION NOT POSSIBLE");
}
```

7. Write a program to check whether a given matrix is identity matrix or not.

```
        flag=0;
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        if(i==j)
                        {
                                if(x[i][j]!=1)
                                {
                                  flag=1;
                                  goto ss;
                                }
                        }
                        else
                        {
                                if(x[i][j]!=0)
                                {
                                  flag=1;
                                  goto ss;
                                }
                        }
                }
        }
        ss: if(flag==0)
            printf("\nGIVEN MATRIX IS IDENTITY MATRIX");
          else
            printf("\nGIVEN MATRIX IS NOT A IDENTITY MATRIX");
```

__INITIALIZATION OF DOUBLE DIMENSIONAL ARRAYS__

1) Like one-dimensional arrays, double dimensional arrays can also be initialized by placing a list of values enclosed within braces as:

Syntax: **datatype ArrayName[size1][size2] = {List of Values};**

Here, List of Values is separated by comma operator.

Example: **int k[2][3] = {11,22,33,44,55,66};**

2) List of values can also be initialized in the form of a matrix representation as:

```
Syntax:     datatype ArrayName[size1][size2] = { {Row1 Values},
                                                 {Row2 Values},
                                                    -   - - - -
                                                    -   - - - -
                                                 };
```

Example: **int k[3][3] = { {1,2,3}, {4,5,6}, {2,4,5} };**

3) While initializing list of values, size1 (Row Size) may be omitted. In such cases, the compiler allocates sufficient memory for all initialized elements.

Example: **int k[ ][3] = { {1,2,3}, {4,5,6}, {2,4,5} };**

4) At the time of initializing, even one element is initialized; by default remaining elements are initialized with ‘0’s by the compiler.

Example: **int k[2][3] = {1,2,3,4};**

3. Multidimensional arrays:

Multidimensional array uses three or more dimensions. The general format of a multidimensional array is:

Syntax: **datatype ArrayName[size1][size2] - - - - - - [sizen];**
Example: **int k[2][3][4];**

The above example is a three dimensional array. Here, compiler allocates memory as in terms of tables.

Let m1, m2, --- , mn are the sizes, then a multidimensional array can be defined as – “Multidimensional array is a collection of m1 x m2 x - - - - x mn homogeneous data elements that are stored in m1 x m2 x - - - - x mn successive memory locations”.

```
/* PROGRAM TO READ A MULTDIMENSIONAL ARRAY AND PRINT IT */

void main()
{
        int m,n,p,i,j,k,x[10][10][10];
        clrscr();
        printf("\nEnter how many Tables:");
        scanf("%d",&m);
        printf("\nEnter how many rows:");
        scanf("%d",&n);
        printf("\nEnter how many columns:");
        scanf("%d",&p);
        printf("\nEnter Array Elements:");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                    for(k=0;k<p;k++)
                    scanf("%d",&x[i][j][k]);
                }
        }
        printf("\nArray Elements Are:");
        for(i=0;i<m;i++)
        {
                printf("\n");
                printf("\n");
                for(j=0;j<n;j++)
                {
                        printf("\n");
                        for(k=0;k<p;k++)
                        printf("%5d",x[i][j][k]);
                }
        }
}
```

INITIALIZATION OF MULTIDIMENSIONAL ARRAYS

1) The general form of initialization of a multidimensional array is:

Syntax: **datatype ArrayName[size1][size2]- - - -[sizen] = {List of Values};**

Here, List of Values is separated by comma operator.

Example: **int k[2][3][2] = {11,22,33,44,55,66,77,88,99,10,11,12};**

2) List of values can also be initialized in the form of a table representation as:

Example: **int k[2][3][2] = { { {1,2}, {4,5},{6,7}}, {{8,9},{10,11},{12,13}} };**

3) While initializing list of values, size1 may be omitted. In such cases, the compiler allocates sufficient memory for all initialized elements.

Example: **int k[ ][3][2] = {11,22,33,44,55,66,77,88,99,10,11,12};**

4) At the time of initializing, even one element is initialized; by default remaining elements are initialized with ‘0’s by the compiler.

Example: **int k[2][3][2] = {1,2,3,4};**

FUNCTIONS WITH ARRAYS

Like the values of simple variables, it is also possible to pass the values of an array as argument to functions. While passing array as an argument to the called function, list name of the array without any subscripts and size of the array at the calling function.

```
/* EXAMPLE TO PASS ONE DIMENSIONAL ARRAY AS AN ARGUMENT */

void sort(int[],int);
void main()
{
        int i,j,x[5]={1,9,10,2,7};
        clrscr();
        printf("\nBefore Sorting Array Elements Are:");
        for(i=0;i<5;i++)
        printf("%5d",x[i]);
        sort(x,5);
        printf("\nAfter Sorting Array Elements Are:");
        for(i=0;i<5;i++)
        printf("%5d",x[i]);

}
void sort(int k[5],int s)
{
        int i,j,temp;
        for(i=0;i<=s-2;i++)
        {
                for(j=i+1;j<=s-1;j++)
                {
                   if(k[i]>k[j])
                   {
                        temp=k[i];
                        k[i]=k[j];
                        k[j]=temp;
                   }
                }
        }
}
```

/* **LP: PROGRAM TO PERFORM ADDITION OF GIVEN TWO MATRICES USING FUNCTIONS */**

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

void read1(int[][],int,int);
void add(int[][],int[][],int[][],int,int);
void display(int[][],int,int);
void main()
{
  int r1,r2,c1,c2,x[10][10],y[10][10],z[10][10];
  clrscr();
  printf("\nEnter number of Rows and Columns of Matrix 1:");
  scanf("%d%d",&r1,&c1);
  printf("\nEnter number of Rows and Columns of Matrix 2:");
  scanf("%d%d",&r2,&c2);
  if(r1==r2 && c1==c2)
  {
    printf("\nEnter Matrix 1 Elements:");
    read1(x,r1,c1);
    printf("\nEnter Matrix 2 Elements:");
    read1(y,r2,c2);
    add(x,y,z,r1,c1);
    printf("\nResultant Matrix Elements Are:\n");
    display(z,r1,c1);
  }
  else
    printf("\nADDITION NOT POSSIBLE");
}
void read1(int k[10][10],int m,int n)
{
         int i,j;
    for(i=1;i<=m;i++)
    {
         for(j=1;j<=n;j++)
         scanf("%d",&k[i][j]);
    }
}
void display(int k[10][10],int m,int n)
{
    int i,j;
    for(i=1;i<=m;i++)
    {
         for(j=1;j<=n;j++)
         printf("%d   ",k[i][j]);
         printf("\n");
    }
}
void add(int a[10][10],int b[10][10],int c[10][10],int s,int t)
{
  int i,j;
  for(i=1;i<=s;i++)
  {
    for(j=1;j<=t;j++)
    c[i][j]=a[i][j]+b[i][j];
  }
}
```

/* **LP: PROGRAM TO PERFORM MULTIPLICATION OF GIVEN TWO MATRICES USING FUNCTIONS */**

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

void read1(int[][],int,int);
void multi(int[][],int[][],int[][],int,int,int,int);
void display(int[][],int,int);
void main()
{
  int r1,r2,c1,c2,x[10][10],y[10][10],z[10][10];
  clrscr();
  printf("\nEnter number of Rows and Columns of Matrix 1:");
  scanf("%d%d",&r1,&c1);
  printf("\nEnter number of Rows and Columns of Matrix 2:");
  scanf("%d%d",&r2,&c2);
  if(c1==r2)
  {
    printf("\nEnter Matrix 1 Elements:");
    read1(x,r1,c1);
```

```
      printf("\nEnter Matrix 2 Elements:");
      read1(y,r2,c2);
      multi(x,y,z,r1,c1,r2,c2);
      printf("\nResultant Matrix Elements Are:\n");
      display(z,r1,c2);
   }
   else
     printf("\nMULTIPLICATION NOT POSSIBLE");
}
void read1(int k[10][10],int m,int n)
{
             int i,j;
      for(i=1;i<=m;i++)
      {
              for(j=1;j<=n;j++)
              scanf("%d",&k[i][j]);
      }
}
void display(int k[10][10],int m,int n)
{
      int i,j;
      for(i=1;i<=m;i++)
      {
              for(j=1;j<=n;j++)
              printf("%d   ",k[i][j]);
              printf("\n");
      }
}
void multi(int a[10][10],int b[10][10],int c[10][10],int m,int n,int p,int q)
{
   int i,j,k;
   for(i=1;i<=m;i++)
   {
      for(j=1;j<=q;j++)
      {
             c[i][j]=0;
             for(k=1;k<=p;k++)
             c[i][j]+=a[i][k]+b[k][j];
      }
   }
}
```

***

UNIT – IV

Pointers – Introduction, Features of Pointers, Pointer Declaration, Arithmetic Operations With Pointers, Pointers and Arrays, Pointers and Two-Dimensional Arrays, Array of Pointers, Pointers to Pointers, Void Pointers, Memory Allocation functions, Programming Applications, Pointer to Functions, Command-Line Arguments.
Strings – String Basics, String Library Functions, Longer Strings, String Comparison, Array of Pointers, Character Operations, String-To-Number and Number-To-String Conversions, Pointers and Strings.

STRINGS

A String is defined as "an array of characters". The general form of declaration of a string variable is:

Syntax: **char StringName [size];**

Ex: char str[10];

- Any group of characters defined between double quotation marks is a constant string.
  Ex: "KAVALI"
- Each string is terminated by the null character ('\0'), which indicates the end of the string.
- When the compile assigns a character string to a character array, it automatically supplies a null character at the end of the string.
- %s format specification is used to read and write a string.
- The ampersand '&' symbol is not required before the variable name in scanf() function while reading strings.
- Main problem with the scanf() function is that it terminates its input on the first white space it found.

```
/* EXAMPLE PROGRAM TO READ AND PRINT A STRING */

main()
{
	char city[10];
	clrscr();
	printf("\nEnter City Name:");
	scanf("%s",city);
	printf("\nRESULT:%s",city);
}
```

```
/* PROGRAM TO READ A CHARACTER ARRAY AND PRINT ENTIRE STRING*/

main()
{
	char city[10];
	int i,n;
	clrscr();
```

```
        printf("\nEnter how many Characters:");
        scanf("%d",&n);
        printf("\nEnter %d Characters:",n);
        for(i=0;i<n;i++)
        scanf("%c",&city[i]);
        city[i]='\0';
        printf("\nRESULT:%s",city);
}
```

gets() and puts() functions: gets() and puts() functions are library functions used for to read and write an entire line of text including blank spaces.

Syntax: gets(string);
puts(string);

Each of these library functions accepts a single argument as string. These functions are defined in "stdio.h" head file. In gets(), the string will be entered from the keyboard and terminated with a new line character.

```
/* EXAMPLE PROGRAM */

main()
{
        char str[50];
        clrscr();
        printf("\nEnter a line of text:");
        gets(str);
        printf("\nRESULT:");
        puts(str);
}
```

STRING HANDLING FUNCTIONS

string.h header file provides a large number of string handling functions that can be used to carry out many of the string manipulations. Some of the important library functions are:

1. **strlen():** It refers to string length.
This function counts and returns the number of characters in a string. It gives actual length of the string including blank spaces excluding null character ('\0'). The general format of this function is:
Syntax: int strlen(string);

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str[50];
        int n;
        clrscr();
        printf("\nEnter a string:");
        gets(str);
        n=strlen(str);
        printf("\nSTRING LENGTH IS:%d",n);
}
```

2. **strcat():** It refers to string concatenation function.
This function is used to add the given two strings. The general form of this function is:

Syntax: **strcat(targetstring , sourcestring);**

Here, the sourcestring add at the end of the targetstring.

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        printf("\nEnter string 2:");
        gets(str2);
        strcat(str1,str2);
        printf("\nSTRING 1:");
        puts(str1);
        printf("\nSTRING 2:");
        puts(str2);
}
```

3. **strcpy():** It refers to string copy function.
This function copies contents of one string into another string. The general form of this function is:

Syntax: **strcpy(targetstring , sourcestring);**

Note: While declaring target string, its size should be equal or greater than the size of the source string.

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        printf("\nEnter string 2:");
        gets(str2);
        strcpy(str1,str2);
        printf("\nSTRING 1:");
        puts(str1);
        printf("\nSTRING 2:");
        puts(str2);
}
```

4. **strcmp():** It refers to string comparison function.

This function compares the given two strings and find out whether they are same or different. The general form of this function is:

Syntax: int strcmp(string1 , string2);

This function performs comparison between two strings character by character until there is a mismatch or end of the strings is reached, which ever occurs first. If the two strings are identical, function returns 0 value. If they are not identical, function returns numerical difference between the ASCII values of the non-matching characters.

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        int k;
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        printf("\nEnter string 2:");
        gets(str2);
        k=strcmp(str1,str2);
        if(k>0)
           printf("\nSTRING 1 IS GREATER THAN STRING 2");
        else if(k<0)
           printf("\nSTRING 2 IS GREATER THAN STRING 1");
        else
           printf("\nBOTH STRIGNS ARE IDENTICAL");
}
```

5. **<u>strcmpi():</u>** It refers to string comparison ignore case function.
This function is used to compare the given two strings without case consideration. The general form of this function is:

Syntax: int strcmpi(string1 , string2);

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        printf("\nEnter string 2:");
        gets(str2);
        if((strcmpi(str1,str2))==0)
            printf("\nBOTH STRIGNS ARE IDENTICAL");
        else
            printf("\nBOTH STRINGS ARE NOT IDENTICAL");
}
```

6. **<u>strrev():</u>** It refers to string reverse function.
This function is used to reverse the contents of the given string. The general form of this function is:

Syntax: strrev(string);

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str[10];
        clrscr();
        printf("\nEnter a String:");
        gets(str);
        strrev(str);
        printf("\nRESULT:");
        puts(str);
}
```

7. **<u>strlwr():</u>** It refers to string lower function.
This function is used to convert all characters in the given string from upper case to lower case characters. The general form of this function is:

Syntax: strlwr(string);

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str[10];
        clrscr();
        printf("\nEnter a String:");
        gets(str);
        strlwr(str);
        printf("\nRESULT:");
        puts(str);
}
```

8. **strupr():** It refers to string upper function.
This function is used to convert all characters in the given string from lower case to upper case characters. The general form of this function is:

Syntax: strupr(string);

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str[10];
        clrscr();
        printf("\nEnter a String:");
        gets(str);
        strupr(str);
        printf("\nRESULT:");
        puts(str);
}
```

9. **strncat():** The general format of strncat() function is:

Syntax: strncat(targetstring , sourcestring, n);

Where, n is an integer argument. This function appends atmost n characters of source string to target string.

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        printf("\nEnter string 2:");
        gets(str2);
        strncat(str1,str2,3);
        printf("\nSTRING 1:");
        puts(str1);
        printf("\nSTRING 2:");
        puts(str2);
}
```

10. **strncpy():** The general format of strncpy() function is:

Syntax: **strncpy(targetstring , sourcestring , n);**

This function copies atmost n characters of source string to target string.

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        strncpy(str2,str1,3);
        str2[3]='\0';
        printf("\nRESULT 1:");
        puts(str1);
        printf("\nRESULT 2:");
        puts(str2);
}
```

11. **strncmp():** The general format of strncmp() function is:

Syntax: **strncmp(string1 , string2 , n);**

This function compares atmost n characters of the two strings.

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        printf("\nEnter string 2:");
        gets(str2);
        printf("\nRESULT:%d",strncmp(str1,str2,3));
}
```

12. **strncmpi():** The general format of strncmpi() function is:

Syntax: **strncmpi(string1 , string2 , n);**

This function compares atmost n characters of the two strings without case consideration.

```
/* EXAMPLE PROGRAM */

#include<string.h>
main()
{
        char str1[50],str2[50];
        clrscr();
        printf("\nEnter string 1:");
        gets(str1);
        printf("\nEnter string 2:");
        gets(str2);
        printf("\nRESULT:%d",strncmpi(str1,str2,3));
}
```

PROGRAMS

1. Write a program to print length of the string without using library functions.

```
main()
{
    char str[50];
    int i;
    clrscr();
    printf("\nEnter a String:");
    gets(str);
    for(i=0;str[i]!='\0';i++);
    printf("\nLength is:%d",i);
}
```

2. Write a program to copy contents of one string to another without using library functions.

```
main()
{
    char str1[50],str2[50];
    int i;
    clrscr();
    printf("\nEnter a String:");
    gets(str1);
    printf("\nOriginal String is:");
    puts(str1);
    for(i=0;str1[i]!='\0';i++)
    str2[i]=str1[i];
    str2[i]='\0';
    printf("\nCopied String is:");
    puts(str2);
}
```

3. Write a program to perform string concatenation without using library functions.

```
main()
{
    char str1[50],str2[50];
    int i,j;
    clrscr();
    printf("\nEnter String 1:");
    gets(str1);
    printf("\nEnter String 2:");
    gets(str2);
    for(i=0;str1[i]!='\0';i++);
    for(j=0;str2[j]!='\0';j++)
    str1[i+j]=str2[j];
    str1[i+j]='\0';
    printf("\nResult String is:");
    puts(str1);
}
```

4. Write a program to perform string comparison without using library functions.

```
main()
{
    char str1[50],str2[50];
    int i,k;
    clrscr();
    printf("\nEnter String 1:");
    gets(str1);
    printf("\nEnter String 2:");
    gets(str2);
    k=0;
    for(i=0;str1[i]!='\0'||str2[i]!='\0';i++)
    {
            if(str1[i]!=str2[i])
            {
                    k=str1[i]-str2[i];
                    goto pp;
            }
    }
    pp:
            if(k>0)
              printf("\nSTRING 1 IS GREATER THAN STRING 2");
            else if(k<0)
              printf("\nSTRING 1 IS LESS THAN STRING 2");
            else
              printf("\n BOTH ARE SAME");
}
```

/* LP: Write a program to check whether the given string is palindrome or not */

```
#include<string.h>
main()
{
    char str1[50],str2[50];
    int i,k;
    clrscr();
    printf("\nEnter a String :");
    gets(str1);
    strcpy(str2,str1);
    strrev(str1);
    if((strcmp(str1,str2))==0)
      printf("\nPALINDROME");
    else
      printf("\nNOT PALINDROME");
}
```

5. Write a program to check whether the given string is palindrome or not without using library functions.

```
#include<string.h>
main()
{
    char str1[50];
    int i,n,j,k=0;
    clrscr();
    printf("\nEnter a String :");
    gets(str1);
    for(n=0;str1[n]!='\0';n++);
    for(i=0,j=n-1;i<j;i++,j--)
    {
            if(str1[i]!=str1[j])
            {
                    k=1;
                    break;
            }
    }
    if(k==0)
      printf("\nPLAINDROME");
    else
      printf("\nNOT PALINDROME");
}
```

6. Write a program to insert a sub string into main string from a given position.

```
#include<string.h>
main()
{
    char str[50],sub[50];
    int i,j,len1,len2,pos;
    clrscr();
    printf("\nEnter Main String:");
    gets(str);
    printf("\nEnter Sub String to Insert:");
    gets(sub);
    printf("\nEnter Position to Insert:");
    scanf("%d",&pos);
    len1=strlen(str);
    len2=strlen(sub);
    if(pos-1>len1)
      printf("\nINSERTION NOT POSSIBLE");
    else
    {
            for(i=len1;i>=pos-1;i--)
            str[i+len2]=str[i];
            for(i=pos-1,j=0;j<len2;i++,j++)
            str[i]=sub[j];
    }
    printf("\nResultant String is:");
    puts(str);
}
```

7. Write a program to delete n characters from a given position in a main string.

```
#include<string.h>
main()
{
    char str[50];
    int i,len,pos,nc;
    clrscr();
    printf("\nEnter Main String:");
    gets(str);
    printf("\nEnter Position to Delete:");
    scanf("%d",&pos);
    printf("\nEnter Number of Characters to Delete:");
    scanf("%d",&nc);
    len=strlen(str);
    for(i=pos-1;i<len;i++)
    str[i]=str[i+nc];
    printf("\nResultant String is:");
    puts(str);
}
```

/* LP: Write a program to convert the given Roman numeral into Decimal number */

```
#include<stdio.h>
#include<conio.h>
int index(char[],char);
main()
{
char R[]={'I','V','X','L','C','D','M','\0'},a[10];
int D[]={1,5,10,50,100,500,1000},i,m1,m2,s=0;
clrscr();
roman:
printf("\nEnter a roman numeber(I,V,X,L,C,D,M):");
gets(a);
for(i=0;a[i]!='\0';i++)
{
    if(index(R,a[i])==-1)
    {
            printf("\nInvalid Number");
            goto roman;
    }
}
for(i=0;a[i]!='\0';i++)
{
      m1=index(R,a[i]);
      m2=index(R,a[i+1]);
      if(a[i+1]=='\0')
                  s=s+D[m1];
      else if(D[m1]>=D[m2])
                  s=s+D[m1];
      else if(D[m1]<D[m2])
                  s=s-D[m1];
}
printf("Decimal number=%d",s);
getch();
}

int index(char ch[],char k)
{
int i;
for(i=0;ch[i]!='\0';i++)
{
      if(ch[i]==k)
      return i;
}
return -1;
}
```

/* LP: Write a program to display the index of a search string in the main string */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int search(char[],char[]);

main()
{
   char str[50],sub[50];
   int k;
   clrscr();
   printf("\nEnter Main String:");
   gets(str);
   printf("\nEnter Sub String:");
   gets(sub);
   k=search(str,sub);
   if(k!=-1)
      printf("\nSub String Found at %d Position",k);
   else
      printf("\nSUB STRING NOT FOUND");
}

int search(char Ms[50],char Ss[50])
{
    int i,j=0,p,len;
    len=strlen(Ss);
```

```
        for(i=0;Ms[i]!='\0';i++)
        {
          if(Ms[i]==Ss[j])
          {
           for(j=1,p=i+1;j<len;j++,p++)
           {
             if(Ms[p]!=Ss[j])
                  goto T;
           }
           return i;
          }
          T:   j=0;
        }
        return -1;
     }
```

DATA CONVERSION FUNCTIONS

atoi() and itoa() are data conversion functions available in stdlib.h header file.

1. **atoi():** This function converts a string of digits into an integer value. The general form of the atoi() function is:

Syntax: **int atoi(string);**

```
/* EXAMPLE PROGRAM */
#include<stdlib.h>
main()
{
        char x[10]="2009";
        int k;
        clrscr();
        k=atoi(x)+2;
        printf("\nRESULT VALUE:%d",k);
}
```

2. **itoa():** This function converts an integer value into a string. The general form of the itoa() function is:

Syntax: **char *itoa(int , char[] , int);**

```
 /* EXAMPLE PROGRAM */
#include<stdlib.h>
main()
{
        char x[10];
        int k=49;
        clrscr();
        itoa(k,x,10);
        printf("\nDecimal Format:");
        puts(x);
        printf("\nOctal Format:");
        itoa(k,x,8);
        puts(x);
        printf("\nHexadecimal Format:");
        itoa(k,x,16);
        puts(x);
}
```

INITIALIZATION OF STRINGS

Character array may also be initialized, when they are declared. C permits a character array to be initialized in either of the following two forms:

Syntax 1: char stringname[size] = “string”;
Syntax 2: char stringname[size] = { ‘value1’ , ‘value2’, … };

Note: When we initializing a character array by listing its individual characters, we must supply explicitly the null character.

Ex: 1 char city[10] = “KAVALI”;
Ex: 2 char city[10] = {‘K’ , ‘A’ , ‘V’ , ‘A’ , ‘L’ , ‘I’ , ‘\0’};

C also permits to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically based on the number of elements initialized.

Ex: char city[] = “CHENNAI”;

TWO DIMENSIONAL ARRAY OF CHARACTERS

A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list. The general format of a two dimensional character array is:

Syntax: datatype stringname[size1][size2];

Ex: char str[10][15];

```
/* PROGRAM TO PRINT LIST OF NAMES IN ALPHABETICAL ORDER */

#include<stdio.h>
#include<string.h>
main()
{
        char x[15][15],temp[15];
        int i,j,n;
        clrscr();
        printf("\nEnter how many names:");
        scanf("%d",&n);
        printf("\nEnter %d Names:",n);
        for(i=0;i<n;i++)
        {
                fflush(stdin);
                gets(x[i]);
        }
        for(i=0;i<=n-2;i++)
        {
                for(j=i+1;j<=n-1;j++)
                {
                        if((strcmp(x[i],x[j]))>0)
                        {
                                strcpy(temp,x[i]);
                                strcpy(x[i],x[j]);
                                strcpy(x[j],temp);
                        }
                }
        }
        printf("\nLIST OF NAMES IN ALPHABETICAL ORDER ARE:\n");
        for(i=0;i<=n-1;i++)
        puts(x[i]);
}
```

INITIALIZATION OF TWO DIMENSIONAL CHARACTER ARRAYS

Syntax 1:

```
datatype arrayname[size1][size2] = {“string1” , “string2” , ……… };
```

Syntax 2:

```
dataype arrayname[size1][size2] = {   {‘char1’ , ‘char2’ , …… , ‘\0’} ,
                                      {‘char1’ , ‘char2’ , …… , ‘\0’} ,
                                                -
                                                -
                                      {‘char1’ , ‘char2’ , …… , ‘\0’}
                                  };
```

```
Ex 1:   char x[3][10]={ “India” , “Pakistan” , “Srilanka”};
Ex 2:   char x[3][10]={    {‘I’  , ‘n’ , ‘d’ , ‘i’ , ‘a’ , ‘\0’} ,
                           {‘P’ , ‘a’ , ‘k’ , ‘i’ , ‘s’ , ‘t’ , ‘a’ , ‘n’ , ‘\0’} ,
                           {‘S’ , ‘r’ , ‘i’ , ‘l’ , ‘a’ , ‘n’ , ‘k’ , ‘a’ }
                      };
```

Size1 (Row Size) is optional while initializing character arrays. For this type of representation uses like,

```
Ex 1:   char x[ ][10]={ “India” , “Pakistan” , “Srilanka”};
Ex 2:   char x[ ][10]={    {‘I’  , ‘n’ , ‘d’ , ‘i’ , ‘a’ , ‘\0’} ,
                           {‘P’ , ‘a’ , ‘k’ , ‘i’ , ‘s’ , ‘t’ , ‘a’ , ‘n’ , ‘\0’} ,
                           {‘S’ , ‘r’ , ‘i’ , ‘l’ , ‘a’ , ‘n’ , ‘k’ , ‘a’ }
                      };
```

/* LP: WRITE A PROGRAM THAT USES FUNCTIONS TO PERFORM I) INSERT A SUB-STRING INTO A MAIN STRING FROM AGIVEN POSITION II) DELETE N CHARACTERS FROM A GIVEN POSITION */

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

void Insert(char[],char[],int);
void Delete(char[],int,int);

main()
{
  char str[50],sub[50];
  int ch,pos,m,n;
  clrscr();
  printf("\nEnter the Main String:");
  gets(str);

  while(1)
  {
    printf("\n1:INSERTION\n2:DELETION\n3:DISPLAY\n4:EXIT\n");
    printf("Enter Ur Choice:");
    scanf("%d",&ch);
    switch(ch)
    {
         case 1:fflush(stdin);
                 printf("\nEnter Sub String to Insert:");
                 gets(sub);
                 printf("\nEnter Position to Insert:");
                 scanf("%d",&pos);
                 Insert(str,sub,pos);
                 break;
         case 2:printf("\nEnter how many characters to delete:");
                 scanf("%d",&n);
                 printf("\nEnter Position to Delete:");
                 scanf("%d",&m);
                 Delete(str,n,m);
                 break;
         case 3:printf("\nRESULT STRING IS:");
                 puts(str);
                 break;
         case 4:exit();
    }
  }
}


void Insert(char Ms[50],char Ss[10],int p)
{
  int len1,len2,i,j;
  len1=strlen(Ms);
  len2=strlen(Ss);
  if(p-1>len1)
    printf("\nINSERTION NOT POSSIBLE");
  else
  {
    for(i=len1;i>=p-1;i--)
          Ms[i+len2]=Ms[i];
    for(i=p-1,j=0;j<len2;i++,j++)
          Ms[i]=Ss[j];
  }
}

void Delete(char Ms[50],int n,int m)
{
  int i,len1;
  len1=strlen(Ms);
  if(m>len1-1)
    printf("\nDELETION NOT POSSIBLE");
  else
  {
    for(i=m-1;i<len1;i++)
         Ms[i]=Ms[i+n];
  }
}
```

****

POINTERS

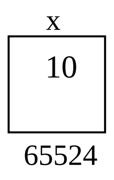
Accessing the address of a variable: Suppose 'x' is a variable that represents a particular data item as:

Example: int x;

Then compiler will allocate a memory cell for this data item. The data item can be accessed, if we know the location (i.e., address) of the item. The address of a variable is accessed with **address operator &**, which is a unary operator preceded by the variable. Address of the variable is always unsigned integer.

Example: &x;

```
/* EXAMPLE PROGRAM FOR ADDRESS OPERATOR */

main()
{
        int x;
        clrscr();
        printf("\nAddress of x:%u",&x);
        x=10;
        printf("\nValue of x:%d",x);
}
```



Note:

1. The address operator & can be used only with a simple variable (or) an array element.
2. Address operator cannot act upon constants, and arithmetic expression.

Example:	&125	->	INVALID
	&(x+y)	->	VALID

POINTERS

A pointer is a variable which holds the memory address of another variable.

Pointer Declaration: As similar to an ordinary variable, pointer variables must be declared before they are using. A pointer variable is declared by preceding its name with an asterisk * symbol. The symbol indicates that the variable is a pointer variable. The general format of declaring a pointer variable is:

Syntax: datatype *ptrvariable;

Where,

- ptrvariable is name of the pointer variable that follows same rules as a valid identifier.
- '*' tells the compiler that ptrvariable is a pointer variable.
- datatype is the type of the data that the pointer is allowed to hold the address.

Example: int *ptr;

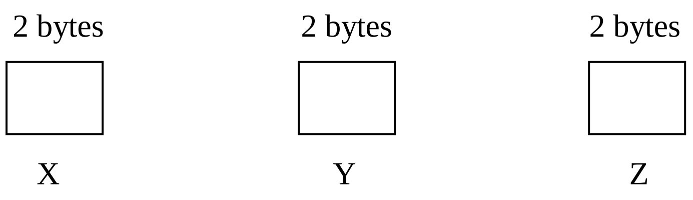
Here, ptr is a pointer variable that can points to an integer variable.

Example: char *ptr;

Here, ptr is a pointer variable that can points to a character variable.

Note: Any type of pointer occupies only 2 bytes of memory. Since, pointer holds address of the variable which is always an unsigned integer.

Example:
```
int *X;
float *Y;
char *Z;
```



```
/* EXAMPLE PROGRAM */

main()
{
        char *x;
        int *y;
        float *z;
        clrscr();
        printf("\nSize of X is:%d Bytes",sizeof(x));
        printf("\nSize of Y is:%d Bytes",sizeof(y));
        printf("\nSize of Z is:%d Bytes",sizeof(z));
}
```

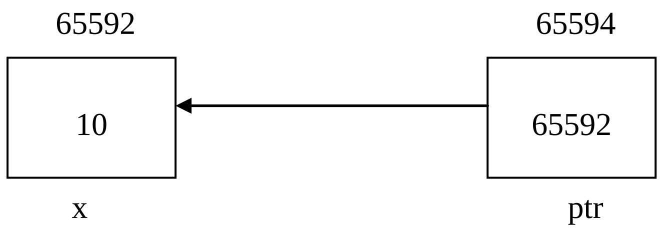
Initializing Pointers: Once a pointer variable has been declared, it can be made to point to a variable using an assignment operator as:

Syntax: **ptrvariable = &ordinaryvariable;**

Example: **ptr = &x;**

Where, x is an ordinary variable.
Here, address of the variable x is stored in ptr. With this, a link is formed from pointer variable to ordinary variable.

```
Example:    int x,*ptr;
            x=10;
            ptr=&x;
```



A pointer variable can also be initialized at the time of its declaration itself.

Example: int x, *ptr = x;

Accessing variable value through pointer: The value of a variable can be accessed through pointer variable using a unary operator '*', usually known as **indirection operator.** The operator refers to "**value at address".**
(*p = value at the address contain in p).

```
/* EXAMPLE PROGRAM TO DEMONSTRATE POINTERS CONCEPT */

main()
{
        int x,*ptr;
        clrscr();
        printf("\nAddress of x is:%u",&x);
        printf("\nAddress of ptr is:%u",&ptr);
        x=100;
        printf("\nx value using x:%d",x);
        ptr=&x;
        printf("\nValue in p:%u",ptr);
        printf("\nx value using ptr:%d",*ptr);
}
```

PASSING PARAMETERS TO FUNCTIONS

Arguments can be passed to a function in two ways. Those are,

1. Call by value (or) Pass by value
2. Call by address (or) Pass by address

1. Call by value: The process of passing actual value of the variable as an argument to a function is known as call by value.

In this mechanism, a copy of the value is passed from calling function to the called function. Since, it's only a copy of the value, if we made any changes inside the function definition; those changes are not effect on the original variable. Entire operations are performed only on temporary variables.

```
/* EXAMPLE PROGRAM FOR CALL BY VALUE */

void change(int,int);
main()
{
        int x=10,y=20;
        clrscr();
        printf("\nBefore Passing to the Function");
        printf("\nx value:%d\ny value:%d",x,y);
        change(x,y);
        printf("\nAfter Passing to the Function");
        printf("\nx value:%d\ny value:%d",x,y);
}
void change(int p,int q)
{
        p=p+5;
        q=q+5;
}
```

2. Call by address: The processing of passing address of the variable as an argument to a function is known as call by address.

In this mechanism, when we pass address of the variable as an argument to a function, the receiving argument at the called function must be a pointer variable. When the pointer variable received the address, it points to the actual variable. So that if we made any changes inside the function definition; those changes are recognized by both the calling function and the called function.

```
/* EXAMPLE PROGRAM FOR CALL BY ADDRESS */

void change(int*,int*);
main()
{
        int x=10,y=20;
        clrscr();
        printf("\nBefore Passing to the Function");
        printf("\nx value:%d\ny value:%d",x,y);
        change(&x,&y);
        printf("\nAfter Passing to the Function");
        printf("\nx value:%d\ny value:%d",x,y);
}
void change(int *p,int *q)
{
        *p=*p+5;
        *q=*q+5;
}
```

/* Write a program to swap the given two variables using functions */

POINTER INCREMENTS AND SCALE FACTOR

1. C language allows adding integers or subtracting integers from pointers. Whenever an integer is added or subtracted from a pointer, the value is in terms of the length of the data type it points to. This length is called as a scale factor.

```
        Example:    int *p, x;
                    p = &x;
                    p = p+2;
                    p = p-1;
```

2. Pointers can also be subtracted one from each other.

```
        Example:    int *p1,*p2, x, y, z;
                    p1 = &x;
                    p2 = &y;
                    z = p1 – p2;
```

3. Pointers can also be compared using the relational operators.

```
        Example:    int *p1,*p2, x, y;
                    p1 = &x;
                    p2 = &y;
                    p1 > p2        /* Valid */
                    p1 == p2       /* Valid */
                    p1 != p2       /* Valid */
```

Note: The following operations are invalid operations on pointers.
Let p1 and p2 are pointer variables.

4. Pointer multiplication and division operations are invalid

```
        Example:    p1 * p2             /* Invalid */
                    p1 / p2             /* Invalid */
```

5. Division with a scale factor is invalid.
   Example: p1 / 3 /* Invalid */

6. Addition of pointers is also invalid.
   Example: p1 + p2 /* Invalid */

```
/* EXAMPLE PROGRAM FOR POINTER INCREMENTS AND SCALE FACTOR */

main()
{
        int *p1,*p2,x,y;
        clrscr();
        p1=&x;
        p2=&y;
        printf("\np1 Address:%u",&p1);          /* 65518 */
        printf("\np1 value:%u",p1);             /* 65522 */
        printf("\np2 Address:%u",&p2);          /* 65520 */
        printf("\np2 value:%u",p2);             /* 65524 */
        x=p2-p1;
        printf("\nx value:%d",x);               /*   1   */
        p1=p1+2;
        printf("\np1 value:%u",p1);             /* 65526 */
        p2=p2-1;
        printf("\np2 value:%u",p2);             /* 65522 */
}
```

VOID POINTER

A void pointer is a special type of pointer that can points to any data type.
   Syntax: void *ptrvariable;

At the time of assigning address of the variable to ptrvariable, type casting must be placed to refer the data item.

```
/* EXAMPLE PROGRAM FOR VOID POINTER */

main()
{
        int a=10;
        double b=3.45678;
        void *p;
        clrscr();
        p=&a;
        printf("\nValue 1:%d",*(int *)p);
        p=&b;
        printf("\nValue 2:%lf",*(double *)p);
}
```

NULL POINTER

Uninitialized pointer variable is initialized to a value in such a way that it is certain not to point to any object or function. A pointer initialized in this manner is called a NULL pointer.

A null pointer is a special type of pointer that cannot points to any where. Null pointer is assigned by using the predefined constant NULL; which is defined by several header files including stdio.h, stdlib.h and alloc.h.

Example: **int *p=NULL;**

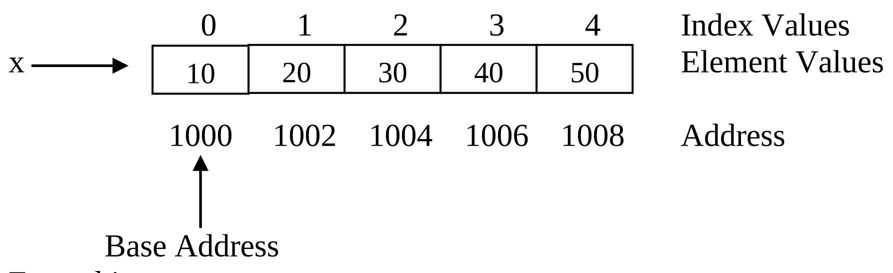
```
/* EXAMPLE PROGRAM FOR NULL POINTER */

#include<stdio.h>
main()
{
        int *p;
        clrscr();
        p=NULL;
        printf("\nValue :%d",*p);
}
```

POINTERS AND ARRAYS

In C language, there is a close relationship between array data type and pointers. An array name in C language is very much like a pointer but there is difference between each other. The pointer is a variable that can appear on the left side of an assignment operator. The array name is a constant and cannot appear on the left side of the assignment operator.

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in successive memory locations. The name of the array is the beginning address (first element index 0) of the array, called the base address. With this, the array name is referred to as an address constant.

Example: **int x[5] = {10,20,30,40,50};**



From this,

x = &x[0] = &x all are referred to the address location 1000.

Array subscripting is defined in terms of pointer arithmetic operations.

Let

int x[3], i=0;

```
&x[0] = x+0            x[0] = *(x+0)
&x[1] = x+1            x[1] = *(x+1)
&x[2] = x+2            x[2] = *(x+2)

i.e., &x[i] = x+i      x[i] = *(x+i)
```

Similarly, for a double dimensional arrays,

```
&x[i][j] = (*(x+i)+j)      x[i][j] = *(*(x+i)+j)
```

```
/* WRITE A PROGRAM TO READ A ONE DIMENSIONAL ARRAY AND PRINT IT USING
POINTERS */

main()
{
        int x[10],i,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        printf("\nEnter %d Numbers:",n);
        for(i=0;i<n;i++)
        scanf("%d",x+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
        printf("%5d",*(x+i));
}
```

(OR)

```
main()
{
        int x[10],i,n,*p;
        clrscr();
        p=x;
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        printf("\nEnter %d Numbers:",n);
        for(i=0;i<n;i++)
        scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
        printf("%5d",*(p+i));
}
```

```
/* WRITE A PROGRAM TO READ LIST OF FLOATING POINT NUMBERS AND PRINT SUM
OF THE ARRAY ELEMENTS USING POINTERS */

main()
{
        float x[10],*p,sum;
        int i,n;
        clrscr();
        p=x;
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
```

```
	printf("\nEnter %d Numbers:",n);
	for(i=0;i<n;i++)
	scanf("%f",p+i);
	sum=0;
	for(i=0;i<n;i++)
	sum=sum+*(p+i);
	printf("\nTotal :%.2f",sum);
}
```

/* WRITE A PROGRAM TO READ A DOUBLE DIMENSIONAL ARRAY AND PRINT IT USING POINTERS */

```
main()
{
	int x[10][10],i,j,m,n;
	clrscr();
	printf("\nEnter how many Rows:");
	scanf("%d",&m);
	printf("\nEnter how many Columns:");
	scanf("%d",&n);
	printf("\nEnter Array Elements:");
	for(i=0;i<m;i++)
	{
		for(j=0;j<n;j++)
		scanf("%d",*(x+i)+j);
	}
	printf("\nArray Elements Are:");
	for(i=0;i<m;i++)
	{
		printf("\n");
		for(j=0;j<n;j++)
		printf("%5d",*(*(x+i)+j));
	}
}
```

(OR)

```
main()
{
	int x[10][10],i,j,m,n,(*p)[10];
	clrscr();
	printf("\nEnter how many Rows:");
	scanf("%d",&m);
	printf("\nEnter how many Columns:");
	scanf("%d",&n);
	p=x;
	printf("\nEnter Array Elements:");
	for(i=0;i<m;i++)
	{
		for(j=0;j<n;j++)
		scanf("%d",*(p+i)+j);
	}
```

```
        printf("\nArray Elements Are:");
        for(i=0;i<m;i++)
        {
                printf("\n");
                for(j=0;j<n;j++)
                printf("%5d",*(*(p+i)+j));
        }
}
```

ARRAY OF POINTERS

Array of pointers are used to define more than one pointer variable of the same type. The general format of declaring array of pointers is:

Syntax: datatype *ptrvariable[size];

Example: float *ptr[10];

```
/* EXAMPLE PROGAM 1 FOR ARRAY OF POINTERS */

main()
{
        int i,*p[5],x,y,z;
        clrscr();
        x=10;
        y=20;
        z=30;
        p[0]=&x;
        p[1]=&y;
        p[2]=&z;
        printf("\nElements Are:");
        for(i=0;i<3;i++)
        printf("%5d",*p[i]);
}

/* EXAMPLE PROGRAM 2 FOR ARRAY OF POINTERS */

main()
{
        int i,j,*p[5];
        int x[]={1,2,3},y[]={4,5,6},z[]={7,8,9};
        clrscr();
        p[0]=x;
        p[1]=y;
        p[2]=z;
        printf("\nElements Are:");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
                printf("%5d",*(*(p+i)+j));
        }
}
```

DYNAMIC MEMORY ALLOCATION

The memory allocation may be classified as static memory allocation and dynamic memory allocation.

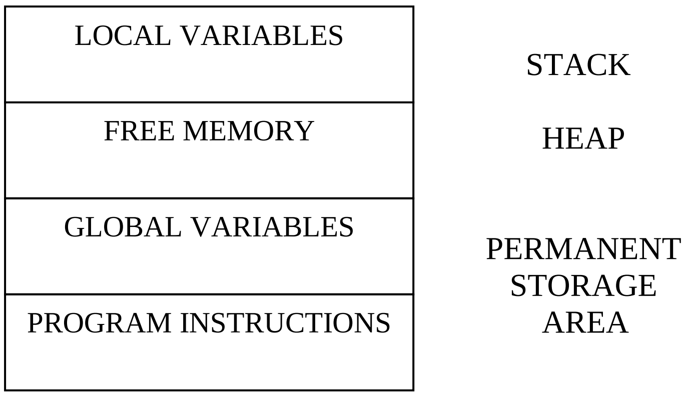
Static memory allocation: Memory for the variables is created at the time of compilation is known as static memory.

Dynamic memory allocation: Memory for the variables is allocated at the time of execution of the program is called dynamic memory. The following functions are used for dynamic memory allocation which are defined in **stdlib.h** and **alloc.h** header files.

1. malloc() 2. calloc() 3. realloc() 4. free()

malloc(), calloc() and realloc() are memory allocation functions and free() is a memory releasing function.

Memory allocation process



The program instructions, global and static variables are stored in a region known as **permanent storage area.** Local variables are stored in another region called **stack.** The memory spaced located between stack and permanent storage area is available for dynamic memory allocation during execution of the program. This free memory region is called as **heap.**

1. malloc(): malloc() function is used to allocate memory for the variables at run time. The malloc() function reserves a single block of memory with the specified size and returns a pointer of type void. With this, we can assign it to any type of pointer. The general form of malloc() function is:

Syntax: **prtvariable = (casttype*)malloc(size);**

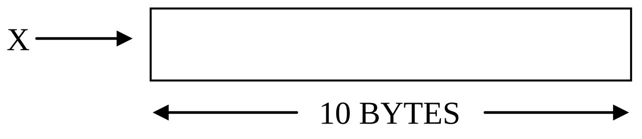
Where,

ptrvariable is a pointer variable of type casttype.
size represents number of bytes of memory to be allocated.

Example: int *X;

X = (int*)malloc(10); (or) X = (int*)malloc(5*sizeof(int));

For this, memory allocation will be:



/* EXAMPLE PROGRAM FOR MALLOC() FUCNTION */

```
main()
{
        int *p,i,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)malloc(n*sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
        scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
        printf("%3d",*(p+i));
}
```

<u>2. calloc():</u> calloc() function is another memory allocation function used for dynamic memory allocation for storing derived data types such as arrays and structures etc.,

malloc() function allocates a single block of storage space, whereas calloc() function allocates multiple blocks of storage space with each of same size and all locations are initialized with ‘0’. The general form of calloc() function is:

Syntax: ptrvariable = (casttype*)calloc(n,elesize);

Where,

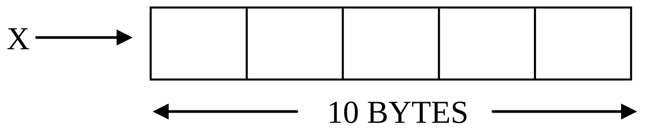
ptrvariable is a pointer variable of type casttype.
n represents number of blocks.
elesize represents block size.

All blocks are initialized with ‘0’ and a pointer of the first block is returned. If there is not enough space to allocate, then it returns a NULL pointer.

Example: int *X;

X = (int*)calloc(5,sizeof(int));

For this, memory allocation will be:



```
/* EXAMPLE PROGRAM FOR CALLOC() FUNCTION */

main()
{
        int *p,i,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)calloc(n,sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
        scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
        printf("%3d",*(p+i));
}
```

```
/* EXAMPLE PROGRAM FOR MALLOC() AND CALLOC() FUNCTIONS */

main()
{
        int *p,*q,i,m,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&m);
        p=(int*)malloc(m*sizeof(int));
        printf("\nArray Elements Are:");
        for(i=0;i<m;i++)
        printf("%7d",*(p+i));
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        q=(int*)calloc(n,sizeof(int));
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
        printf("%7d",*(q+i));
}
```

3. realloc(): Suppose previously allocated dynamic memory is not sufficient (or) memory is much larger than the requirement, in both cases some memory changes are required. Memory changes can be done by using a library function called realloc() function.

realloc() function provides the altering the size of the memory allocation and the process is called reallocation of memory. The general form of realloc() function is:

Syntax: **ptrvariable = (casttype*)realloc(ptrvariable,newsize);**

Where,

ptrvariable is a pointer variable of type casttype.

Here,

This function allocates a new memory space of the specified newsize and returns a pointer variable that represent first byte of the new memory block. The newsize may be larger or smaller than the previous size.

Note:

I. The new memory block may or may not be begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and moves the contents of the old block into the new block.
II. If the function is unsuccessful to allocate the memory space, it returns a NULL pointer and the original block is lost.

```
/* EXAMPLE PROGRAM FOR REALLOC() FUNCTION */

main()
{
        int *p,i,n;
        clrscr();
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)malloc(n*sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
        scanf("%d",p+i);
        p=(int*)realloc(p,(n+2)*sizeof(int));
        printf("\nEnter %d Elements:",n+2);
        for(i=0;i<n+2;i++)
        scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n+2;i++)
        printf("%3d",*(p+i));
}
```

4. free(): The memory allocation done by malloc(), calloc() and realloc() functions at run time are released by invoking the function free() by the user explicitly. The releasing of storage space becomes very important when the storage is space is limited. The general form of free() function is:

Syntax: **free(ptrvariable);**

Example: **free(ptr);**

POINTERS AND STRINGS

A string is an array of characters that terminated with a null character ‘\0’. As similar to arrays, we can also user pointers to access the individual characters in the given string.

```
/* EXAMPLE PROGRAM FOR PONTER TO STRING */

main()
{
	char s[]="KAVALI",*p;
	int i;
	clrscr();
	p=s;
	printf("\nString Elements Are:");
	for(i=0;*(p+i)!='\0';i++)
	printf("%3c",p[i]);
}
```

A string constant is like an array name by itself, it is treated by the compiler as a pointer. Its value is the base address of the string.

```
/* EXAMPLE PROGRAM */

main()
{
	char s[]="KAVALI",*p="NELLORE";
	int i;
	clrscr();
	printf("\nResult 1:");
	puts(s);
	printf("\nResult 2:");
	puts(p);
}
```

```
/* WRITE A PROGRAM TO FIND THE LENGTH OF THE GIVEN STRING USING POINTERS */

main()
{
	char *p="pbr vits";
	int i;
	clrscr();
	for(i=0;*(p+i)!='\0';i++);
	printf("\nLength is:%d",i);
}
```

```
/* WRITE A PROGRAM TO COPY THE CONTENTS OF ONE STRING INTO ANOTHER USING
POINTERS */

main()
{
	char *p="kavali",*q;
	clrscr();
	q=p;
	printf("\nOriginal String is:");
	puts(p);
	printf("\nCopied String is:");
	puts(q);
}
```

/* WRITE A PROGRAM TO PRINT A STRING IN ALPHABETICAL ORDER USING POINTERS */

```
main()
{
        char *p="kavali",temp;
        int i,j,len;
        clrscr();
        printf("\nOriginal String:");
        puts(p);
        for(len=0;*(p+len)!='\0';len++);
        for(i=0;i<=len-2;i++)
        {
                for(j=i+1;j<=len-1;j++)
                {
                        if(*(p+i)>*(p+j))
                        {
                                temp=*(p+i);
                                *(p+i)=*(p+j);
                                *(p+j)=temp;
                        }
                }
        }
        printf("\nString Characters in Alphabetical Order is:");
        puts(p);
}
```

RETURNING MORE THAN ONE VALUE FROM A FUNCTION

Functions usually return only one value and when arguments are passed by value, the called function cannot alter the original argument.

Pointers allow the user to return more than one value by allowing the arguments to be passed by address, which allows the function to alter the values that pointed to and thus return more than one value from the function.

/* WRITE A PROGRAM TO PASS ONE DIMENSIONAL ARRAY AS AN ARGUMENT TO A FUNCTION AND RETURN MAXIMUM NUMBER FROM THE ARRAY */

```
int Max(int*,int);
main()
{
        int x[]={11,33,66,99,22},k;
        clrscr();
        k=Max(x,5);
        printf("\nMaximum Number:%d",k);
}

int Max(int *p,int n)
{
        int a,i;
        a=*p;
        for(i=0;i<n;i++)
        {
                if(*(p+i)>a)
                  a=*(p+i);
        }
        return a;
}
```

/* WRITE A PROGRAM TO PASS ONE DIMENSIONAL AS AN ARGUMENT TO A FUNCTION AND SORT USING POINTERS */

```
int* sort(int*,int);
main()
{
        int x[]={77,11,33,88,55},*k,i;
        clrscr();
        printf("\nBefore Sorting Elements Are:");
        for(i=0;i<5;i++)
        printf("%4d",x[i]);
        k=sort(x,5);
        printf("\nAfter Sorting Elements Are:");
        for(i=0;i<5;i++)
        printf("%4d",*(k+i));
}
int* sort(int *p,int n)
{
        int i,j,temp;
        for(i=0;i<=n-2;i++)
        {
                for(j=i+1;j<=n-1;j++)
                {
                        if(*(p+i)>*(p+j))
                        {
                                temp=*(p+i);
                                *(p+i)=*(p+j);
                                *(p+j)=temp;
                        }
                }
        }
        return p;
}
```

/* WRITE A PROGRAM TO RETURN REVERSE THE PASSED STRING */

```
char* rev(char*);
main()
{
        char *s="pbr vits",*p;
        clrscr();
        printf("\nOriginal String:");
        puts(s);
        p=rev(s);
        printf("\nReverse String:");
        puts(p);
}
```

```
char* rev(char *k)
{
        strrev(k);
        return k;
}
```

Differences between arrays and pointers:

ARRAY	**POINTER**
1. For an array, compiler allocates memory space automatically and used by the array.	1. For a pointer, it is explicitly assigned to point to an allocated space.
2. It can not be resized.	2. It can be resized using realloc() function.
3. It can not be reassigned.	3. It can be reassigned.
4. sizeof(arrayname) gives the number of bytes of the memory occupied by the array.	4. sizeof(pointervaraible) returns only 2 bytes.

POINTER TO POINTER

A pointer is a variable that contains address of another variable. The pointer variable containing address of another pointer variable is known as pointer-to-pointer. For this, add an asterisk for each level of reference.

Example:

```
int x = 10;
int *p;
p=&a;

int **q;
q=&p;
```

```
/* EXAMPLE PROGRAM FOR POINTER TO POINTER VARIABLE */

main()
{
        int a=5,*p,**q;
        clrscr();
        p=&a;
        q=&p;
        printf("\nValue with p is:%d",*p);
        printf("\nValue with q is:%d",**q);
}
```

```
/* EXAMPLE PROGRAM THAT SHOWS TO READ THE VALUE OF A VARIABLE USING
POINTER-TO-POINTER */

main()
{
        int x,*p,**q,***r;
        clrscr();
        p=&x;
        q=&p;
        r=&q;
        printf("\nEnter a value:");
        scanf("%d",&x);
        printf("\nValue with x is:%d",x);
        printf("\nEnter another value with p:");
        scanf("%d",p);
        printf("\nValue with p is:%d",*p);
        printf("\nEnter another value with q:");
        scanf("%d",*q);
        printf("\nValue with q is:%d",**q);
        printf("\nEnter another value with r:");
        scanf("%d",**r);
        printf("\nValue with r is:%d",***r);
}
```

```
/* EXAMPLE PROGRAM TO IMPLEMENT DOUBLE DIMENSIONAL ARRAY USING
POINTERS */

#define row 3
#define col 3
main()
{
        int **a,i,j;
        clrscr();
        a=(int**)malloc(row*sizeof(int*));
        for(i=0;i<row;i++)
        a[i]=(int*)malloc(col*sizeof(int));
        printf("\nEnter Array Elements:");
        for(i=0;i<row;i++)
        {
           for(j=0;j<col;j++)
           scanf("%d",&a[i][j]);
        }
        printf("\nArray Elements Are:");
     for(i=0;i<row;i++)
        {
           printf("\n");
           for(j=0;j<col;j++)
           printf("%4d",a[i][j]);
        }
        free(a);
}
```

POINTER TO FUNCTION

In C language every variable has an address except register variables. Similarly, C functions also have an address. We can also invoke the function by using its address.

```
/* EXAMPLE PROGRAM TO DISPLAY THE ADDRESS OF LIBRARY FUNCTIONS */

#include<stdio.h>
main()
{
        clrscr();
        printf("\nAddress of printf function is:%u",printf);
        printf("\nAddress of scanf function is:%u",scanf);
        printf("\nAddress of clrscr function is:%u",clrscr);
}
```

```
/* EXAMPLE PROGRAM TO DISPLAY THE ADDRESS OF USER-DEFINED FUNCTIONS */

void show();
main()
{
        clrscr();
        show();
        printf("\nAddress of the Function:%u",show);
}

void show()
{
        printf("\nFunction Called");
}
```

The address of the function can also be assigned to a pointer variable. For this, the general form of declaring the pointer variable is:

Syntax: datatype (*ptrvariable)( );

Here,

ptrvariable is name of the pointer variable and the parenthesis indicates that it is a pointer variable to function.

```
/* EXAMPLE PROGRAM TO CALL A FUNCTION USING POINTER */

int show();
main()
{
        int(*p)();
        clrscr();
        p=show;   /* ASSIGN ADDRESS OF SHOW() TO P */
        (*p)();   /* FUNCTION CALL USING POINTER */
        printf("\nAddress Is:%u",show);
}
int show()
{
        printf("\nFunction Called");
}
```

<u>USES OF POINTERS</u>

I. A pointer enables us to access a variable that is defined outside the function.
II. Pointers reduce length and complexity of the program.
III. They increase execution speed of the program.
IV. They provide to allocate memory dynamically. So, that to save a lot of memory space area.
V. They provide to return more than one value from a function.

***

UNIT – V

Structure and Union – Introduction, Features of Structures, Declaration and Initialization of Structures, Structure within Structure, Array of Structures, Pointer to Structure, Structure and Functions, typedef, Bit Fields, Enumerated Data Type, Union, Union of Structures.

STRUCTURES

STRUCTURE DECLARATION AND DEFINITION:

"Structure is a collection of heterogeneous (different) data elements that can be grouped together under a common name".

A structure definition is specified by the keyword "**struct**". The keyword is followed by a user defined name surrounded by pair of braces, which describes the members of the structure. The general format of a structure declaration is:

Syntax:
```
struct Tag
{
        Datatype Member1;
        Datatype Member2;
                - - -
                - - -
        Datatype Membern;
};
```

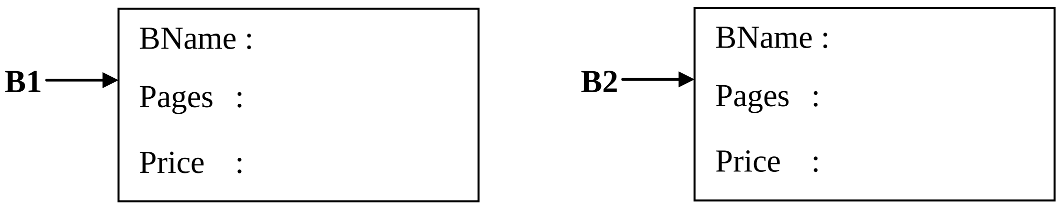
- struct is a keyword used to define structure declaration.
- Tag is name of the structure that follows same rules as a valid identifier.
- Members declared inside the structure declaration are called structure elements (or) structure members.
- Structure declaration must be ended with a semicolon.
- Structure is an user-defined data type.

Example:
```
struct Book
{
        char BName[50];
        int Pages;
        float Price;
};
```

Declaration of the structure does not reserve any storage space. Memory is allocated only at the time of defining a structure variable. The general format of defining a structure variable is:

Syntax: **struct Tag varname1, varname2, - - - - - , varnamep;**
Example: **struct Book B1,B2;**

Now, the compiler allocates memory for B1 and B2 as:



i.e., Memory is allocated individually for each structure variable as well as individual memory area for each member of the structure.

Accessing members of the structure:

'.' Dot operator is used to access members of the structure with its structure variable. Here dot operator is also known as **member operator** (or) **period operator.** It forms link between structure member and structure variable. The general format of accessing a structure member with structure variable is:

Syntax: **structurevariable.member;**
Example: **B1.Pages;**

```
/* EXAMPLE PROGRAM FOR PRINTING BOOK DETAILS USING STRUCTURE */

struct Book
{
        char BName[50];
        int Pages;
        float Price;
};

main()
{
        struct Book B1;
        clrscr();
        printf("\nEnter Titile of the Book:");
        gets(B1.BName);
        printf("\nEnter Number of Pages:");
        scanf("%d",&B1.Pages);
        printf("\nEnter Cost of the Book:");
        scanf("%f",&B1.Price);
        printf("\nBOOK TITLE:%s",B1.BName);
        printf("\nNUMBER OF PAGES:%d",B1.Pages);
        printf("\nBOOK COST:%.2f RS",B1.Price);
}
```

1. Write a program to print an employee database using structure.
2. Write a program to print student database using structure with the members name, branch and 3 subject marks. Print average marks obtained by the student.
3. Write a program to print the product details using structure.

Note:

Structure declaration and definition can also be combined into a single statement. For this, use the syntax as:

```
Syntax:     struct Tag
            {
                    Datatype Member1;
                    Datatype Member2;
                            - - -
                            - - -
                    Datatype Membern;
            }
            varname1, varname2, - - - - - , varnamep;
```

```
/* EXAMPLE PROGRAM FOR PRINTING CURRENT DATE */

struct Date
{
        int Day,Month,Year;
};

main()
{
        struct Date x;
        clrscr();
        printf("\nEnter Current Day:");
        scanf("%d",&x.Day);
        printf("\nEnter Current Month:");
        scanf("%d",&x.Month);
        printf("\nEnter Current Year:");
        scanf("%d",&x.Year);
        printf("\nTODAY DATE:%d/%d/%d",x.Day,x.Month,x.Year);
}
```

INITIALIZING A STRUCTURE

A structure can be initialized with a list of values at the time of defining structure variable. But, individual member initialization inside the structure declaration is not possible. The general format of initializing a structure is:

```
Syntax:     struct Tag varname = { List of Values };

Example:    struct Date x ={7,11,2010};

            struct   Book b = {“Let Us C”,350,125.75};
```

- Initialization values for a structure must be enclosed within a pair of braces.
- The constants to be assigned to the members of the structure must be in the same order as that in which the members are specified.
- If some of the structure members are not initialized, then C compiler will automatically initialize with zero.

/* WRITE A PROGRAM TO COPY THE CONTENTS OF ONE STRUCTURE INTO ANOTHER STRUCTURE */

```
struct product
{
        int pid;
        char pname[10];
        float price;
};
main()
{
        struct product s1={111,"soap",25.00},s2;
        clrscr();
        printf("\nOriginal structure is:");
        printf("\n%d\t%s\t%.2f",s1.pid,s1.pname,s1.price);
        s2=s1;
        printf("\nCopied structure is:");
        printf("\n%d\t%s\t%.2f",s2.pid,s2.pname,s2.price);
}
```

/* WRITE A PROGRAM TO COMPARE THE GIVEN TWO STRUCTURES */

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

struct product
{
        int pid;
        char pname[10];
        float price;
};
main()
{
        struct product s1,s2;
        clrscr();
        printf("\nEnter Product 1 id:");
        scanf("%d",&s1.pid);
        fflush(stdin);
        printf("\nEnter Prouduct 1 Name:");
        gets(s1.pname);
        printf("\nEnter Product 1 Cost:");
        scanf("%f",&s1.price);
        printf("\nEnter Product 2 id:");
        scanf("%d",&s2.pid);
        fflush(stdin);
        printf("\nEnter Prouduct 2 Name:");
        gets(s2.pname);
        printf("\nEnter Product 2 Cost:");
        scanf("%f",&s2.price);
        if((s1.pid==s2.pid)&&((strcmp(s1.pname,s2.pname))==0)&&(s1.price==s2.price))
          printf("\nBOTH ARE EQUAL");
        else
          printf("\nBOTH ARE DIFFERENT");
}
```

ARRAY OF STRUCTURES

A similar type of structure variables placed with a common variable name is called an array of structures. Array of structure concept is used to define more than one structure variable of the same type.

```
/* EXAMPLE PROGRAM TO READ A LIST OF STUDENTS INFORMATION AND PRINT
THEM IN ASCENDING ORDER OF THEIR AVERAGE MARKS USING ARRAY OF
STRUCTURES */

#include<stdio.h>
struct student
{
        char name[50],branch[5];
        int sub1,sub2,sub3;
        float avg;
}s[10];
main()
{
        int i,j,n;
        struct student temp;
        clrscr();
        printf("\nEnter how many students:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
          fflush(stdin);
          printf("\nEnter student %d Name:",i);
          gets(s[i].name);
          fflush(stdin);
          printf("\nEnter student %d Branch:",i);
          gets(s[i].branch);
          printf("\nEnter 3 subject Marks:");
          scanf("%d%d%d",&s[i].sub1,&s[i].sub2,&s[i].sub3);
        }
        for(i=1;i<=n;i++)
        s[i].avg=(s[i].sub1+s[i].sub2+s[i].sub3)/3.0;

        for(i=1;i<=n-1;i++)
        {
                for(j=i+1;j<=n;j++)
                {
                        if(s[i].avg>s[j].avg)
                        {
                                temp=s[i];
                                s[i]=s[j];
                                s[j]=temp;
                        }
                }
        }
        printf("\nStudent Details As Per Their Averge Marks Are:");
        for(i=1;i<=n;i++)
        printf("\n%s\t%s\t%.2f",s[i].name,s[i].branch,s[i].avg);
}
```

Initializing Array of Structures:

Array of structures can be initialized at the time of its declaration by a list of values placed within pair of braces.

Example:

```
struct student
{
        char name[50],branch[5];
        int sub1,sub2,sub3;
}
s[10] = {  { “satish kumar”,”cse”,67,89,90} ,
             { “praveen”, “it”,45,78,89}
        };
```

ARRAYS WITHIN STRUCTURES

C permits the use of array as structure members. i.e., an array can also be placed as a member of a structure.

```
/* EXAMPLE PROGRAM FOR ARRAYS WITHIN STRUCTURES */

#include<stdio.h>
struct student
{
        char name[50];
        int sub[3],total;
}s[10];

main()
{
        int i,j,n;
        clrscr();
        printf("\nEnter how many students:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
          fflush(stdin);
          printf("\nEnter student %d Name:",i);
          gets(s[i].name);
          printf("\nEnter 3 subject Marks:");
          for(j=0;j<3;j++)
          scanf("%d",&s[i].sub[j]);
        }
        for(i=1;i<=n;i++)
        {
                s[i].total=0;
                for(j=0;j<3;j++)
                s[i].total=s[i].total+s[i].sub[j];
        }
        printf("\nStudent Details Are:");
        for(i=1;i<=n;i++)
        printf("\n%s\t%d",s[i].name,s[i].total);
}
```

NESTED STRUCTURES (OR) STRUCTURE WITHIN STRUCTURE

When a structure is declared as the member of another structure; it is termed as nested structures (or) structure within structure. The general format of nested structures is:

Syntax:
```
struct Outer
{
        - - -
        - - -
        struct Inner
        {
                - - -
                - - -
        }Variable;
        - - -
        - - -
};
```

- In nested structures, the dot operator in conjunction with the structure variables are used to access the members of the innermost as well as the outermost structures. The general format of accessing inner structure member is:

Syntax: **OuterStructureVariable.InnerStructureVariable.InnerMember;**

```
/* EXAMPLE PROGRAM FOR NESTED STRUCTURES */
#include<stdio.h>

struct student
{
        char name[50];
        struct doj
        {
                int d,m,y;
        }join;
        char branch[5];
}s;
```

```
main()
{
        clrscr();
        printf("\nEnter student Name:");
        gets(s.name);
        printf("\nEnter Date of Joining:");
        scanf("%d%d%d",&s.join.d,&s.join.m,&s.join.y);
        fflush(stdin);
        printf("\nEnter Branch of the student:");
        gets(s.branch);
        printf("\n\nStudent Details Are:");
        printf("\nSTUDENT NAME:%s",s.name);
        printf("\nBRANCH      :%s",s.branch);
        printf("\nJOIN DATE   :%d/%d/%d",s.join.d,s.join.m,s.join.y);
}
```

```
/* EXAMPLE PROGRAM FOR INITIALIZING NESTED STRUCTURES */
#include<stdio.h>
struct student
{
        char name[50];
        struct doj
        {
                int d,m,y;
        }join;
        char branch[5];
}s={"ravi kumar",10,5,2010,"ece"};

main()
{
        int i,j,n;
        clrscr();
          printf("\n\nStudent Details Are:");
          printf("\nSTUDENT NAME:%s",s.name);
          printf("\nBRANCH      :%s",s.branch);
          printf("\nJOIN DATE   :%d/%d/%d",s.join.d,s.join.m,s.join.y);
}
```

<u>STRUCTURES AND POINTERS</u>

A structure pointer variable is created as similar to the creation of a structure variable. A pointer to a structure is not itself a structure, but it's a variable that holds the address of the structure variable. This type of pointer variable also occupies only 2 bytes of memory. The general format of structure pointer variable declaration is:

Syntax: struct Tag *PtrVariable;

Here, PtrvVariable can be assigned to any other structure of the same type, and can be used to access the members of its structure.

```
Example:    struct Account
            {
                    int Acno;
                    float AcBalance;
            }S, *P;
            P = &S;
```

'->' operator symbol is used to access members of the structure with its pointer variable. '->' operator symbol is formed with the combination of a minus sign and a greater than symbol.

Syntax: **PtrVariable -> Member;**

The same representation can also be placed with dot operator is as:

Syntax: **(*PtrVariable) . Member;**

```
/* EXAMPLE PROGRAM FOR STRUCTURE POINTER VARIABLE */

struct Account
{
        int Acno;
        float AcBalance;
}S={111,35000.75}, *P;

main()
{
        clrscr();
        P=&S;
        printf("\nACCOUNT INFORMATION IS:");
        printf("\n%d\t%.2f",P->Acno,P->AcBalance);
}
```

```
/* EXAMPLE PROGRAM FOR DIFFERENT WAYS OF ACCESSING */

struct demo
{
        int x;
        float y;
        char z;
};
main()
{
        struct demo s={10,345.72,'K'},*p;
        clrscr();
        printf("\nWith Variable and Dot Operator:");
        printf("\n%d\t%.2f\t%c",s.x,s.y,s.z);
        p=&s;
        printf("\nWith Pointer Variable and Arrow Opeator:");
        printf("\n%d\t%.2f\t%c",p->x,p->y,p->z);
        printf("\nWith Pointer Variable and Dot Opeator:");
        printf("\n%d\t%.2f\t%c",(*p).x,(*p).y,(*p).z);
}
```

STRUCTURES AND FUNCTIONS

A structure can also be passed as an argument to a function in three ways. Those are:

1. Passing individual member of the structure
2. Passing entire structure
3. Passing address of the structure

1. The first method is to pass each member of the structure individually as an argument to the called function. Then at the called function they are treated as ordinary variables.

```
/* EXAMPLE PROGRAM TO PASS INDIVIDUAL MEMBER OF THE STRUCTURE AS AN
ARGUMENT */

struct cricket
{
        char name[15];
        int matches;
        float avg;
}s={"sachin",400,56.78};
void change(int);
main()
{
        clrscr();
        printf("\nBefore Passing:");
        printf("\n%s\t%d\t%.2f",s.name,s.matches,s.avg);
        change(s.matches);
        printf("\nAfter Passing:");
        printf("\n%s\t%d\t%.2f",s.name,s.matches,s.avg);
}
void change(int x)
{
        x=410;
        printf("\nWith in the Function:%d",x);
}
```

2. The second method is to pass entire structure as an argument to a function. At this stage, the receiving argument at the called function must be a structure of the same type. Here, a copy of the entire structure is copied into the formal structure. Since, the function is working on a copy of the structure; any changes to structure members within the function are not reflected in the original structure.

```
/* EXAMPLE PROGRAM TO PASS ENTIRE STRUCTURE AS AN ARGUMENT */

#include<string.h>
struct cricket
{
        char name[15];
        int matches;
        float avg;
}s={"sachin",400,56.78};
```

```
void change(struct cricket);
main()
{
        clrscr();
        printf("\nBefore Passing:");
        printf("\n%s\t%d\t%.2f",s.name,s.matches,s.avg);
        change(s);
        printf("\nAfter Passing:");
        printf("\n%s\t%d\t%.2f",s.name,s.matches,s.avg);
}
void change(struct cricket x)
{
        strcpy(x.name,"sehwag");
        x.matches=250;
        x.avg=50.23;
        printf("\nWith in the Function:");
        printf("\n%s\t%d\t%.2f",x.name,x.matches,x.avg);
}
```

3. The third method is to pass address of the structure as an argument to a function. When address of the structure is passed, the receiving argument at the called function should be structure pointer variable of the same structure type. With this the pointer points to the original structure.

Now, if made any changes inside the function on structure, those changes are recognized by both the calling function and called function. Since, entire effects are changed on the original structure.

```
/* EXAMPLE PROGRAM TO PASS ADDRESS OF THE STRUCTURE AS AN ARGUMENT */

#include<string.h>
struct cricket
{
        char name[15];
        int matches;
        float avg;
}s={"sachin",400,56.78};
void change(struct cricket *);
main()
{
        clrscr();
        printf("\nBefore Passing:");
        printf("\n%s\t%d\t%.2f",s.name,s.matches,s.avg);
        change(&s);
        printf("\nAfter Passing:");
        printf("\n%s\t%d\t%.2f",s.name,s.matches,s.avg);
}
void change(struct cricket *k)
{
        strcpy(k->name,"sehwag");
        k->matches=250;
        k->avg=50.23;
        printf("\nWith in the Function:");
        printf("\n%s\t%d\t%.2f",k->name,k->matches,k->avg);
}
```

TYPEDEF

The typedef keyword allows the user to create a new data type name for the existing data type. The general format of the declaration statement using the typedef keyword is:

Syntax: **typedef ExistingDataType NewType;**

Where,

ExistingDataType may be either a primitive data type of user-defined data type.

Note: typedef declaration does not create any new data type. It just adds a new name for the existing data type.

Example:

1. **typedef int Integer;**
   Integer x,y,z;
2. **typedef char Gender;**
   Gender x='F', y='M';
3. **typedef char String[10];**
   String Name;

typedef keyword is very useful in the case of user-defined data types like structure.

```
/* EXAMPLE PROGAM OF TYPEDEF WITH STRUCTURE */

typedef struct point
{
        int x,y;
}dot;
main()
{
        dot p;
        clrscr();
        p.x=11;
        p.y=46;
        printf("\nResult is:%d    %d",p.x,p.y);
}
```

Another advantage of typedef statement with the structure is that structure tag is optional.

```
/* EXAMPLE PROGRAM */

typedef struct
{
        float real,imag;
}complex;
main()
{
        complex k;
        clrscr();
        printf("\nEnter Real Part:");
        scanf("%f",&k.real);
        printf("\nImaginary Part:");
        scanf("%f",&k.imag);
        printf("\nComplex Number is:%.2f+%.2fi",k.real,k.imag);
}
```

BIT FIELDS

In C language, bits are manipulated in two ways. Those are:

1. Implementation of bit-wise operators
2. Declaration of bit fields using a structure.

Bit fields utilization with the structure allows using the bit fields to hold data items and hereby pack several data items in a single word of memory. For bit fields memory is allocated in adjacent bits. The name and size of bit fields are defined using structure. The general format of bit field definition is:

Syntax:
```
struct tag
{
        unsigned int member1 : bitwidth1;
        unsigned int member2 : bitwidth2;
                - - - - - -
                - - - - - -
        unsigned int membern : bitwidthn;
} variable;
```

Here,

Each bitwidth specifies number of bits used for the members.
Each member is followed by a colon.

The main advantage of bit fields is to save some memory as against storing variables. By this technique, the exact number of bits required by the bit field is specified. This way a whole word is not required to hold the field. This helps in packing a number of bit fields into single word of memory.

Note: We cannot take address of a bit field variable. i.e., we cannot use scanf() function to read values into bit fields.

```
/* EXAMPLE PROGRAM FOR BIT FIELDS */

struct test
{
        unsigned int x:4;
        unsigned int y:1;
        unsigned int z:7;
}val;
```

```
main()
{
        val.x=10;
        val.y=1;
        val.z=64;
        clrscr();
        printf("\nResult Values Are:");
        printf("\n%d    %d       %d",val.x,val.y,val.z);
        printf("\nSize of Val variable is:%d",sizeof(val));
}
```

Bit fields are also used to force a gap in between the data member fields. In the following structure program, we provide 3 bits gap in between fields.

```
/* EXAMPLE PROGRAM TO FORCE BIT GAP IN BETWEEN BIT FIELDS */

struct test
{
        unsigned x:2;
                 :3;
        unsigned y:1;
}val;
main()
{
        clrscr();
        printf("\nSize of Val variable is:%d Bytes",sizeof(val));
}
```

ENUMERATION TYPES

Enumeration data types are data items whose values may be any member of a symbolically declared set of values. These members are ranging over a set of named constants called enumerators. For this, we use **enum** keyword. Enumeration data type is also a user-defined data type. The general format of creating enum data type is:

Syntax: **enum TagName {Member1, Member2, - - - - - , Membern};**

Where

TagName specifies name of the data type as valid identifier.

enum TagName specifies the user-defined type.

Member1, Member2, - - -, Membern are called enumeration constants (or) enumeration list. Each enumeration member is assigned with some integer value. By default, Member1 is given the value 0, Member2 is given the value 1 and so on.

Example: **enum Days{Mon, Tue, Wed, Thu, Fri, Sat, Sun};**

Once the enumeration list has been defined, corresponding enumeration variables can be created as:

Syntax: **enum TagName Variable1, Variable2, - - - -, Variablep;**
Example: **enum Days start,end;**

```
/* EXAMPLE PROGRAM TO USE ENUMERATED DATA TYPE */

enum Days{Mon, Tue, Wed, Thu, Fri, Sat, Sun};
main()
{
        enum Days start,end;
        clrscr();
        start=Tue;
        end=Sat;
        printf("\n%d     %d",start,end);
        start=65;
        printf("\n%d     %d",start,end);
}
```

It is also to associate numbers other than the sequence starting at zero with the names in the enum data type by including a specific initialization in the variable name list.

```
/* EXAMPLE PROGRAM FOR ENUM WITH DIFFERENT INITIALIZATIONS */

enum coins{p1,p2=45,p3,p4=80};
main()
{
        clrscr();
        printf("\nValue of p1:%d",p1);
        printf("\nValue of p2:%d",p2);
        printf("\nValue of p3:%d",p3);
        printf("\nValue of p4:%d",p4);
}
```

Enumeration provides a convenient way to associate constant values with names as alternative to #define statements.

Example: **#define TRUE 1**
#define FALSE 0

Example: **enum flag{FALSE, TRUE};**

```
/* LP: WRITE A PROGRAM THAT USES FUNCTIONS TO PERFORM THE FOLLOWING
OPERATIONS. I) READING A COMPLEX NUMBER 2) WRITING A COMPLEX NUMBER
3) ADDITION, SUBTRACTION, MULTIPLICATION AND DIVISION OF COMPLEX
NUMBERS */

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

typedef struct
{
  float x,y;
}complex;
```

```
void display(complex);
complex add(complex,complex);
complex sub(complex,complex);
complex mul(complex,complex);
complex div(complex,complex);
complex c1,c2;

main()
{
  complex c3;
  int ch;
  clrscr();
  printf("\nEnter first complex number:");
  printf("\nEnter Real Part:");
  scanf("%f",&c1.x);
  printf("\nEnter Imaginary Part:");
  scanf("%f",&c1.y);
  printf("\nEnter second complex number:");
  printf("\nEnter Real Part:");
  scanf("%f",&c2.x);
  printf("\nEnter Imaginary Part:");
  scanf("%f",&c2.y);
  while(1)
  {
    printf("\n1:ADDITION\n2:SUBTRACTION\n3:MULTIPLICATION\n");
    printf("4:DIVISION\n5:EXIT");
    printf("\nEnter Ur choice:");
    scanf("%d",&ch);
    switch(ch)
    {
         case 1:c3=add(c1,c2);
                display(c3);
                break;
         case 2:c3=sub(c1,c2);
                display(c3);
                break;
         case 3:c3=mul(c1,c2);
                display(c3);
                break;
         case 4:c3=div(c1,c2);
                display(c3);
                break;
         case 5:exit();
    }
  }
}

void read1(complex k)
{
   printf("\nEnter real part:");
   scanf("%f",&k.x);
   printf("\nEnter imaginary part:");
   scanf("%f",&k.y);
}
```

```
void display(complex k)
{
   printf("\nRESULATANT COMPLEX NUMBER IS:");
   if(k.y<0)
     printf("%.2f-i%.2f",k.x,fabs(k.y));
   else
     printf("%.2f+i%.2f",k.x,k.y);
}

complex add(complex s, complex t)
{
   complex u;
   u.x=s.x+t.x;
   u.y=s.y+t.y;
   return u;
}

complex sub(complex s, complex t)
{
   complex u;
   u.x=s.x-t.x;
   u.y=s.y-t.y;
   return u;
}

complex mul(complex s, complex t)
{
   complex u;
   u.x=((s.x*t.x)-(s.y*t.y));
   u.y=((s.x*t.y)+(s.y*t.x));
   return u;
}

complex div(complex s, complex t)
{
   complex u;
   float z=(t.x*t.x)+(t.y*t.y);
   u.x=((s.x*t.x)+(s.y*t.y))/z;
   u.y=((s.y*t.x)-(t.y*s.x))/z;
   return u;
}
```

UNION

Union is also a user-defined data type which is similar to structure. i.e., Union is a collection of heterogeneous (different) data types that can pack together into a single unit. The general form of union declaration is:

```
Syntax:     union Tag
            {
                    Datatype Member1;
                    Datatype Member2;
                            - - -
                            - - -
                    Datatype Membern;
            };
```

Where,

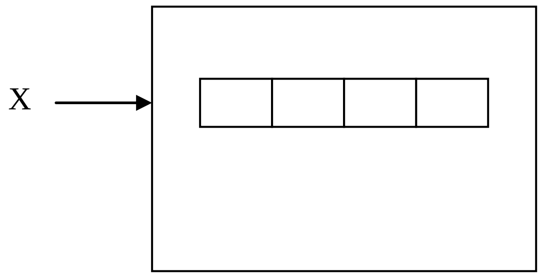
union is a keyword used to declare the union data type.
Tag is name the union that follows same rules as valid identifier.
Member1, Member2, - - - , Membern are elements of the union.

```
Example:    union Account
            {
                    int acno;
                    float balance;
                    char type;
            };
```

Memory is allocated for the union only at the time of creating union variable. The general form of creating union variables is:

```
Syntax:     union Tag Variable1, Variable2, - - - - - , Variablep;
Example:    union Account X;
```

For this, the memory allocation will be:



Note: The major difference between a structure and union is in terms of their storage space.

In structures, each member has its own storage location, where as in union, all members share the common location.

In union, compiler selects the member which occupies highest memory, and that memory is reserved only. So, that all the members of the union are shared that common memory. It implies that, although a union may contain many members of different types, it can handle only one member at a time.

```
/* WRITE A PROGRAM TO DECLARE MEMBERS OF THE UNION ACCOUNT AS ACCOUNT
NUMBER, ACCOUNT TYPE AND BALANCE */

#include<stdio.h>
union Account
{
        int acno;
        char actype[10];
        float acbalance;
};
main()
{
        union Account x;
        clrscr();
        printf("\nEnter Account Number:");
        scanf("%d",&x.acno);
        printf("\nAccount Number:%d",x.acno);
        fflush(stdin);
        printf("\n\n\nEnter Account Type:");
        gets(x.actype);
        printf("\nAccount Type:%s",x.actype);
        printf("\n\n\nEnter Balance Amount:");
        scanf("%f",&x.acbalance);
        printf("\nBalance Amount:%.2f RS",x.acbalance);
}
```

```
/* EXAMPLE PROGRAM TO DISPLAY SIZE OF UNION ELEMENTS AND UNIION */

union demo
{
        int x;
        float y;
        char z;
};
main()
{
        union demo var;
        clrscr();
        printf("\nSize of x:%d Bytes",sizeof(var.x));
        printf("\nSize of y:%d Bytes",sizeof(var.y));
        printf("\nSize of z:%d Bytes",sizeof(var.z));
        printf("\nSize of union:%d Bytes",sizeof(var));
}
```

1. A union may be a member of a structure, and the structure may be a member of a union.
2. The main advantage of union is to save the memory compared to the structure.

UNION OF STRUCTURES

A structure can be nested within another structure. In the same way a union can be nested within another union. We can also create a structure in a union and vice-versa.

```
/* EXAMPLE PROGRAM TO USE STRUCTURE WITHIN UNION */

main()
{
        struct x
        {
                float f;
                char p[2];
        };
        union y
        {
                struct x obj;
        };
        union y demo;
        clrscr();
        demo.obj.f=5.46789;
        demo.obj.p[0]='K';
        demo.obj.p[1]='#';
        printf("\nValue 1:%.2f",demo.obj.f);
        printf("\nValue 2:%c",demo.obj.p[0]);
        printf("\nValue 3:%c",demo.obj.p[1]);
}
```

***

UNIT – VI

Files – Introduction, Streams and File Types, Steps for File Operations, File I/O Structures, Read and Write Other File Function, Searching Errors in Reading/Writing of Files, Low Level Disk I/O, Command Line Arguments, Application of Command Line Arguments, File Status Fuctions.

I/O FUNCTIONS

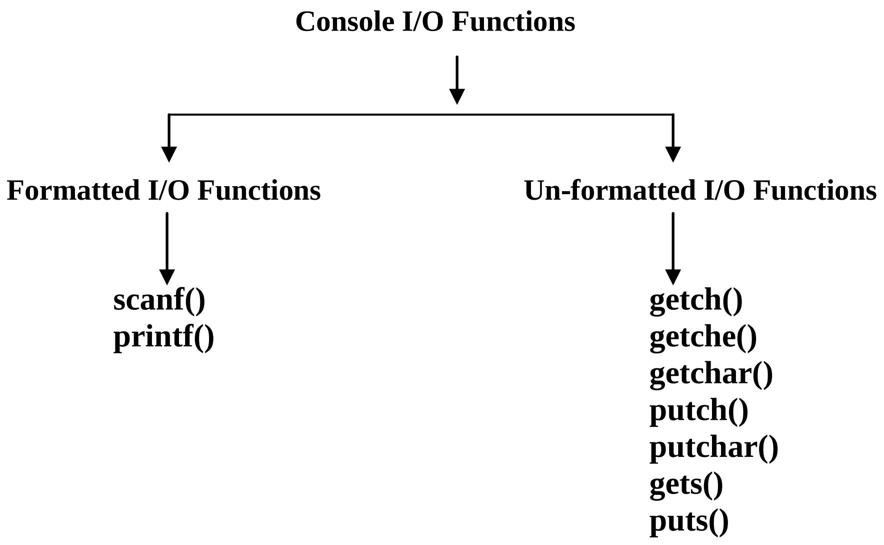
scanf() and printf() library functions are console oriented I/O functions which always use keyboard and screen to read and write data. As similar to these I/O functions number of library functions are available. Those functions can be classified into three categories as:

1.	Console I/O Functions	: Functions to receive input from keyboard and display on monitor
2.	Disk I/O Functions	: Functions to perform I/O operations on a Floppy disk or Hard disk
3.	Port I/O Functions	: Functions to perform I/O operations on various ports of the computer.

1. CONSOLE I/O FUNCTIONS

Console I/O functions always use the terminals keyboard and screen as the target place. These functions work on console effectively as long as the data is small. Console representation is not suitable to manage large volume of data.

Console I/O functions can further classified into two categories as formatted and un-Formatted console I/O functions.

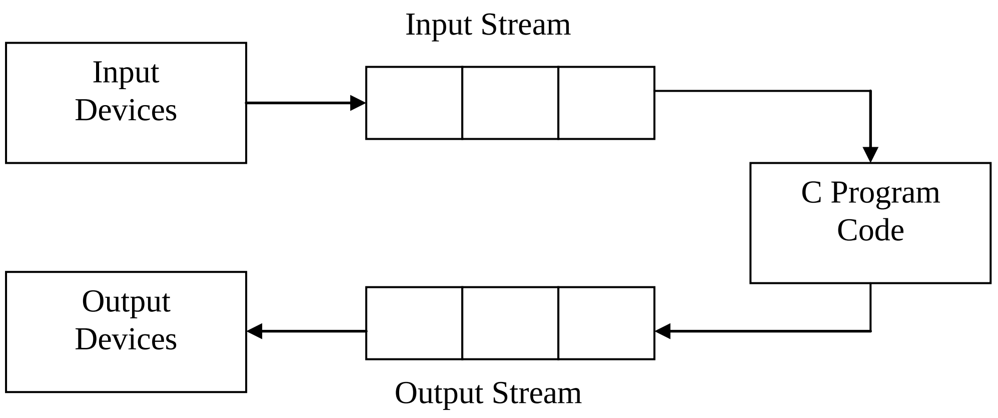


2. DISK I/O FUNCTIONS

Disk I/O functions are performed on entities called **Files**.

Streams:

A stream is a sequence of flow of data bytes, which is used to read and write data. The streams that represent the input data of program are known as **input streams** and the streams that represent the output data of program are known as **output streams**.



Input streams interpret data from different input devices such as keyboard, mouse etc., and provide as input data to the program.

Output streams obtain data from the program and write the data on different output devices such as memory unit or print them on the screen.

Therefore, stream acts as an interface between a program and input/output devices.

FILES

A **file** is a collection of information that is stored at a particular area on the disk.

File stream is a sequence of bytes, which is used to read and write data on files.

File name is a string of characters that make up a valid name for the operating system. It may contain two parts as a primary name and an optional period with an extension.

Example:	**Input.dat**	**(Data File)**	
	Sum.c	**(C Language File)**	
	Cpath.bat	**(Batch File)**	
	Hello.exe	**(Executable File)**	
	Sample.obj	**(Objective File)**	**etc.,**

Basic operations performed on the files are:

1. Defining (or) Naming a file
2. Opening a file
3. Reading data from a file
4. Writing data into a file
5. Closing a file.

Defining and Opening a File

All files should be declared as type **FILE**, which is a defined data type by the compiler. The general format of declaring and opening a file is:

Syntax: **FILE *FilePointer;** **/* Declaration */**
FilePointer = fopen("FileName", "Mode"); **/* Opening */**

Where,

- In the declaration statement, FilePointer defines as a "pointer to the data type FILe".
- fopen() function receives two arguments – FileName and Mode.
    - FileName is name of the file to be opened. FileName in C language can also contain path information. The path specifies the drive and/or director where the file is located.
    - Mode specifies the job to be performed on the file. Mode can be any one of the following.

MODE	MEANING
r	Open the file for reading only
w	Open the file for writing only
a	Open the file for appending data

- When the file is opened in **read** mode, if it exists, the file is opened with current contents safe; otherwise, an error occurs.
- When the file is opened in **write** mode, if it exists, the file is opened with current contents are deleted; otherwise, a file with specified name is created by the compiler.
- When the file is opened in **append** mode, if it exists, the file is opened with current contents safe and the file pointer placed at the end of the file; otherwise, a file with specified name is created by the compiler.

Most of the recent compilers include additional modes of operations as

MODE	MEANING
r+	Open the file for reading and writing
w+	Open the file for writing and reading
a+	Open the file for appending data and reading

Input and Output operations on Files

1. getc() and putc() functions:

Simplest I/O functions used for reading and writing from and to a file are getc() and put() functions.

getc() function is used to read a character from a file that has been opened in read mode. The general format of getc() function is:

Syntax: ch = getc(FilePointer);

Where,
ch is character type variable.
Here,
File Pointer is opened in read mode, and the function read the data character by character from the file and assigned to the variable ch. Whenever the end of the file has been reached, getc() function will return and end-of-file marker EOF.

putc() function is used to write a character into a file that has been opened in write mode. The general format of putc() function is:

Syntax: putc(ch, FilePointer);

Where,
ch is character type variable.
Here,
FilePointer is opened in write mode and the function writes the data of ch into the file.

Closing a File

When all operations are completed over files, it is better to close the files; when we want to open the same file in a different mode. For this, use the syntax as:

Syntax: fclose(FilePointer);

ERROR HANDLING DURING I/O OPERATIONS

It is possible that errors may occur during I/O operations on files. Typical errors are:

- ✓ Trying to use a file that has not been opened.
- ✓ Trying to perform an operation on a file, when the file is opened in another mode.
- ✓ Opening a file with an invalid name.
- ✓ Trying to read beyond the end-of-file mark etc.,

To handle most of these errors on files, C language provides two library functions **ferror()** and **feof()** that can help us to detect I/O errors.

When the file is opened using fopen() function, a file pointer is returned. If the file cannot be opened with some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not.

Example:

```
if(fp == NULL)
{
        - - -
        - - -
}
```

feof():

This function used to test for end-of-file condition. The general format of feof() function is:

Syntax: **feof(FilePointer)**

Function accepts FilePointer as its only argument and returns a non-zero integer value if EOF reached; otherwise, it returns zero.

ferror():

This function used to report status of the file indication. The general format of ferror() function is:

Syntax: **ferror(FilePointer)**

Function accepts FilePointer as its only argument and returns a non-zero integer value if an error has been detected up to that point during processing; otherwise, it returns zero.

perror():

perror() is a standard library function used to print error messages specified by the compiler.

With this function, whenever an error occurred; the function prints error message to stderr. First the argument string is printed, then a colon, then a message corresponding to the current value of error, finally a new line.

Syntax: **perror("string");**

/* WRITE A PROGRAM TO READ THE DATA FROM A KEYBOARD AND WRITE IT INTO A FILE */

```
#include<stdio.h>
main()
{
	char ch;
	FILE *fp;
	clrscr();
	fp=fopen("demo.txt","w");
	printf("\nEnter Text:");
	ch=getchar();
	while(ch!=EOF)
	{
		putc(ch,fp);
		ch=getchar();
	}
	fclose(fp);
}
```

**/* LP:	WRITE A PROGRAM TO READ THE DATA FROM A FILE AND PRINT IT ON MONITOR */**

```
#include<stdio.h>
main()
{
	char ch;
	FILE *fp;
	clrscr();
	fp=fopen("demo.txt","r");
	if(fp==NULL)
	{
		printf("\nFILE OPENING ERROR");
		exit();
	}
	printf("\nFile Contents Are:");
	ch=getc(fp);
	while(ch!=EOF)
	{
		printf("%c",ch);
		ch=getc(fp);
	}
	fclose(fp);
}
```

/* WRITE A PROGRAM TO APPEND THE DATA INTO A FILE */

```
#include<stdio.h>
main()
{
	char ch;
	FILE *fp;
	clrscr();
	fp=fopen("demo.txt","a");
	printf("\nEnter Additional Data:\n");
	ch=getchar();
```

```
        while(ch!=EOF)
        {
                putc(ch,fp);
                ch=getchar();
        }
        fclose(fp);
}
```

/* LP: WRITE A PROGRAM TO COPY THE CONTENTS OF ONE FILE INTO ANOTHER FILE */

```
#include<stdio.h>
main()
{
        char ch;
        FILE *fp1,*fp2;
        clrscr();
        fp1=fopen("demo.txt","r");
        if(fp1==NULL)
        {
                printf("\nFILE OPENING ERROR");
                exit();
        }
        fp2=fopen("exam.txt","w");
        while((ch=getc(fp1))!=EOF)
        {
                putc(ch,fp2);
        }
        fclose(fp2);
        fclose(fp1);
}
```

/* **LP: WRITE A PROGRAM TO MERGE TWO FILES INTO A THIRD FILE */**

```
#include<stdio.h>
main()
{
        char ch;
        FILE *f1,*f2,*f3;
        clrscr();
        f1=fopen("demo1.txt","r");
       f3=fopen("result.txt","w");
        if(f1==NULL)
        {
                printf("\nFile Opening Error");
                exit();
        }
        while((ch=fgetc(f1))!=EOF)
        fputc(ch,f3);
        fcloseall();

        f2=fopen("demo2.txt","r");
       f3=fopen("result.txt","a");
```

```
	if(f2==NULL)
	{
		printf("\nFile Opening Error");
		exit();
	}
	while((ch=fgetc(f2))!=EOF)
	fputc(ch,f3);
	fcloseall();
	f3=fopen("result.txt","r");
	if(f3==NULL)
	{
		printf("\nFile Opening Error");
		exit();
	}
	printf("\nMERGING CONTENTS ARE:\n");
	while((ch=fgetc(f3))!=EOF)
	printf("%c",ch);
	fclose(f3);
}
```

/* WRITE A PROGRAM TO READ THE DATA FROM A FILE AND PRINT NUMBER OF CHARACTERS, WORDS, DIGITS, LINES AND SPECIAL SYMBOLS */

```
#include<stdio.h>
main()
{
	char ch;
	FILE *fp;
	int nc=0,nd=0,nw=1,nl=1,nsp=0;
	clrscr();
	fp=fopen("check.txt","r");
	if(fp==NULL)
	{
		printf("\nFILE OPENING ERROR");
		exit();
	}
	while((ch=getc(fp))!=EOF)
	{
		if((ch>='a'&&ch<='z') || (ch>='A'&&ch<='Z'))
			nc=nc+1;
		else if(ch>='0'&&ch<='9')
			nd=nd+1;
		else if(ch==' '||ch=='\t')
			nw=nw+1;
		else if(ch=='\n')
		{
			nw=nw+1;
			nl=nl+1;
		}
		else
		nsp=nsp+1;
	}
```

```
        printf("\n No of Characters:%d",nc);
        printf("\n No of Digits:%d",nd);
        printf("\n No of Words:%d",nw);
        printf("\n No of Lines:%d",nl);
        printf("\n No of Special Symbols:%d",nsp);
        fclose(fp);
}
```

```
/* EXAMPLE PROGRAM FOR ERROR HANDLING MECHANISM */

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

main()
{
        FILE *f;
        char ch;
        clrscr();
        f=fopen("out.dat","w");
        while(!feof(f))
        {
                ch=fgetc(f);
                if(ferror(f))
                {
                        perror("Error Occured:");
                        exit();
                }
                putchar(ch);
        }
        fclose(f);
}
```

Input and Output operations on Files

1. fgetc() and fputc() functions:

fgetc() and fputc() functions provides same functionality as getc() and putc() functions.

fgetc(): **fgetc()** function is used to read a character from a file that has been opened in read mode. The general format of the fgetc() function is:

Syntax: **ch = fgetc(FilePointer);**

Where,

ch is character type variable.

Here,

File Pointer is opened in read mode, and the function read the data character by character from the file and assigned to the variable ch. Whenever the end of the file has been reached, fgetc() function will return and end-of-file marker EOF.

fputc(): **fputc()** function is used to write a character into a file that has been opened in write mode. The general format of the fputc() function is:

Syntax: **fputc(ch, FilePointer);**

Where,

ch is character type variable.

Here,

FilePointer is opened in write mode and the function writes the data of ch into the file.

2. fgets() and fputs() functions:

fgets() and fputs() functions are string oriented functions that can be handled an entire line as a string at a time.

fgets(): fgets() function is used to read a set of characters as a string from a given file. The general format of the fgets() function is:

Syntax: **fgets(char[], int, FilePointer);**

Where,

The first argument is the character array where the string is stored.
The second argument is the maximum size of the string.
The third argument is the FilePointer of the file to be read.

The function returns a NULL pointer if the end of file has been reached.

fputs(): fputs() function is used to write a string into a file. The general format of the fputs() function is:

Syntax: **fputs(char[], FilePointer);**

Where,

The first argument is the character array to be written into the file.
The second argument is the FilePointer of the file to write.

The function returns a non-negative value on successful completion; otherwise, it returns EOF.

```
/* WRITE A PROGRAM TO READ THE DATA FROM A FILE AND PRINT NUMBER OF LINES IN IT */

#include<stdio.h>
#include<string.h>
main()
{
        char ch[50];
        FILE *fp;
        int lines=0;
        clrscr();
        fp=fopen("check.txt","r");
        if(fp==NULL)
        {
                printf("\nFILE OPENING ERROR");
                exit();
        }
        printf("\nFile Contents Are:\n");
        while((fgets(ch,sizeof(ch),fp))!=NULL)
        {
        puts(ch);
        lines=lines+1;
        }
        printf("\n No of Lines:%d",lines);
        fclose(fp);
}
```

```
/* WRITE A PROGRAM TO COPY THE CONTENTS OF ONE FILE TO ANOTHER FILE LINE BY LINE */

#include<stdio.h>
#include<string.h>
main()
{
        char ch[50];
        FILE *fp1,*fp2;
        clrscr();
        fp1=fopen("check.txt","r");
        if(fp1==NULL)
        {
                printf("\nFILE OPENING ERROR");
                exit();
        }

        fp2=fopen("ptr.txt","w");
        while((fgets(ch,sizeof(ch),fp1))!=NULL)
        fputs(ch,fp2);
        fclose(fp2);
        fclose(fp1);
}
```

3. getw() and putw() functions:

getw() and putw() functions are used to read and write integer values on a given file.

getw(): getw() function is used to read an integer value from a given file. The general format of the getw() function is:

Syntax: **getw(FilePointer);**

This function receives FilePointer as an argument and returns next integer from the input file. It returns EOF when an error encountered.

putw(): putw() function is used to write an integer value into the specified file. The general format of the putw() function is:

Syntax: putw(N,FilePointer);

Where,

N is an integer value to be written into the given file with FilePointer opened in write mode.

/* WRITE A PROGRAM TO CREATE AN INPUT DATA FILE WHICH CONTAINS A LIST OF INTEGERS. READ THE SAME DATA FROM THE FILE AND WRITE EVEN AND ODD NUMBERS INTO TWO SEPARTE FILES */

```
#include<stdio.h>
main()
{
        int item,n,i;
        FILE *f1,*f2,*f3;
        clrscr();
        f1=fopen("input.dat","w");
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        printf("\nEnter %d Numbers:",n);
        for(i=1;i<=n;i++)
        {
                scanf("%d",&item);
                putw(item,f1);
        }
        fclose(f1);

        f1=fopen("input.dat","r");
        if(f1==NULL)
        {
                printf("\nFILE OPENING ERROR");
                exit();
        }
        f2=fopen("even.dat","w");
        f3=fopen("odd.dat","w");
        while((item=getw(f1))!=EOF)
        {
                if(item%2==0)
                        putw(item,f2);
                else
                        putw(item,f3);
        }
        fcloseall();
        printf("\nEVEN NUMBERS ARE:");
        f2=fopen("even.dat","r");
        while((item=getw(f2))!=EOF)
        printf("%6d",item);
        fclose(f2);
        printf("\nODD NUMBERS ARE:");
        f3=fopen("odd.dat","r");
        while((item=getw(f3))!=EOF)
        printf("%6d",item);
        fclose(f3);
}
```

4. fprintf() and fscanf() functions:

Most compilers support two functions namely fprintf() and fscanf() functions, that can handle a group of mixed data simultaneously. The functions fprintf() and fscanf() perform I/O operations that are identical to printf() and scanf() functions, except that they work on files.

fscanf(): fscanf() function is used to read mixed data simultaneously form a given file. The general format of the fscanf() function is:

Syntax: fscanf(FilePointer, “ControlString”, List);

Where,

FilePointer associated with the file that has been opened for reading.
ControlString consists of format specification for the items in the list.
List may include variables, constants, strings etc.,

fprintf(): fprintf() function is used to write mixed data simultaneously into a given file. The general format of the fpritnf() function is:

Syntax: fprintf(FilePointer, “ControlString”, List);

Where,

FilePointer associated with the file that has been opened for writing.
ControlString consists of output format specification for the items in the list.
List may include variables, constants, strings etc.,

```
/* WRITE A PROGRAM TO CREATE A FILE THAT CONTAINS INVENTORY DETAILS LIKE PRODUCT
NUMBER, PRODUCT NAME AND PRODUCT COST. READ THE SAME DATA FROM THE FILE AND
PRINT IT */

#include<stdio.h>
struct product
{
        int id;
        char name[20];
        float price;
}p;
main()
{
        char flag='y';
        FILE *f;
        clrscr();
        f=fopen("product.dat","w");
        while(flag=='y')
        {
          printf("\nEnter product id:");
          scanf("%d",&p.id);
          fflush(stdin);
          printf("\nEnter product name:");
          gets(p.name);
          printf("\nEnter product cost:");
          scanf("%f",&p.price);
          fprintf(f,"%d\t%s\t%f\n",p.id,p.name,p.price);
          printf("\nDo you want add another record(y/n):");
          fflush(stdin);
          flag=getchar();
        }
        fclose(f);
        f=fopen("product.dat","r");
        if(f==NULL)
        {
                printf("\nFILE OPENING ERROR");
                exit();
        }
        printf("\nPRODUCT DETAILS ARE:\n");
        while((fscanf(f,"%d%s%f",&p.id,p.name,&p.price))!=EOF)
        printf("\n%d\t%s\t%.2f",p.id,p.name,p.price);
        fclose(f);
}
```

STANDARD DOS SERVICES

fopen() function is used to open a file in a specified mode. Ms-Dos also predefines pointers for five standard files. To access these pointers, we need not use fopen() function. Those standard file pointers are:

STANDARD FILE POINTER	DESCRIPTION
Stdin	Standard input device (Keyboard)
Stdout	Standard output device (Monitor)
Stderr	Standard error device (Monitor)
Stdaux	Standard auxiliary device (Serial Port)
Stdprn	Standard printing device (Printer)

```
/* WRITE A PROGRAM TO READ DATA FROM STANDARD INPUT DEVICE AND PRINT IT
ON STANDARD OUTPUT DEVICE */

#include<stdio.h>
main()
{
        char ch;
        clrscr();
        printf("\nEnter data:");
        while((ch=fgetc(stdin))!=EOF)
        fputc(ch,stdout);
}
```

```
/* WRITE A PROGRAM TO READ DATA FROM A FILE AND PRINT IT ON THE PRINTER */

#include<stdio.h>
main()
{
        char ch;
        FILE *f;
        clrscr();
        f=fopen("product.dat","r");
        if(f==NULL)
        {
                printf("\nFile Opening Error");
                exit();
        }
        while((ch=fgetc(f))!=EOF)
        fputc(ch,stdprn);
}
```

FILE FORMATS

File formats can be categorized into two ways as: Text mode format and Binary mode format. This classification arises at the time of opening the file.

When a file is opened either in “r”, “w” or “a” modes, default file format is **text mode format.** If the user wants to open the file in binary format, explicitly necessary to specify as “rb”, “wb”, or “ab”.

There are three main differences raised between a text file and binary files. Those are:

1. Handling of new lines
2. Representation of End-Of-File
3. Storage of numbers.

1. In text mode, a new line character is converted into the combination of carriage return – line feed before being written into the disk.

In binary mode, conversions not take place. A new line character is written into the disk as in the original format.

2. In text mode, a special character is inserted after the last character in the file to mark the End-Of-File. If this character is detected at any point in the file, then read function would return the EOF signal to the program.

In binary format, there is no such special character present to mark the End-Of-File. The binary mode files keep track of the End-Of-File from the number of characters present in the directory entry of the file.

3. In text mode, while storing numbers in files, numbers are stored as string of characters.

Consider a number 4523.

In memory, it occupies 2 bytes. Whereas when the number placed on the disk, it would occupy 4 bytes as one byte per each character. Since, it depends on magnitude of the number.

In such case, large amount of data storage in a disk file is inefficient.

In binary mode, number would occupy same number of bytes on disk as it occupies in memory unit. With this, the above number occupies only 2 bytes even on the disk file.

fread() and **fwrite()** functions are used to read and write data in binary format. The general formats of fread() and fwrite() functions are:

Syntax: **fread(&x, sizeof(x), 1, FilePointer);**
fwrite(&x, sizeof(x), 1, FilePointer);

Here,

The first argument is address of the argument.
The second argument is size of the argument in bytes.
The third argument is number of arguments read of write at one time.
The final argument is the FilePointer.

```
/* WRITE A PROGRAM TO DEMONSTRATE THE DIFFERENCE BETWEEN TEXT MODE Vs BINARY
MODE FORMATS */

#include<stdio.h>
main()
{
        int x;
        float y;
        char z='\n';
        FILE *fp;
        clrscr();
        fp=fopen("Exam.txt","w");
        printf("\nEnter One Integer and One Floating Number:\n");
        scanf("%d%f",&x,&y);
        fprintf(fp,"%d\n%f",x,y);
        fclose(fp);
        fp=fopen("Del.txt","wb");
        fwrite(&x,sizeof(x),1,fp);
        fwrite(&z,sizeof(z),1,fp);
        fwrite(&y,sizeof(y),1,fp);
        fclose(fp);
}
```

```
/* WRITE A PROGRAM TO MAINTAIN STRUDENT DATA BASE IN A FILE PRINT IT ON THE
MONITOR USING fread() AND fwrite() FUNCTIONS */
#include<stdio.h>
typedef struct
{
        int rno;
        char name[25],branch[5];
}student;
main()
{
        int i,n;
        student s[20];
        FILE *f=fopen("st.txt","wb");
        clrscr();
        printf("\nEnter how many students:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("\nEnter student %d details:",i);
                scanf("%d%s%s",&s[i].rno,s[i].name,s[i].branch);
                fwrite(&s[i],sizeof(s[i]),1,f);
        }
        fclose(f);
        f=fopen("st.txt","rb");
        printf("\n Database Details Are:\n");
        for(i=1;i<=n;i++)
        {
                fread(&s[i],sizeof(s[i]),1,f);
                printf("\n%d\t%s\t%s",s[i].rno,s[i].name,s[i].branch);
        }
        fclose(f);
}
```

FILE ACCESSING TECHNIQUES (or) FILE TYPES

Every open file has an associated file position indicator, which describes where read and write operations take place in the file. The position is always specified in bytes from the beginning of the file.

When a file is opened in either read (or) write mode, the position indicator is always at beginning of the file i.e., at position ‘0’. If the file is opened in append mode, the position indicator is at the end of the file.

File accessing techniques can be classified into two types as:

1. Sequential file processing
2. Random access file processing

In **sequential file processing**, the file pointer moves character by character without skipping. i.e., read or write operations performed sequentially. getc(), putc(), fgets(), fputs() etc., functions support sequential file processing.

<u>Random Access File Processing</u>: Random access file processing is done to change the position of file pointer from one position to another position. Here, operations are performed in random access manner. Most important functions used in random access file processing are:

1. fseek()   2. ftell()   3. rewind()

<u>1. fseek():</u> fseek() function is used to move the position indicator to a desired location with in the file. The general format of the fseek() function is:

Syntax: fseek(FilePointer, OffSet, Position);

Where,

OffSet is the number of bytes to be moved and it must be long integer. The value may be positive of negative. If the OffSet value be positive, file pointer moves to forward direction; otherwise, file pointer moves to backward direction.

Position specifies the starting position in the file to move. Position can take any one of the following three values.

VALUE	MEANING
0	Beginning of File
1	Current Position
2	End of File

When the operation is successful, fseek() function returns zero; otherwise, it return -1.

Examples:

1.	fseek(fp,0L,0)	-	Go to beginning of the file
2.	fseek(fp,0L,1)	-	Stay at the current position
3.	fseek(fp,0L,2)	-	Go to end of the file
4.	fseek(fp,m,0)	-	Move to m bytes from beginning of the file
5.	fseek(fp,m,1)	-	Go to forward by m bytes from current position
6.	fseek(fp,-m,2)	-	Go to backward by m bytes from the end of file.

<u>2. ftell():</u> ftell() function is used to return the current position of the file pointer in the file. The general format of the ftell() function is:

Syntax: N = ftell(FilePointer);

Where,

N is a long integer variable.

Function returns current position (in bytes) as long integer value. If any error encountered, then the function returns -1.

3. rewind(): rewind() function is used to reset the file pointer to beginning of the file. The general format of the rewind() function is:

Syntax: rewind(FilePointer);

```
/* WRITE A PROGRAM TO READ DATA FROM  A FILE AND PRINT POSITION BYTE OF
EACH CHARACTER IN THE FILE */

#include<stdio.h>
main()
{
        FILE *fp;
        long n;
        char ch;
        clrscr();
        fp=fopen("win.dat","r");
        if(fp==NULL)
        {
                printf("\nFile Opening Error");
                exit();
        }
        while(1)
        {
                n=ftell(fp);
                if((ch=fgetc(fp))==EOF)
                exit();
                printf("\n%c position is: %ld Byte",ch,n);
        }
        fclose(fp);
}
```

```
/* WRITE A PROGRAM DISPLAY EVERY NTH CHARACTER FROM A FILE */

#include<stdio.h>
main()
{
        FILE *fp;
        long n;
        int count=0,i;
        clrscr();
        fp=fopen("den.txt","r");
        if(fp==NULL)
        {
                printf("\nFile Opening Error");
                exit();
        }
        while(fgetc(fp)!=EOF)
        count=count+1;
        printf("\n\nNo of Characters:%d",count);
        rewind(fp);
        printf("\nEnter Which Character U Want to print:");
        scanf("%ld",&n);
```

```
	for(i=0;i<count;i+=n)
	{
		fseek(fp,n-1L,1);
		printf("    %c",fgetc(fp));
	}
	fclose(fp);
}
```

/* WRITE A PROGRAM TO REPLACE EVERY NTH CHARACTER BY A SPECIFIED CHARACTER */

```
#include<stdio.h>
main()
{
	FILE *fp;
	long n;
	int count=0,i;
	char ch,item;
	clrscr();
	fp=fopen("den.txt","r+");
	if(fp==NULL)
	{
		printf("\nFile Opening Error");
		exit();
	}
	while(fgetc(fp)!=EOF)
	count=count+1;
	rewind(fp);
	printf("\nBefore Replacing File Contents Are:\n");
	while((ch=fgetc(fp))!=EOF)
	printf("%c",ch);
	printf("\n\nNo of Characters:%d",count);
	rewind(fp);
	printf("\nEnter Which Character U Want to replace:");
	scanf("%ld",&n);
	printf("\nEvery a Character to replace:");
	scanf(" %c",&item);
	for(i=0;i<count;i+=n)
	{
		fseek(fp,n-1L,1);
		fputc(item,fp);
	}
	rewind(fp);
	printf("\n\nAfter Replacing File Contents Are:\n");
	while((ch=fgetc(fp))!=EOF)
	printf("%c",ch);
	printf("\n\nNo of Characters:%d",count);
	fclose(fp);
}
```

```
/* WRITE A PROGRAM TO PRINT THE CONTENTS OF A FILE IN REVERSE ORDER */

#include<stdio.h>
main()
{
        FILE *fp;
        long n,i;
        char ch;
        clrscr();
        fp=fopen("den.txt","r");
        if(fp==NULL)
        {
                printf("\nFile Opening Error");
                exit();
        }
        fseek(fp,-1L,2);
        n=ftell(fp);
        printf("\nFile Contents in Reverse Order:\n");
        for(i=1;i<=n+1;i++)
        {
                fseek(fp,-i,2);
                putchar(fgetc(fp));
        }
        fclose(fp);
}
```

COMMAND LINE ARGUMENTS

The arguments that pass to main() function at the command prompt of the operating system are called command line arguments. The general format of the main() function with command line arguments is:

Syntax: **int main(int argc, char *argv[])**

- main() function returns an integer value on successful completion.
- The first argument of main() function **argc** is an integer argument called **argument counter**, that counts number of arguments passed at the command prompt.
- The second argument **argv** is argument vector; which is an array of pointer to strings. Arguments are stored in terms of strings.
- The arguments passed at the command prompt are stored in argv[0], argv[1] and so on. Here, argv[0] is reserved for to store name of the executing program.

/* WRITE A PROGRAM TO PERFORM ADDITION OF GIVEN NUMBERS USING COMMAND LINE ARGUMENTS */

```
#include<string.h>
main(int argc,char *argv[])
{
        int x,y;
        clrscr();
        x=atoi(argv[1]);
        y=atoi(argv[2]);
        printf("\nNumber of arguments:%d",argc);
        printf("\nAddition Result:%d",x+y);
}
```

/* WRITE A PROGRAM TO REVERSE THE FIRST N CHARACTERS IN A FILE BY SPECIFYING FILE NAME AND N ARE AT COMMAND LINE */

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

 main(int argc,char *argv[])
 {
        FILE *f;
        char ch,x[10];
        int n,i;
        clrscr();
        f=fopen(argv[1],"r+");
        n=atoi(argv[2]);
        for(i=0;i<n;i++)
        x[i]=fgetc(f);
        rewind(f);
        for(i=n-1;i>=0;i--)
        fputc(x[i],f);
        rewind(f);
        printf("\nFile Contents Are:");
        while((ch=fgetc(f))!=EOF)
        putchar(ch);
        fclose(f);
 }
```

***

UNIT - VIII

Searching and Sorting – Exchane (Bubble) sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort, Searching – Linear and Binary Search Methods.

SORTING

Sorting refers to the arrangement of data items either in the ascending (increasing) order or in the descending(decreasing) order.

Some of the most important sorting techniques are:

a) Bubble Sort
b) Insertion Sort
c) Selection Sort
d) Quick Sort
e) Merge Sort

Sorting techniques are classified into two types as: Internal sorting techniques and External sorting techniques.

Sorting that performed in main memory is called internal sorting. Internal sorting techniques are used to handle small amount of data.

Examples: Bubble Sort, Insertion Sort, Selection Sort, Quick Sort.

Sorting that performed with the interaction of secondary storage devices like disks or tapes is called external sorting. External sorting techniques are used to handle large volume of data.

Examples: Merge Sort.

a) Bubble Sort (Exchange Sort):

Let K be a list of 'n' elements. Sorting refers to the operation rearranging the elements of K such that: $K[1] \leq K[2] \leq \ldots\ldots\ldots \leq K[n]$.
For this, Bubble sort procedure works as:

Step 1: Compare the first and second data items. If the first data item is greater than the second data item, then make an interchange. Compare the second and third data items. If the second data item is greater than the third data item, then make an interchange. The process is repeated till the last data item is reached.

Step 2: When the last data item is reached, it is said to be one pass. At the end of the first pass, the largest element is bubble out and occupies at the last position in the array.

Step 3: Step1 & 2 are repeated for the data items between 1 to n-1. At the end of the second pass, the next highest data item bubble out and occupies it's appropriate place.

Step 4: The steps are repeated till the last pass n-1 is reached.

Step 5: At the end of the last pass, entire elements are available in sorted order.

Example: **Sort the following elements using bubble sort**

16	**12**	**85**	**67**	**11**			
Pass 1:	16	12	85	67	11		
	16 > 12		TRUE		Interchage 16 & 12		
	12	16	85	67	11		
	16 > 85		FALSE				
	85 > 67		TRUE		Interchage 85 & 67		
	12	16	67	85	11		
	85 > 11		TRUE		Interchage 85 & 11		
	12	16	67	11	**85**		
Pass 2:	12	16	67	11	**85**		
	12 > 16		FALSE				
	16 > 67		FALSE				
	67 > 11		TRUE		Interchange 67 & 11		
	12	16	11	67	**85**		
	67 > 85		FALSE				
	12	16	11	**67**	**85**		
Pass 3:	12	16	11	**67**	**85**		
	12 > 16		FALSE				
	16 > 11		TRUE		Interchange 16 & 11		
	12	11	**16**	**67**	**85**		
Pass 4:	12	11	**16**	**67**	**85**		
	12 > 11		TRUE		Interchange 12 & 11		
	11	**12**	**16**	**67**	**85**		
Sorted Elements Are:			11	12	16	67	85

ALGORITHM

BSort(K,n): Let K is an array with ‘n’ elements. This algorithm sorts the elements of K in ascending order.

```
Step 1:     Repeat Steps 2 & 3 for i ← 1 to n-1
Step 2:         j ← 1
Step 3:         Repeat while j ≤ n-i
                    If K[j] > K[j+1] Then
                        Interchange K[j] and K[j+1]
                    EndIf
                    j ← j+1
                EndRepeat
Step 4:     Return
```

/* PROGRAM TO SORT THE GIVEN ELEMENTS USING BUBBLE SORT */

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

void BSort(int[],int);

main()
{
        int x[10],i,n;
        clrscr();
        printf("\nEnter How Many Elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=1;i<=n;i++)
        scanf("%d",&x[i]);
        BSort(x,n);
        printf("\nElements In Sorted Order Are:");
        for(i=1;i<=n;i++)
        printf("%5d",x[i]);
}

void BSort(int k[10],int n)
{
        int i,j,temp;
        for(i=1;i<=n-1;i++)
        {
                j=1;
                while(j<=n-i)
                {
                        if(k[j]>k[j+1])
                        {
                                temp=k[j];
                                k[j]=k[j+1];
                                k[j+1]=temp;
                        }
                        j=j+1;
                }
        }
        return;
}
```

Complexity of Bubble Sort:

Number of comparisons in the first pass = n-1
Number of comparisons in the second pass = n-2
-
-
Number of comparisons in the last pass = 1

Total number of comparisons at the end of the procedure = (n-1) + (n-2) + - - - - - + 2 + 1
= (n * (n-1)) / 2
= $n^2/2 - n/2$
= $O(n^2)$

- **Worst case and Average case time complexity of bubble sort is $O(n^2)$.**

INSERTION SORT

In insertion sort, at each pass such number of items are placed in sorted order. Suppose K is an array that contains ‘n’ elements such as K[1], K[2], ... , K[n]. Then insertion sort procedure works as:

Step 1: Select the second data item and compare it with the first data item. If the second data item is less than the first data item then it insert it before the first data item. Otherwise, proceed with the next step.

Step 2: Select the third data item and compare it with the second data item. If the it is less than the second data item then compare it with the first data item. If it is less than the first data item then insert it before the first data item. Otherwise, insert it in between first data item and second data item.

Step 3: Repeat the above steps for n-1 times. The entire list of items are available in sorted order at the end of the last pass n-1.

Example:	**Sort the following elements using insertion sort.**					
	12	**11**	**16**	**20**	**19**	
	-999	12	11	16	20	19
Pass 1:	temp = 11					
	11 < 12 TRUE					
	-999	12	12	16	20	19
	11 < -999 FALSE					
	-999	**11**	**12**	16	20	19
Pass 2:	temp = 16					
	16 < 12 FALSE					
	-999	**11**	**12**	**16**	20	19
Pass 3:	temp = 20					
	20 < 16 FALSE					
	-999	**11**	**12**	**16**	**20**	19
Passe 4:	temp = 19					
	19 < 20 TRUE					
	-999	11	12	16	20	20
	19 < 16 FALSE					
	-999	**11**	**12**	**16**	**19**	**20**

Sorted Elements Are: 11 12 16 19 20

<u>ALGORITHM</u>

InSort(K,n): Let K is an array with ‘n’ elements. This algorithm sorts the elements of K in ascending order.

```
Step 1:     K[0] ← -999
Step 2:     Repeat Steps 3 to 5 for i ← 2 to n
Step 3:         temp ← K[i]
                j ← i-1
Step 4:         Repeat while temp < K[j]
                    K[j+1] ← K[j]
                    j ← j-1
                EndRepeat
Step 5:         K[j+1] ← temp
Step 6:     Return
```

```
/* PROGRAM TO SORT THE GIVEN ELEMENTS USING INSERTON SORT */

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

void InSort(int[],int);

main()
{
        int x[10],i,n;
        clrscr();
        printf("\nEnter How Many Elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=1;i<=n;i++)
        scanf("%d",&x[i]);
        InSort(x,n);
        printf("\nElements In Sorted Order Are:");
        for(i=1;i<=n;i++)
        printf("%5d",x[i]);
}

void InSort(int k[10],int n)
{
        int i,j,temp;
        k[0]=-999;
        for(i=2;i<=n;i++)
        {
                temp=k[i];
                j=i-1;
                while(temp<k[j])
                {
                        k[j+1]=k[j];
                        j=j-1;
                }
                k[j+1]=temp;
        }
        return;
}
```

Time Complexity:

The worst-case and average-case time complexity of insertion sort is O(n^2).

<u>SELECTION SORT</u> (Straight Forward Sorting)

In selection sort procedure, at each pass the smallest element from the list moves to its appropriate position.

Suppose K is an array that contains 'n' elements such as K[1], K[2], ... , K[n]. Then selection sort procedure works as:

Step 1: Select the smallest data item from index positions 1 to n, and interchange the element with the 1st index element. So, that the smallest element is available in the first position.

Step 2: Now, select the next smallest data item from index positions 2 to n, and interchange the element with the 2nd index element. So that the next smallest element is available in the second position.

Step 3: The above steps are repeated for n-1 times. At the end of the n-1th pass, total elements are available in sorted order.

Example: **Sort the following elements using selection sort.**

	16	**12**	**85**	**67**	**11**
Pass 1:	Min = 16				
	12 < 16		TRUE		Min = 12
	85 < 12		FALSE		
	67 < 12		FALSE		
	11 < 12		TRUE		Min = 11
	Interchange 16 & 11				
	11	12	85	67	16
Pass 2:	Min = 12				
	85 < 12		FALSE		
	67 < 12		FALSE		
	16 < 12		FALSE		
	11	**12**	85	67	16
Pass 3:	Min = 85				
	67 < 85		TRUE		Min = 67
	16 < 67		TRUE		Min = 16
	Interchange 85 & 16				
	11	**12**	**16**	67	85
Pass 4:	Min = 67				
	85 < 67		FALSE		
	11	**12**	**16**	**67**	85

Sorted Elements Are: 11 12 16 67 85

ALGORITHM

SelSort(K,n): Let K is an array with 'n' elements. This algorithm sorts the elements of K in ascending order.

```
Step 1:        Repeat Thru Step 4 for i ← 1 to n-1
Step 2:            MinIndex ← i
Step 3:            Repeat for j ← i+1 to n
                   If K[j] < K[MinIndex] Then
                       MinIndex ← j
                   EndIf
               EndRepeat
Step 4:            If MinIndex ≠ i Then
                   Interchange K[i] and K[MinIndex]
               EndIf
Step 5:        Return
```

```
/* PROGRAM TO SORT THE GIVEN ELEMENTS USING SELECTION SORT */

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

void SelSort(int[],int);

main()
{
        int x[10],i,n;
        clrscr();
        printf("\nEnter How Many Elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=1;i<=n;i++)
        scanf("%d",&x[i]);
        SelSort(x,n);
        printf("\nElements In Sorted Order Are:");
        for(i=1;i<=n;i++)
        printf("%5d",x[i]);
}

void SelSort(int k[10],int n)
{
        int i,j,MinIndex,temp;
        for(i=1;i<=n-1;i++)
        {
                MinIndex=i;
```

```
            for(j=i+1;j<=n;j++)
            {
                    if(k[j]<k[MinIndex])
                      MinIndex=j;
            }
            if(MinIndex!=i)
            {
                    temp=k[i];
                    k[i]=k[MinIndex];
                    k[MinIndex]=temp;
            }
    }
    return;
}
```

Time Complexity:

The worst-case and average-case time complexity of insertion sort is $O(n^2)$.

QUICK SORT (Partition Exchange Sort)

Quick sort is a procedure based on the divide-and-conquer strategy. In this sorting technique, array elements are divided into two sub sub arrays depending on a specialized element called "**pivot"** element.

Let K be an array with n elements. Then the procedure for quick sort is as follows:

```
Step 1:    Initialize first element as pivot element.
Step 2:    Initialize a variable i at the first index and another variable j at last
           index+1.
Step 3:    Increment i value by 1 until K[i] ≥ pivot element.
Step 4:    Decrement j value by 1 until K[j] ≤ pivot element.
               If i < j Then
                   Interchange K[i] & K[j]
               EndIf
Step 5:    Repeat Step 3 and 4 until i ≥ j
Step 6:    Interchange the values of K[j] and pivot element.
```

With this one pass completed. At the end of the pass, the pivot element is positioned at the correct appropriate position. Now, the elements before the pivot element are less than or equal to pivot element and after the pivot element are greater than the pivot element.

Now, the same procedure is repeated on the elements before the pivot element as well as on the elements after the pivot element.

When all passes are completed, then list of elements are available in sorted order.

Example: **Sort the following elements using quick sort.**

12 9 17 16 94

<u>Pass 1:</u>

K[1:5] 12 9 17 16 94

pivot = 12
i = 1
j = 6

i = 2 $9 \geq 12$ FALSE
i = 3 $17 \geq 12$ TRUE

12 9 17 16 94

j = 5 $94 \leq 12$ FALSE
j = 4 $16 \leq 12$ FALSE
j = 3 $17 \leq 12$ FALSE
j = 2
i > j (3 > 2)
Interchange 9 & 12

K[1:5] = 9 **12** 17 16 94
K[1:1] K[3:5]

<u>Pass 2:</u>

K[3:5] = 17 16 94
pivot = 17
i =3
j = 6

i = 4 $16 \geq 17$ FALSE
i = 5 $94 \geq 17$ TRUE

j = 5 $94 \leq 17$ FALSE
j = 4
i > j (5 > 4)
Interchange 16 & 17

K[1:3] = 16 **17** 94

K[1:5] = 9 **12** 16 **17** 94

Sorted Elements Are: 9 12 16 17 94

Example: **Sort the following elements using quick sort.**

76 92 11 24 49 33

ALGORITHM

QuickSort(K,LB,UB): Let K is an array with ‘n’ elements. LB refers to the first index 1 and UB refers to the last index n at the initial call. This algorithm sorts the elements of K in ascending order.

```
Step 1:     flag ← TRUE
Step 2:     If LB < UB Then
                i ← LB
                j ← UB + 1
                pivot ← K[LB]
                Repeat while flag
                    i ← i+1
                    Repeat while K[i] < pivot
                        i ← i+1
                    EndRepeat
                    j ← j-1
                    Repeat while K[j] > pivot
                        j ← j-1
                    EndRepeat
                    If i < j Then
                        K[i] ↔ K[j]
                    Else
                        flag ← FALSE
                    EndIf
                EndRepeat
                K[LB] ↔ K[j]
                Call QuickSort(K,LB,j-1)
                Call QuickSort(K,j+1,UB)
            EndIf
Step 3:     Return
```

```
/* PROGRAM TO SORT THE GIVEN ELEMENTS USING QUICK SORT */


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

void qsort(int[],int,int);

void main()
{
        int x[10],i,n;
        clrscr();
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=1;i<=n;i++)
        scanf("%d",&x[i]);
        printf("\nBefore Sorting Elements Are:");
        for(i=1;i<=n;i++)
        printf("%4d",x[i]);
        qsort(x,1,n);
        printf("\nAfter Sorting Elements Are:");
        for(i=1;i<=n;i++)
        printf("%4d",x[i]);
}

void qsort(int k[10],int lb,int ub)
{
        int i,j,temp,pivot,flag;
        flag=1;
        if(lb<ub)
        {
                i=lb;
                j=ub+1;
                pivot=k[lb];
                while(flag)
                {
                        i=i+1;
                        while(k[i]<pivot)
                        i=i+1;
                        j=j-1;
                        while(k[j]>pivot)
                        j=j-1;
                        if(i<j)
                        {
                                temp=k[i];
                                k[i]=k[j];
                                k[j]=temp;
                        }
                        else
                        flag=0;
                }
                temp=k[lb];
                k[lb]=k[j];
                k[j]=temp;
                qsort(k,lb,j-1);
                qsort(k,j+1,ub);
        }
        return;
}
```

Time Complexity:

The worst-case time complexity of quick sort is O(n^2). It occurs when the list of elements already in sorted order.

Whereas the average-case time complexity of quick sort is O(nlogn), which is less compare to worst-case time complexity.

MERGE SORT

Merge sort is also an example based on divide-and-conquer strategy. In this sorting technique, the array elements are divided into two sub arrays based on

Mid = (Low + High) / 2

Where,

Low is the first index of the array and High is the last index of the array.

Once, the sub arrays are formed, each set is individually sorted and the resulting sub sequences are merged to produce a single sorted sequence of data elements.

Divide-and-Conquer strategy is applicable as splitting the array into sub arrays; and combining operating is merging the sub arrays into a single sorted array.

Merging is the process of combining two sorted lists into a single sorted list. While performing merging operation, the two sub lists muste be in sorted order.

Example: **Sort the following elements using merge sort.**

12 9 17 16 94

Consider k[1:5] = 12 9 17 16 94

Low = 1 High = 5 Mid = (1+5)/2 = 3

Then, k[1:5] splitted into two sub arrays as: k[1:3] and k[4:5]

Consider k[1:3] = 12 9 17

Low = 1 High = 3 Mid=(1+3)/2 = 2

Then, k[1:3] splitted into two sub array as: k[1:2] and k[3:3]

Consider k[1:2] = 12 9

Low = 1 High = 2 Mid=(1+2)/2 = 1

Then, k[1:2] splitted into two sub array as: k[1:1] and k[2:2]

Apply Merge operation on k[1:1] and k[2:2], it produces a sorted list k[1:2] as

K[1:2] = 9 12

Apply Merge operation on k[1:2] and k[3:3], it produces a sorted list k[1:3] as

K[1:3] = 9 12 17

Consider k[4:5] = 16 94

Low = 4 High = 5 Mid=(4+5)/2 = 4

Then, k[4:5] splitted into two sub array as: k[4:4] and k[5:5]

Apply Merge operation on k[4:4] and k[5:5], it produces a sorted list k[4:5] as
k[4:5] = 16 94

Apply Merge operation on k[1:3] and k[4:5], it produces a sorted list k[1:5] as

K[1:5] = 9 12 16 17 94

Sorted Elements Are: 9 12 16 17 94

ALGORITHM

MergeSort(Low, High): Let K is an array with 'n' elements. Low refers to the first index 1 and High refers to the last index n at the initial call. This algorithm sorts the elements of K in ascending order.

```
Step 1:     If Low < High Then
                Mid ← (Low+High) / 2
                Call Msort(Low,Mid)
                Call Msort(Mid+1,High)
                Call Merge(Low,Mid,High)
            EndIf
```

ALGORITHM

Merge(Low,Mid,High):

```
Step 1:     h ← Low
            i ← Low
            j ← Mid+1
Step 2:     Repeat while h ≤ Mid AND j ≤ High
                If K[h] ≤ K[j] Then
                    S[i]←K[h]
                    h ← h+1
                Else
                    S[i]←K[j]
                    j ← j+1
                EndIf
                i ← i+1
            EndRepeat
```

```
Step 3:     If h > Mid Then
                Repeat for p ← j to High
                    S[i]←K[p]
                    i ←i+1
                EndRepeat
            Else
                Repeat for p ← h to Mid
                    S[i]←K[p]
                    i ←i+1
                EndRepeat
            EndIf
Step 4:     Repeat for p ← Low to High
                K[p] ← S[p]
            EndRepeat
```

/* PROGRAM TO SORT THE GIVEN ELEMENTS USING MERGE SORT */

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

void msort(int,int);
void merge(int,int,int);

int K[10],S[10];

void main()
{
        int i,n;
        clrscr();
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=1;i<=n;i++)
        scanf("%d",&K[i]);
        printf("\nBefore Sorting Elements Are:");
        for(i=1;i<=n;i++)
        printf("%4d",K[i]);
        msort(1,n);
        printf("\nAfter Sorting Elements Are:");
        for(i=1;i<=n;i++)
        printf("%4d",K[i]);
}
```

```
void msort(int Low, int High)
{
        int Mid;
        if(Low<High)
        {
                Mid=(Low+High)/2;
                msort(Low,Mid);
                msort(Mid+1,High);
                merge(Low,Mid,High);
        }
}

void merge(int Low, int Mid,int High)
{
        int i,j,h,p;
        h=Low;
        i=Low;
        j=Mid+1;
        while(h<=Mid && j<=High)
        {
                if(K[h]<=K[j])
                {
                        S[i]=K[h];
                        h=h+1;
                }
                else
                {
                        S[i]=K[j];
                        j=j+1;
                }
                i=i+1;
        }
        if(h>Mid)
        {
                for(p=j;p<=High;p++)
                {
                        S[i]=K[p];
                        i=i+1;
                }
        }
         else
         {
                for(p=h;p<=Mid;p++)
                {
                        S[i]=K[p];
                        i=i+1;
                }
         }
        for(p=Low;p<=High;p++)
        K[p]=S[p];
        return;
}
```

Time Complexity:

The worst-case and average-case time complexity of merge sort is O(nlogn).

Note:

The main disadvantage of merge sort is its storage representation. In merge sort technique, while performing merging operation, the procedure required an auxiliary array which has same as the original array.

SEARCHING

Searching refers to the operation of finding the location of a given item in list of items. Consider an array is given with 'n' elements. A specific element 'item' is given to search. Now, we want to find whether the item is available in the list of n elements or not. If the search item is exist, then it refers to successful search; otherwise, it refers to unsuccessful search.

Most important techniques used for search operation are:

1. Linear search
2. Binary search

1. LINEAR SEARCH

Consider an array K with n elements as K[1], K[2], . . . K[n]. Suppose an item of information is given to search.

In this technique, compare item value with each element of K form index 1 to index n. At any position i, if K[i]=item, then return index i value refers to successful search; otherwise, return -1 refers to unsuccessful search.

Example: Search an element 19 from the list of elements: 18 35 78 23 19 709

item = 19

```
i=1     K[1]=item     →     18 = 19     FALSE
i=2     K[2]=item     →     35 = 19     FALSE
i=3     K[3]=item     →     78 = 19     FALSE
i=4     K[4]=item     →     23 = 19     FALSE
i=5     K[5]=item     →     19 = 19     TRUE

                    ELEMENT FOUND
```

Algorithm NonRecLSearch(K, n, item):

Suppose K is an array that contains ‘n’ elements. Search element is given in the variable ‘item’.

This function returns an index position ‘i' if the element is found; otherwise, return -1.

```
Step 1:     Repeat for i ← 1 to n
                If K[i]=item Then
                    Return i
                EndIf
            EndRepeat
Step 2:     Return -1
```

Algorithm RecLSearch(K, n,item):

```
Step 1:     If n = 0 Then
                Return -1
            ElseIf K[n]=item Then
                Return n
            Else
                Return RecLSearch(K, n-1, item)
            EndIf
```

```
// PROGRAM TO SEARCH A GIVEN ELEMENT USING LINEAR SEARCH

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

int NonRecLSearch(int[],int,int);
int RecLSearch(int[],int,int);

main()
{
        int x[10],n,p,s,i;
        clrscr();
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=1;i<=n;i++)
        scanf("%d",&x[i]);
        printf("\nEnter an Element to Search:");
        scanf("%d",&p);

        printf("\nNon-Recursive Manner:\n");
        s=NonRecLSearch(x,n,p);
        if(s!=-1)
                printf("\nELEMENT FOUND");
        else
                printf("\nELEMENT NOT FOUND");
        printf("\nRecursive Manner:\n");
        s=RecLSearch(x,n,p);
        if(s!=-1)
                printf("\nELEMENT FOUND");
        else
                printf("\nELEMENT NOT FOUND");
}

int NonRecLSearch(int K[10],int n, int item)
{
        int i;
        for(i=1;i<=n;i++)
        {
                if(K[i]==item)
                   return i;
        }
        return -1;
}

int RecLSearch(int K[10], int n, int item)
{
        if(n==0)
            return -1;
        else if(K[n]==item)
                return n;
        else
                return RecLSearch(K,n-1,item);
}
```

<u>Time Complexity:</u>

The worst-case and average-case time complexity of linear search is O(n).

<u>2. Binary Search</u> :

Binary search is another searching algorithm, that takes less time complexity compared with the linear search. Binary search can be applied only on the array which is available in sorted order.

Consider K is an with n elements as K[1]≤K[2]≤. . . . ≤K[n]. Suppose an item of information is given to search in the variable ITEM.
In binary search technique, first compute

Mid = (Low+High) / 2

Where, Low refers to the first index and High refers to last index of the array in the initial call. Now, the process falls into any one of the following three cases.

Case 1: If ITEM = K[Mid]; Then the search is successful search.

Case 2: If ITEM > K[Mid]; Then the ITEM can appear only in the right half of the array. So, we reset the Low value as Low = Mid+1 and begin search again.

Case 3: If ITEM < K[Mid]; Then the ITEM can appear only in the left half of the array. So, we reset the High value as High = Mid-1 and begin search again.

This procedure is repeated upto we reach Low > High. When we obtain this condition, it indicates that the search is unsuccessful search.

Example 1: Search an element 44 from the list

11 22 30 33 41 44 55

```
Low = 1         High = 7        Mid = (1+7) / 2 = 4
        ITEM = K[Mid]           44 = 33 FALSE
        ITEM > K[Mid]           44 > 33 TRUE
                        Reset Low = 4+1 = 5

Low = 5         High = 7        Mid = (5+7)/2 = 6
        ITEM = K[Mid]           44 = 44 TRUE

                SUCCESSFUL SEARCH, ITEM FOUND
```

<u>Algorithm Non-Recursive BSearch(K, Low, High, ITEM):</u>

Consider K is a sorted array and ITEM of information is given to search. Low, High and Mid variables refers to beginning, ending and middle locations of the given array K.
This function is used to find the location of the search ITEM and returns index value if found; otherwise, it return -1.

```
Step 1:     Repeat while Low ≤ High
                Mid ← (Low+High)/2
                If ITEM < K[Mid] Then
                    High ← Mid-1
                ElseIf ITEM > K[Mid] Then
                    Low ← Mid+1
                Else
                    Return Mid
                EndIf
            EndRepeat
Step 2:     Return -1
```

<u>Algorithm Recursive BSearch(K, Low, High, ITEM):</u>

```
Step 1:     If Low ≤ High Then
                Mid ← (Low+High)/2
                If ITEM = K[Mid] Then
                    Return Mid
                ElseIf ITEM < K[Mid] Then
                    Return RBsearch(K, Low, Mid-1, ITEM)
                Else
                    Return RBsearch(K,Mid+1,High,ITEM)
                EndIf
            EndIf
Step 2:     Return -1
```

```c
// PROGRAM TO SEARCH A GIVEN ELEMENT USING BINARY SEARCH

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

int NonRecBSearch(int[],int,int,int);
int RecBSearch(int[],int,int,int);

main()
{
        int x[10],n,p,s,i;
        clrscr();
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=1;i<=n;i++)
        scanf("%d",&x[i]);
        printf("\nEnter an Element to Search:");
        scanf("%d",&p);

        printf("\nNon-Recursive Manner:\n");
        s=NonRecBSearch(x,1,n,p);
        if(s!=-1)
                printf("\nELEMENT FOUND");
        else
                printf("\nELEMENT NOT FOUND");

        printf("\nRecursive Manner:\n");
        s=RecBSearch(x,1,n,p);
        if(s!=-1)
                printf("\nELEMENT FOUND");
        else
                printf("\nELEMENT NOT FOUND");
}
```

```
int NonRecLSearch(int K[10],int Low,int High,int item)
{
        int Mid;
        while(Low <= High)
        {
                Mid=(Low+High)/2;
                if(item<K[Mid])
                        High=Mid-1;
                else if(item>K[Mid])
                        Low=Mid+1;
                else
                   return Mid;
        }
        return -1;
}

int RecBSearch(int K[10], int Low,int High,int item)
{
        int Mid;
        if(Low<=High)
        {
                Mid=(Low+High)/2;
                if(item==K[Mid])
                        return Mid;
                else if(item<K[Mid])
                        return RecBSearch(K,Low,Mid-1,item);
                else
                        return RecBSearch(K,Mid+1,High,item);
        }
        return -1;
}
```

Analysis:

The complexity is measured by the number of comparisions to locate the search item in the given array elements.

In binary search, each comparison reduces the size of the array into half. So that number of comparsions are less compare to linear search. Hence, the worst-case and average-case time complexity of binary search is O(log n).

***

UNIT - VII

Data Structures – Overview of Data Structure, Representation of a Stack, Stack Related Terms, Operation on Stack, Implementation of a Stack, Representation of Arithmetic Expressions, Infix, Prefix and Postfix Notations, Evaluation of Postfix Expression, Convertion Expression from Infix to Postfix, Recursion, Queues – Various Positions of Queue, Representation of Queue, Insertion, Deletion, Searching Operations.
Linked List – Singly Linked List, Lined List with and without header, Insertion, Deletion and Searching Operations.