MALLA REDDY INSTITUTE OF TECHNOLOGY & SCIENCE
(SPONSORED BY MALLA REDDY EDUCATIONAL SOCIETY) Permanently Affiliated
to JNTUH & Approved by AICTE, New Delhi
NBA Accredited Institution, An ISO 9001:2015 Certified, Approved by UK Accreditation Centre
Granted Status of 2(f) & 12(b) under UGC Act. 1956, Govt. of India.

PROGRAMMING FOR PROBLEM SOLVING

COURSE FILE

B.Tech (CSE,IT,AI&ML,CS,DS) I Year – I Semester
R22 Regulation

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

(2022-2026)

Compiled by

1. **S.Shirisha**, Assistant Professor
2. **N.Akhilandeshwari,** Assistant Professor
3. **V.Naveen Kumar**, Assistant Professor
4. **B.Pradeep Kumar**, Assistant Professor

MALLA REDDY INSTITUTE OF TECHNOLOGY AND SCIENCE
(SPONSORED BY MALLA REDDY EDUCATIONAL SOCIETY)
Affiliated to JNTUH & Approved by AICTE, New Delhi
NAAC & NBA Accredited, ISO 9001:2015 Certified, Approved by UK Accreditation Centre
Granted Status of 2(f) & 12(b) under UGC Act 1956,Govt. of India.
Maisammaguda, Dhulapally, Post via kompally, Secunderabad – 500 100
www.mrits.ac.in

COURSE FILE SUBJECT:

PROGRAMMING FOR PROBLEM SOLVING

ACADEMIC YEAR: 2022-2026.

REGULATION: R22

NAME OF THE FACULTY : S.SHIRISHA
N.AKHILANDESHWARI
V.NAVEEN KUMAR
B. PRADEEP KUMAR

DEPARTMENT: CSE & IT

YEAR & SECTION: I Year-I SEM (CSE,CS,DS,AI&ML & IT)

SUBJECT CODE: CS103ES

Programming For Problem Solving

R22 B.Tech. Syllabus **JNTU Hyderabad**

PROGRAMMING FOR PROBLEM SOLVING

B.Tech. I Year I Sem.	**L**	**T**	**P**	**C**
	3	**0**	**0**	**3**

Course Objectives:

- To learn the fundamentals of computers.
- To understand the various steps in program development.
- To learn the syntax and semantics of the C programming language.
- To learn the usage of structured programming approaches in solving problems.

Course Outcomes: The student will learn

- To write algorithms and to draw flowcharts for solving problems.
- To convert the algorithms/flowcharts to C programs.
- To code and test a given logic in the C programming language.
- To decompose a problem into functions and to develop modular reusable code.
- To use arrays, pointers, strings and structures to write C programs.
- Searching and sorting problems.

UNIT - I: Introduction to Programming

Compilers, compiling and executing a program.Representation of Algorithm - Algorithms for finding roots of a quadratic equations, finding minimum and maximum numbers of a given set, finding if a number is prime number Flowchart/Pseudocode with examples, Program design and structured programming

Introduction to C Programming Language: variables (with data types and space requirements), Syntax and Logical Errors in compilation, object and executable code, Operators, expressions and precedence, Expression evaluation, Storage classes (auto, extern, static and register), type conversion, The main method and command line arguments Bitwise operations: Bitwise AND, OR, XOR and NOT operators Conditional Branching and Loops: Writing and evaluation of conditionals and consequent branching with if, if-else, switch-case, ternary operator, goto, Iteration with for, while, do- while loops
I/O: Simple input and output with scanf and printf, formatted I/O, Introduction to stdin, stdout and stderr. Command line arguments

UNIT - II: Arrays, Strings, Structures and Pointers:

Arrays: one and two dimensional arrays, creating, accessing and manipulating elements of arrays
Strings: Introduction to strings, handling strings as array of characters, basic string functions available in C (strlen, strcat, strcpy, strstr etc.), arrays of strings
Structures: Defining structures, initializing structures, unions, Array of structures

Programming For Problem Solving

Pointers: Idea of pointers, Defining pointers, Pointers to Arrays and Structures, Use of Pointers in self-referential structures, usage of self referential structures in linked list (no implementation) Enumeration data type

UNIT - III: Preprocessor and File handling in C:

Preprocessor: Commonly used Preprocessor commands like include, define, undef, if, ifdef, ifndef
Files: Text and Binary files, Creating and Reading and writing text and binary files, Appending data to existing files, Writing and reading structures using binary files, Random access using fseek, ftell and rewind functions.

UNIT - IV: Function and Dynamic Memory Allocation:

Functions: Designing structured programs, Declaring a function, Signature of a function, Parameters and return type of a function, passing parameters to functions, call by value, Passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc., Limitations of Recursive functions Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types

UNIT - V: Searching and Sorting:

Basic searching in an array of elements (linear and binary search techniques), Basic algorithms to sort array of elements (Bubble, Insertion and Selection sort algorithms), Basic concept of order of complexity through the example programs

TEXT BOOKS:

1. Jeri R. Hanly and Elliot B.Koffman, Problem solving and Program Design in C 7th Edition, Pearson
2. B.A. Forouzan and R.F. Gilberg C Programming and Data Structures, Cengage Learning, (3rd Edition)

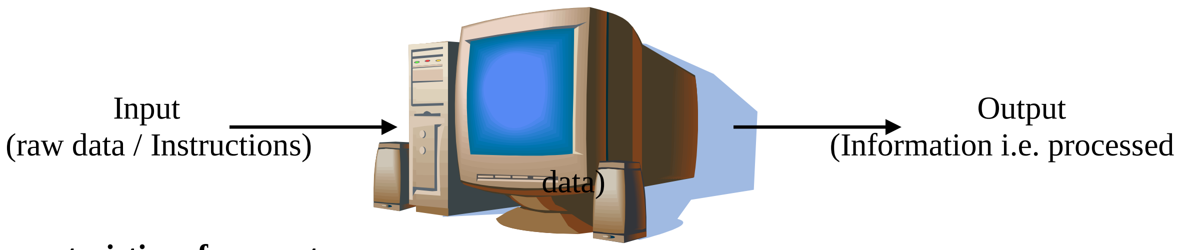
REFERENCE BOOKS:

1. Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall of India
2. E. Balagurusamy, Computer fundamentals and C, 2nd Edition, McGraw-Hill
3. Yashavant Kanetkar, Let Us C, 18th Edition, BPB
4. R.G. Dromey, How to solve it by Computer, Pearson (16th Impression)
5. Programming in C, Stephen G. Kochan, Fourth Edition, Pearson Education.
6. Herbert Schildt, C: The Complete Reference, Mc Graw Hill, 4th Edition
7. Byron Gottfried, Schaum's Outline of Programming with C, McGraw-Hill

UNIT -1

COMPUTER :

It is a high speed electronic device that accepts and stores input data and instructions, processes the data and produces the desired output.



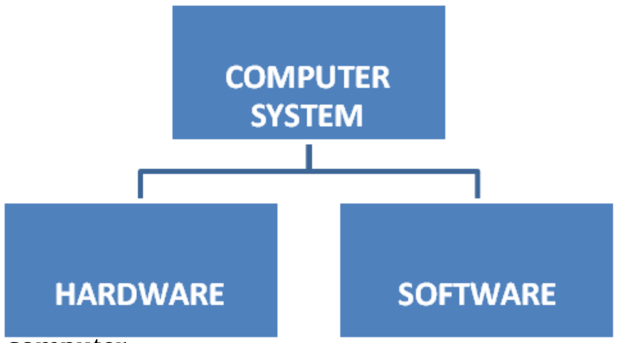
Characteristics of computers

- **Speed:** the computer process data at an extremely fast rate, at millions of instructions per second.
- **Reliability**: computer provide very high speed accompanied by an equally high level of reliability.
- **Accuracy:** the level of accuracy depends on the instrucations and the type of processor used in the computer
- **Storage capacity**: computers are capable of storing enormous amounts of data that can be located and retrieved very quickly
- Resource sharing: In organizations computers can be connected to each other to from a n/w

COMPUTER SYSTEMS

* It is made up of 2 major components

* They are : 1) Hardware
2) Software



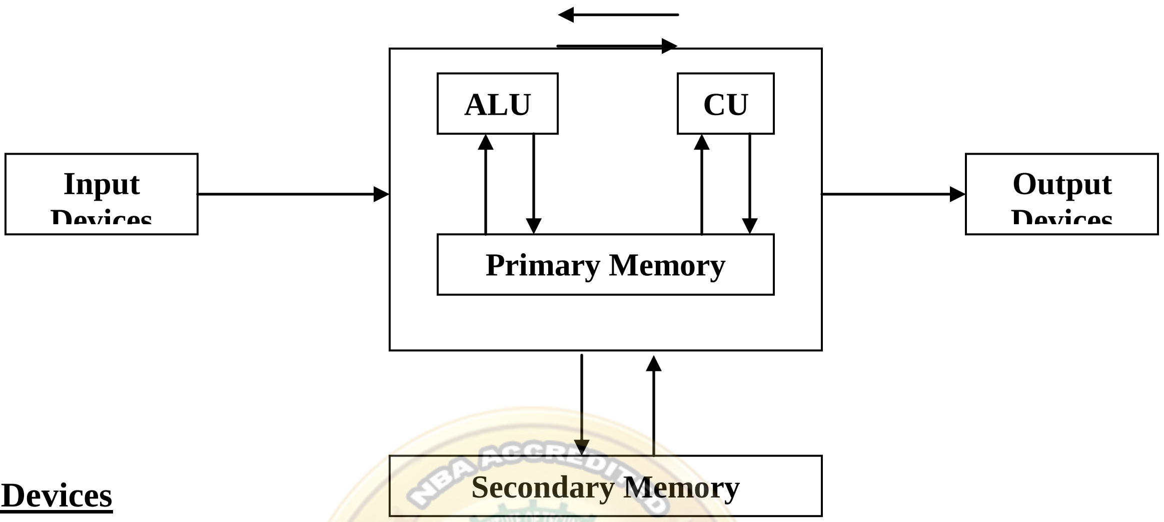
Computer Hardware :

❖ These are the physical components of a computer

❖ It consists of 5 Parts

1) Input Devices 2) Output Devices 3) CPU 4) Primary Storage 5) Secondary Storage

Block diagram of a Computer



Input Devices

- These are used to enter data and programs into the computer
- These are for man to machine communication
- egs: Keyboard, mouse, scanner, touch screen, audio input.

Output Devices

- These are used to get the desired output from the computer
- These are for machine to man communication
- egs: Printer, Monitor, Speakers
- If the output is shown on monitor then it is called "**Soft copy**"
- If the output is printed on a paper using printer then it is called "**hard copy**"

CPU (Central Processing Unit)

- It is responsible for processing the instructions
- It consists of 3 parts
  1) ALU – Arithmetic & Logic Unit
  2) CU- Control Unit
  3) Memory
- ALU performs arithmetic operations like addition,subtraction,multiplication,division and logical operations like comparisons among data
- CU is responsible for movement of data inside the system

- Memory is used for storage of data and programs. It is divided into 2 parts.
  1) Primary Memory/ Main Memory
  2) Secondary Memory/ Auxilary Memory

1) Primary Memory

- It is also called main memory
- Data is stored temporarily i.e. data gets erased when computer is turned off
- Eg: RAM

2) Secondary Memory

- It is also called as auxilary memory
- Data is stored permanently so that user can reuse the data even after power loss.
- Eg: Hard disk, CD, DVD, Floppy etc.

Operating System

- It acts as an interface between the user and the hardware
- It makes the system to operate in an efficient manner
- Egs: MS DOS, Windows, UNIX, LINUX, etc.

System Support

- They provide some utilities and other operating services
- eg: Sort utilities, disk formatting utilities, Security monitors

COMPUTER LANGUAGES

- Computer programming languages are used to give instructions to the computer in a language which computer understands
- Computer languages are classified into 3 types
  1) Machine languages
  2) Symbolic languages
  3) High level languages

1) Machine languages

- Computer is a machine and since its memory can store only 1's and 0's, instructions must be given to the computer in streams of 1's and 0's i.e. binary code.
- These are easily understandable by the machine

- Programs written in binary code can be directly fed to computer for execution and it is known as machine language.

Advantage :

- Execution is very fast since there is no need of conversion

Disadvantage :

- Writing and reading programs in machine language is very difficult
- Machine instructions are difficult to remember

2) <u>Symbolic Languages</u>

- It is also called as assembly language
- An assembly program contains “Mnemonics”
- “Mnemonic” means information can be memorized easily and is generally in the form of abbreviations.

Advantage :

- Easy to read and write programs in assembly language when compared to machine language
- Mnemonics are easy to remember

Disadvantage :

- Assembly programs are machine dependent
- Execution is slow since it needs conversion into machine language
- “Assembler” is used to convert assembly language program into machine language.

$$\begin{tikzcd} \text{Assembly language program code} \arrow[r] & \boxed{\textbf{Assembler}} \arrow[r] & \text{Machine Language} \end{tikzcd}$$

3) <u>High level languages</u>

- A set of languages which are very close to our native languages are called “ high-level languages”.
- High level languages have control structures, I/O facilities, hardware independence
- eg: FORTRAN, COBOL, PASCAL, C, C++ etc..

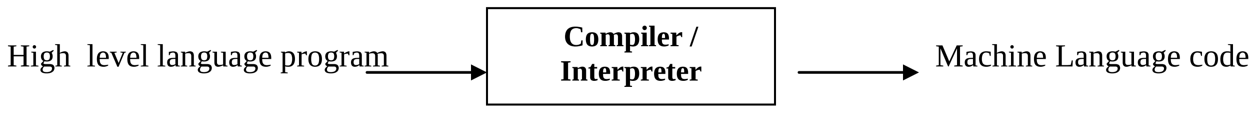
Advantage :

- Machine independence i.e. programs are “Portable” i.e. programs can be moved from one system to another

- Easy to learn and understand
- Takes less time to write programs

Disadvantage :

- High level language programs needs a translator for conversion into machine language
- ‘Compilers’ (or) ‘Interpreters’ are used for converting high level language program into machine language..



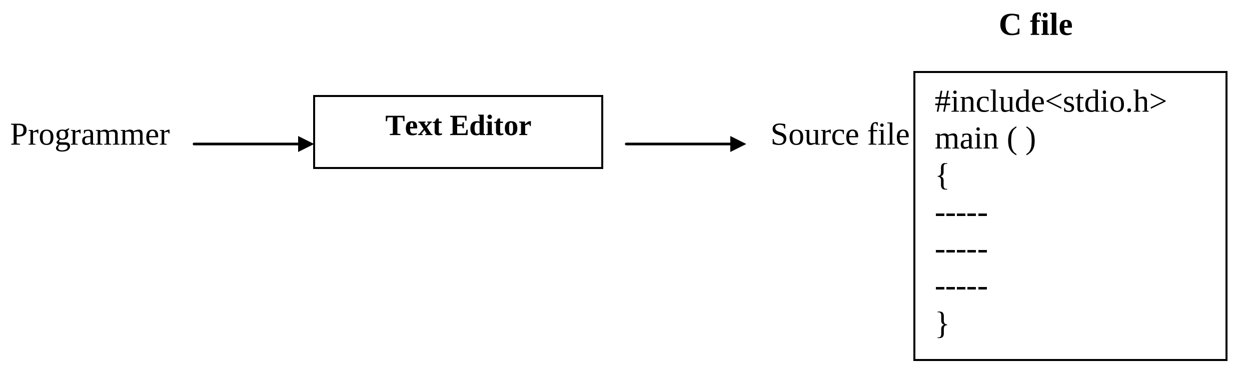
- Compiler converts entire statements in the program at a time.
- Interpreter converts one statement at a time.

CREATING AND RUNNING PROGRAMS

- Program consists of set of instructions written in a programming language
- The job of a programmer is to write and test the program.
- There are **4 steps** for converting a ‘C’ program into machine language.
  1) Writing and editing the program
  2) Compiling the program
  3) Linking the program
  4) Executing the program

1) Writing and editing the program

- ‘**Text editors**’ are used to write programs.
- Users can enter, change and store character data using text editors
- Special text editor is often included with a compiler
- After writing the program, the file is saved to disk. It is known as **‘source file’**
- This file is input to the compiler



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

2) Compiling the program

- "**Compiler**" is a software that translates the source file into machine language
- The 'C' compiler is actually 2 separate programs

  a) Preprocessor

  b) Translator

A) Preprocessor

- It reads the source code and prepares it for the translator
- It scans for special instructions known as 'preprocessor' commands which start with ' #' symbol
- These commands tell the preprocessor to look for special code libraries and make substitutions
- The result of preprocessing is called 'translation' unit

$$\begin{tikzcd}\text{Source File} \arrow[r] & \boxed{\textbf{Preprocessor}} \arrow[r] & \text{translation unit}\end{tikzcd}$$

B) Translator

- It does the actual work of converting the program into machine language
- It reads the translation unit and results in '**object module'** i.e., code in machine language
- But it is not yet executable because it does not have the 'C' and other functions included.

$$\begin{tikzcd}\text{Translation unit} \arrow[r] & \boxed{\textbf{Translator}} \arrow[r] & \text{Object Module}\end{tikzcd}$$

.obj file

```
00110 100
10101 010
```

3) Linking programs

- '**Linker**' assembles input /output functions, mathematical library functions and some of the functions that are part of source program into final executable program
- It is called as executable file that it is ready for execution

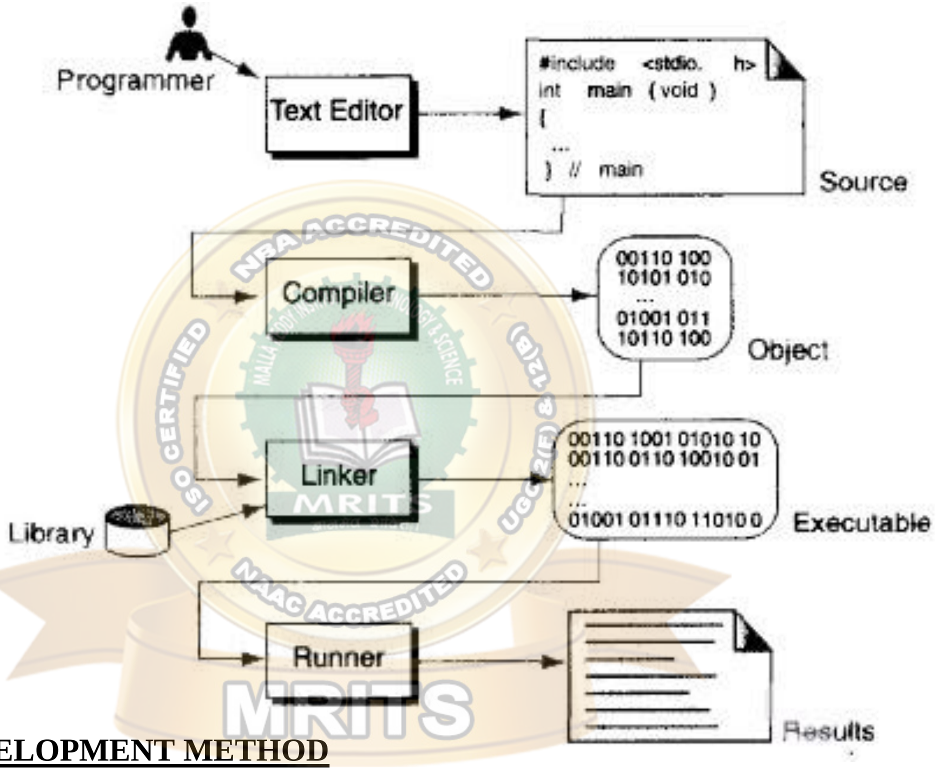
$$\begin{tikzcd}\text{Object file} \arrow[r] & \boxed{\textbf{Linker}} \arrow[r] & \text{executable file}\end{tikzcd}$$

.exe file

```
1011001100
110111011
1100101010
```

4) Executing Programs

- ‘**Loader**’ is the software that gets the program that is ready for execution into the memory
- When everything is loaded, the program takes control and the ‘**Runner**’ begins its execution.
- In the process of execution, the program reads the data from the user, processes the data and prepares the output



<u>SOFTWARE DEVELOPMENT METHOD</u>

- Software is collection of programs
- Programming is a problem solving task / activity
- Programmers use the software development method for solving problems
- It consists of the following **<u>6 phases</u>**

1) Requirements
2) Analysis
3) Design
4) Coding
5) Testing
6) Maintenance

1) <u>Requirements</u>

- Information about the problem must be stated clearly and unambiguously

- Main objective of this phase is to eliminate unimportant aspects and identify the root problem

2) Analysis

- It involves identifying the problem inputs, outputs, that the program must produce
- It also determines the required format in which results should be displayed

3) Design

- It involves designing algorithms, flowcharts (or) GUI's (Graphical User Interfaces)
- Designing the 'algorithm' is to develop a list of steps called algorithm to solve the problem and then verify that the algorithm solves the problem intended.
- "Top – down design" is followed i.e. list the major steps (or) sub problems that need to be solved
- "Flow charts" are used to get the pictorial representation of the algorithm.
- Algorithm for a programming problem consists of at least the following sub problems
  1. Get the data
  2. Perform the computations
  3. Display the results

4) Coding / Implementation

- This step involves writing algorithm as a program by selecting any one of the high – level languages that is suitable for the problem.
- Each step of the algorithm is converted into one (or) more statements in a programming language.

5) Testing

- Checking / verifying whether the completed program works as desired is called " Testing"
- Running the program several times using different sets of data verifies whether a program works correctly for every situation provided in the algorithm.
- After testing, the program must be free from the following errors.

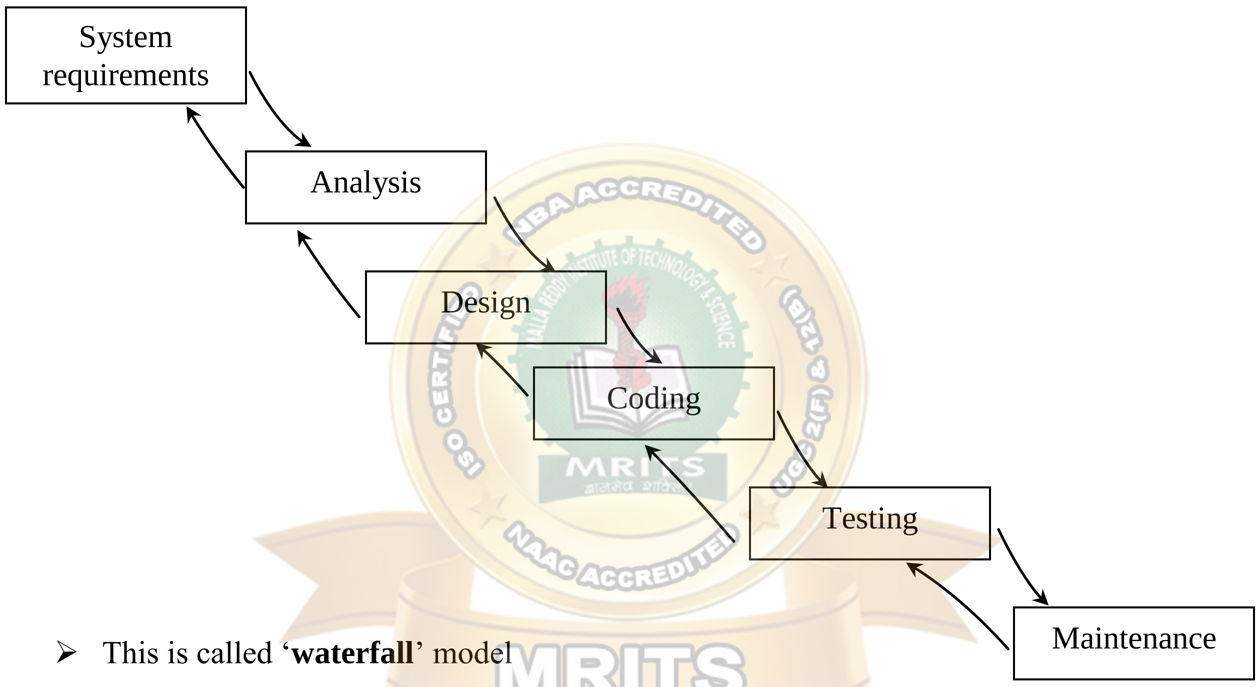
  a) Syntax errors

b) Logical errors

c) Run-time errors

6) <u>Maintenance</u>

- It involves modifying a program to remove previously undetected errors and to keep it up-to-date as government regulations (or) company polices change.
- Many organizations maintains a program for some period of time i.e. 5 years



- This is called '**waterfall**' model
- It is necessary to go back to the previous phase to rework it

<u>APPLYING THE SOFTWARE DEVELOPMENT METHOD</u>

1) <u>Problem requirement</u>

- Finding the roots of a quadratic equation, ax^2+bx+c
- There will be 2 roots for such quadratic equation

2) <u>Analysis</u>

Input : a,b,c values

Output: r_1, r_2 values

Procedure : $r_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$

$$r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

3) Design

Algorithm

```
1. start
2. Read a,b,c values
3. Compute d = $b^2 - 4ac$
4. if d > 0 then
        a) $r_1$ = −b+ sqrt (d)/(2*a)
        b) $r_2$ = −b −sqrt(d)/ (2*a)
5. otherwise if d = 0 then
        a) compute r1 = -b/2a
                     r2=-b/2a
        b) print r1,r2 values
6. otherwise if d < 0 then print roots are imaginary
7. Stop
```

4. Implementation

```
# include<stdio.h>
# include<conio.h>
# include<math.h>
main ( )
{
        float a,b,c,r1,r2,d;
        clrscr ( );
        printf ("enter a,b,c values");
        scanf (" %f %f %f", &a, &b, &c);
        d= b*b – 4*a*c;
        if (d>0)
        {
                r1 = -b+sqrt (d) / (2*a);
                r2 = -b-sqrt (d) / (2*a);
                printf ("Roots are real and are %f %f", r1, r2);
        }
        else if (d= =0)
        {
                r1 = -b/(2*a);
                r2 = -b/(2*a);
                printf ("Roots are equal and are %f %f", r1, r2);
        }
        else
                printf("Roots are imaginary");
getch ( );
}
```

5) Testing

```
Case 1: Enter a,b,c values : 1  4  3
                r1 = -1
                r2 = -3
Case 2: Enter a,b,c values : 1  2  1
                r1 = -1
                r2 = -1
Case 3: Enter a,b,c values : 1   1   4
                Roots are imaginary
```

ALGORITHM:

- It is a step – by – step procedure for solving a problem
- If algorithm is written in English like sentences then it is called as ‘PSEUDO CODE’

Properties of an Algorithm

An algorithm must posses the following 5 properties. They are

1. Input
2. Output

3. Finiteness
4. Definiteness
5. Effectiveness

1. **Input :** An algorithm must have zero (or) more number of inputs
2. **Output:** Algorithm must produce one (or) more number of outputs
3. **Finiteness :** An algorithm must terminate in countable number of steps
4. **Definiteness:** Each step of the algorithm must be stated clearly
5. **Effectiveness:** Each step of the algorithm must be easily convertible into program statements

<u>Example</u>

Algorithm for finding the average of 3 numbers

```
1. start
2. Read 3 numbers a,b,c
3. Compute sum = a+b+c
4. compute avg = sum/3
5. Print avg value
6. Stop
```

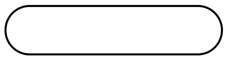
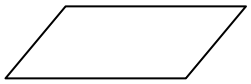
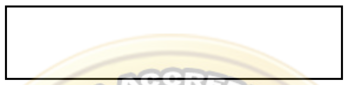
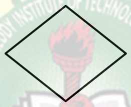
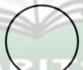
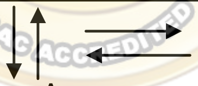
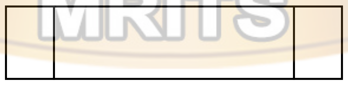
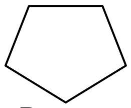
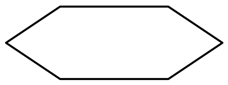
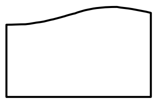
FLOW CHART

Diagramatic representation of an algorithm is called flow chart

<u>Advantages of flow chart</u>

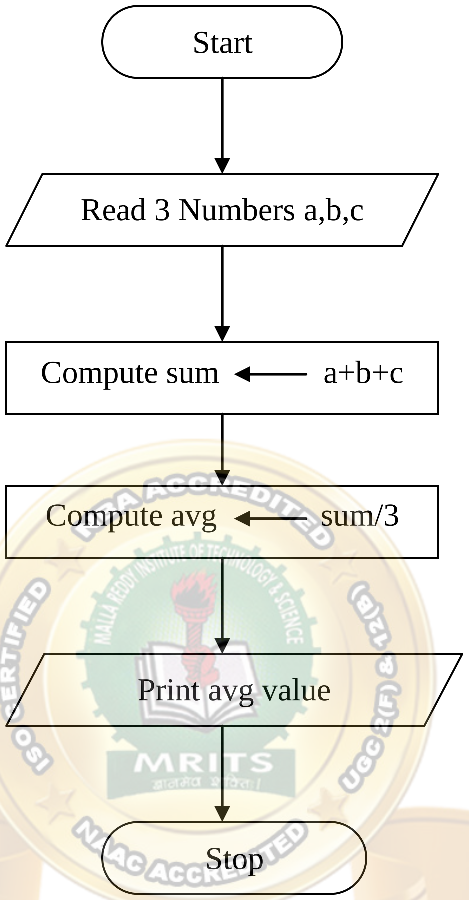
- It is very easy to understand because the reader follows the process quickly from the flowchart instead of going through the text.
- It is the best way of representing the sequence of steps in an algorithm
- It gives a clear idea about the problem
- Various symbols are used for representing different operations
- Arrows are used for connecting the symbols and show the flow of execution

Symbols used in flowchart:

Name	Symbol	Purpose
Terminal	 oval	Start /stop/begin/end
Input / output	 Parallelogram	Input/output of data
Process	 Rectangle	Any processing to be performed can be represented
Decision box	 Diamond	Decision operations that determine which of the alternative paths to be followed
Connector	 Circle	Used to connect different parts of flowchart
Flow	 Arrows	Joins 2 symbols and also represents flow of execution
Pre defined process	 Double sided rectangle	Modules (or)subroutines specified else where
Page connector	 Pentagon	Used to connect flowchart in 2 different pages
For loop symbol	 Hexagon	Shows initialization, condition and incrementation of loop variable.
Document	 Printout	Shows the data that is ready for printout

Example

Flowchart for finding the average of 3 numbers



Importance of 'C' Language

1. It is a **robust language**, whose rich set of built-in functions and operations can be used to write any complex program

2. It is a **middle level language** because the 'C' compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages.

3. 'C' Programs are **efficient and fast**

4. C is highly **portable**, that is 'C' programs written on one computer can be run on another with little (or) no modification.

5. 'C' Language is well suited for **structured programming**, thus requiring the user to think of a problem in terms of function modules (or) blocks.

6. 'C' program has the **ability to extend itself**.

- It was named ‘C’ because it is an offspring of BCPL (Basic Combined Programming Language) which was popularly called ‘B’ language.

General form of a ‘C’ program

```
/* documentation section */
preprocessor directives
global declaration
main ( )
{
        local declaration
        executable statements
}
returntype function name (argument list)
{
        local declaration
        executable statements
}
```

Example:

```
/* Author : Ramu
  Aim : Program for finding circumference of circle*/
#include<stdio.h>
#include<conio.h>
#define PI 3.1415
main ( )
{
        float c, r;
        clrscr ( );
        printf (“enter radius of circle”);
        scanf (“%f”, &r);
        c = 2 * PI * r;
        printf (“Circumference = %f”, c);
        getch ( );
}
```

‘C’ LANGUAGE ELEMENTS

‘C’ program contains several elements which are present in structure of ‘C’ program. They are

1) Comment lines
2) Preprocessor directives
3) Variable declaration & data types
4) Executable statements

1. Comment lines

- ❖ In ‘C’, comment lines are placed in “ /*   */”
- ❖ Single line and multiple lines are enclosed in /* and */
- ❖ Comment lines are ignored by the compiler
- ❖ The documentation section is enclosed in comment lines
- ❖ Documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.

2. Preprocessor directives

- ❖ It consists of a) link section
  b) Definition Section
- ❖ The link section provides instructions to the compiler to link functions from the system library

  eg : #include < stdio.h>
- ❖ The definition section defines all symbolic constants

  eg : #define PI 3.1415
- ❖ Preprocessor directive must start with ‘#’ (hash) symbol.

3. Variable declaration & data types

Variable

- ❖ It is the name given to a memory location that may be used to store a data value
- ❖ A variable may take different values at different times during execution

- ❖ A variable name may be chosen by the programmer in a meaningful way so as to reflect its function (or) nature in the program

Eg: sum, avg , total etc.

Rules for naming a variable

1) They must begin with a letter
2) The length of the variable must not exceed 31 characters in ANSI standard. But first eight characters are treated as significant by many compilers
3) Upper and lowercase characters are different

   Eg: total, TOTAL, Total are 3 different variables
4) The variable name should not be a keyword
5) White space is not allowed

Data Types

- ❖ Data type specifies the set of values and the type of data that can be stored in a variable.
- ❖ They allow the programmer to select the type appropriate to the needs of application.

Types :

1) Primary data types
2) Derived data types

1. Primary data types

'C' compilers support 4 fundamental data types

They are

1) integer
2) character
3) Floating – point
4) Double precision floating point

PRIMRARY DATA TYPES

Integral Type

Integer	
Signed	Unsigned
int short int long int	unsigned int unsigned short int unsigned long int

character
signed char unsigned char

Floating point type		
float	double	long double

1. Integral data type

- ❖ Integral data types are used to store whole numbers and characters
- ❖ It is further classified into
  - a) integer data type
  - b) character data type

a) Integer data type

- ❖ This data type is used to store whole numbers
- ❖ It has 3 classes of integer storage namely, short int, int and long int in both signed and unsigned forms

Integer Data type			
Type	**size (in bytes)**	**Range**	**Control string**
Short int (or) signed short int	1	-128 to 127	% h
Unsigned short int	1	0 to 255	% uh
int (or) signed int	2	-32768 to 32767	% d or %i
unsigned int	2	0 to 65535	% u
Long int (or) signed long int	4	-2147483648 to 2147483647	%ld
Unsigned long int	4	0 to 4294967295	%lu

b) <u>character data type</u>

- ❖ This data type is used to store characters
- ❖ These characters are internally stored as integers
- ❖ Each character has an equivalent ASCII value

  eg: 'A' has ASCII value 65

Character data type			
Type	**Size (in bytes)**	**Range**	**Control string**
Char (or) signed char	1	- 128 to +127	%c
Unsigned char	1	0 to 255	%c

2. <u>Floating – point Data types</u>

- ❖ It is used to store real numbers (i.e., decimal point numbers).
- ❖ For 6 digits of accuracy, 'float' is used.
- ❖ For 12 digits of accuracy, 'double' is used.
- ❖ For more than 12 digits of accuracy, 'long double' is used..

Floating point data type			
Type	**Size (in bytes)**	**Range**	**Control string**
float	4	3.4 E – 38 to 3.4 E + 38	%f
double	8	1.7 E – 308 to 1.7 E +308	%lf
long double	10	3.4 E – 4932 to 1.1 E +4932	%Lf

Variable declaration

<u>Syntax:</u>

Datatype v1,v2,… vn;

Where v1, v2,...vn are names of variables

eg: int sum;

float a,b;

- ❖ Variable can be declared in 2 ways
  1. local declaration
  2. global declaration

- ‘local declaration’ is declaring a variable inside the main block and its value is available within that block
- ‘global declaration’ is declaring a variable outside the main block and its value is available through out the program.
- Eg :

```
 int a, b;          /* global declaration*/
 main ( )
{
  int c;  /* local declaration*/
-         -         -
}
```

EXECUTABLE STATEMENTS:

- A ‘C’ program contains executable statements
- The ‘C’ compiler translates the executable statements into machine language
- The machine language versions of these statements are executed by the compiler when a user runs program

Types

1) Input – output statements
2) Assignment statements

1. Input – output statements

- Storing a value into memory is called ‘ input operation’.
- After executing the computations, the results are stored in memory and the results can be displayed to the user by ‘output operation’
- All input / output operations in ‘C’ are performed using input / output functions
- The most common I/O functions are supplied as part of the ‘C’ standard I/O library through the preprocessor directive # include<stdio.h>
- Most commonly used I/O functions are
  a) printf ( )
  b) scanf ( )

a) printf ( ) function:

Syntax:

printf(“format string”, print list);

e.g.: printf (“average of 3 numbers = %f”,avg);

* The printf ( ) function displays the value of its format string after substituting the values of the expressions in the print list.
* It also replaces the escape sequences such as ‘\n’ by their meanings.

b) scanf ( ) function

Syntax:

scanf (“format string”, input list);

e.g.: scanf (“%d %f”, &a, &b);

➢ The scanf ( ) function copies into memory data typed at the keyboard by the program user during program execution.

➢ The input list must be preceded by ampersand (&)

2) **<u>Assignment statements</u>**

❖ The assignment statements stores a value (or) a computational result in a variable and is used to perform most arithmetic operations in a program.

❖ **Syntax**: variable=expression

❖ e.g.:

1. c = a+b;
2. avg = sum/3;
3. r1 = (b*b – 4 * a*c);

❖ The variable before the assignment operator is assigned the value of the expression after it.

❖ The previous value of variable is destroyed

<u>EXPRESSIONS</u>

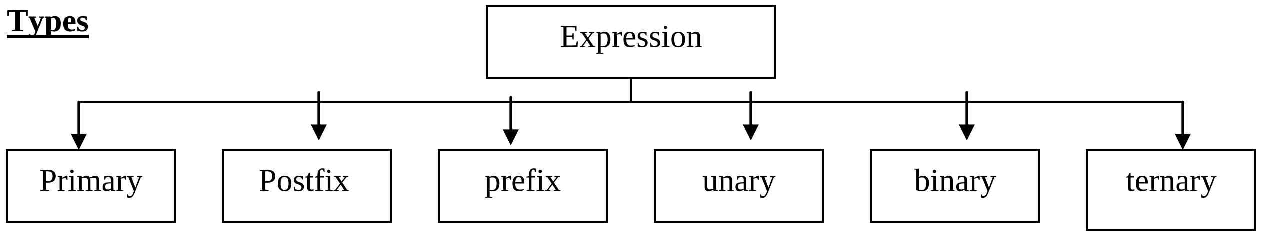
❖ **Def**: An expression is a combination of operators and operands which reduces to a single value

❖ An operand is a data item on which an operation is performed.

❖ An operator indicates an operation to be performed on data eg; z = 3+2*1

z = 5

Types



1. Primary expressions

The operand in the primary expression can be a name, a constant or any parenthesized expression

E.g.: c = a+ (5*b);

2. Postfix expressions:

The operator will be after the operands in a postfix expression

Eg:

ab+

3. Prefix expressions

The operator is before the operand in a prefix expression.

Eg:

+ab

4. **unary expression:**

It contains one operator and one operand

eg: a++, --b

5. Binary expression

It contains 2 operands and one operator

Eg: a+b, c-d

6. Ternary expression

It contains 3 operands and one operator

Eg ; Exp1? Exp2 :Exp3

If Exp1 is true ,Exp2 is executed. otherwise Exp3 is executed.

OPERATORS AND EXPRESSIONS

- Operator performs an operation on data
- Operators are classified into
  1. Arithmetic operators.
  2. Relational operators.
  3. Logical operators.
  4. Assignment operators.
  5. Increment and decrement operators.
  6. Bitwise operators.
  7. Conditional operators.
  8. Special operators.

1). Arithmetic operator

- These operators are used for numerical calculations (or) to perform arithmetic operations like addition, subtraction etc.

Operator	Description	Example	a =20, b=10	output
+	Addition	a+b	20+10	30
-	Subtraction	a-b	20-10	10
*	Multiplication	a*b	20*10	200
/	Division	a/b	20/10	2 (quotient)
%	Modular division	a%b	20%10	0 (remainder)

- Program:

```
main ( )
{
        int a= 20, b = 10;
        printf (“ %d”, a+b);
        printf (“ %d”, a-b);
```

Output

```
30
10
200
2
0
```

```
            printf (" %d", a*b);
            printf (" %d", a/b);
            printf (" %d", a%b);
    }
```

2).Relational operators :

❖ These are used for comparing two expressions.

Operator	Description	Examble	a =10, b=20	output
<	less than	a<b	10<20	1
<=	less than (or) equal to	a<=b	10< = 20	1
>	greater than	a>b	10>20	0
>=	greater than (or) equal to	a>=b	10> =20	0
= =	equal to	a= =b	10 = = 20	0
! =	not equal to	a! = b	10 ! =20	1

❖ The output of a relational expression is either true (1) (or) false (0)

❖ **Program**

```
main ( )
{
        int a= 10, b = 20;                         Output
        printf (" %d", a<b);                         1
        printf (" %d", a<=b);                        1
        printf (" %d", a>b);                         0
        printf (" %d", a>=b);                        0
        printf (" %d", a = =b);                      0
        printf (" %d", a ! =b);                      1

}
```

3. Logical Operators

❖ These are used to combine 2 (or) more expressions logically

- They are logical AND (&&) logical OR (||) and logical NOT (!)

Logical AND (&&)

exp1	**exp2**	**exp1&&exp2**
T	T	T
T	F	F
F	T	F
F	F	F

Logical OR(||)

exp1	**exp2**	**exp1||exp2**
T	T	T
T	F	T
F	T	T
F	F	F

Logical NOT (!)

exp	**!(exp)**
T	F
F	T

Operator	**Description**	**Example**	**a =10, b=20,c=30**	**output**
&&	logical AND	(a>b) && (a<c)	(10>20) & & (10<30)	0
||	logical OR	(a>b) | | (a<c)	(10>20) ||(10<30)	1
!	logical NOT	! (a>b)	! (10>20)	1

Program:

```
main ( )
{
        int a= 10, b = 20, c= 30;
        printf (“ %d”, (a>b) && (a<c));
        printf (“ %d”, (a>b) | | (a<c));
```

Output

```
0
1
1
```

```
        printf (“ %d”, ! (a>b));
}
```

4. Assignment operators

- It is used to assign a value to a variable

Types

1) simple assignment   2)compound assignment

Operator	Description	Example
=	Simple assignment	a=10

+ =, - =, * =, / =, %=	Compound assignment	a+=10 → a=a+10
		a-=10 → a=a-10

Program:

```
main ( )
{       int a= 10,;
        printf (“ %d”, a);
        printf (“ %d”, a+ =10);
}
```

Output

```
10
20
```

5. Increment and decrement operator:

a) Increment operator (++):

- It is used to increment the value of a variable by 1
- 2 types : i) pre increment
  ii) post increment
- increment operator is placed before the operand in preincrement and the value is first incremented and then operation is performed on it.

eg: z = ++a; < a= a+1
z=a

- increment operator is placed after the operand in post increment and the value is incremented after the operation is performed

eg: z = a++; < z=a
a= a+1

Program

```
main  ( )
{
        int a= 10, z;
        z= a++;
        printf (“z= %d”, z);
```

Output

```
z=10
```

Output

```
z=11
```

```
main ( )
{
        int a= 10, z;
        z= ++a ;
        printf (“z= %d”, z);
        printf (“ a=%d”, a);
}
```

b) <u>Decrement operator : (- -)</u>

- ❖ It is used to decrement the values of a variable by 1

  2 types : i) pre decrement

  ii) post decrement

- ❖ decrement operator is placed before the operand in predecrement and the value is first decremented and then operation is performed on it.

eg: z = - - a; $<$ a= a-1
z=a

- ❖ decrement operator is placed after the operand in post decrement and the value is decremented after the operation is performed

eg: z = a--; $<$ z=a
a= a-1

<u>Program:</u>

```
main ( )
{
        int a= 10, z;
        z= --a;
        printf (“z= %d”, z);
        printf (“ a=%d”, a);
}
```

<u>Output</u>

```
z=9
a =9
```

```
main  ( )
{
        int a= 10, z;
        z= a--;
        printf (“z= %d”, z);
        printf (“a=%d”, a);
}
```

<u>Output</u>

```
z=10
a = 9
```

6. Bitwise Operator

Unlike other operators, bitwise operators operate on bits (i.e. on binary values of on operand)

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
~	One's complement

Bitwise AND

a	b	a &b
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise XOR

a	b	a ^b
0	0	0
0	1	1
1	0	1
1	1	0

eg: let a= 12, b=10

a&b

	8	4	2	1
a =12	1	1	0	0
b =10	1	0	1	0
a &b	1	0	0	0

a&b = 8

a | b

	8	4	2	1
a =12	1	1	0	0
b =10	1	0	1	0
a | b	1	1	1	0

a | b = 14

	8	4	2	1

a =12	1	1	0	0
b =10	1	0	1	0
a ^b	0	1	1	0

a ^ b

a ^ b = 6

Program

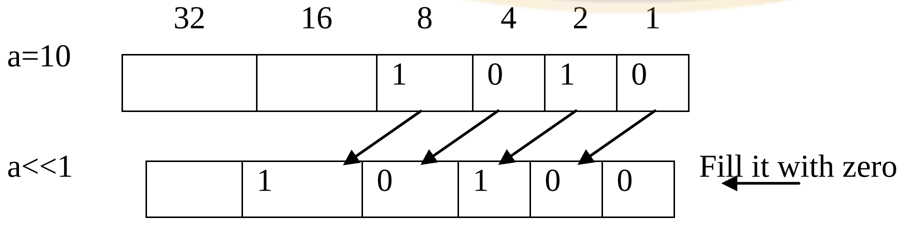
```
main ( )
{
        int a= 12, b = 10;
        printf (“ %d”, a&b);
        printf (“ %d”, a| b);
        printf (“ %d”, a ^ b);
}
```

Output

```
8
14
6
```

Left Shift

- ❖ If the value of a variable is left shifted one time, then its value gets doubled
- ❖ eg: a = 10 then a<<1 = 20

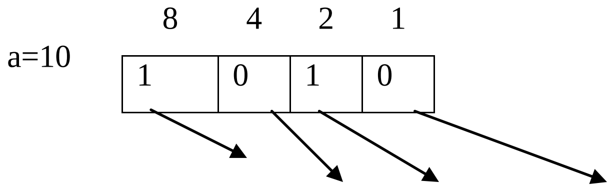


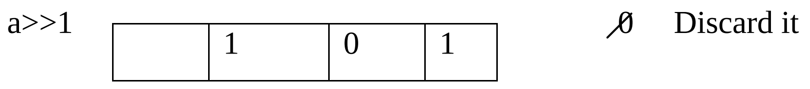
a<<1 = 20

Right shift

If the value of a variable is right shifted one time, then its value becomes half the original value

- ❖ eg: a = 10 then a>>1 = 5

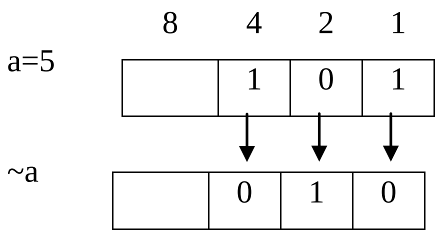




a>>1 = 5

<u>Ones complement</u>

- ❖ It converts all ones to zeros and zeros to ones
- ❖ Eg: a = 5 then ~a=2 [only if 4 bits are considered]



~a = 2

Program

```
main ( )
{
        int a= 20, b = 10,c=10;
        printf (" %d", a<<1);
        printf (" %d", b>>1);
        printf (" %d", ~c);
}
```

<u>Output</u>

```
40
5
11
```

<u>Signed</u>

1's complement = - [give no +1]

Eg : ~10 = - [10+1] = -11

~-10 = - [-10+1] = 9

unsigned

1's complement = [65535 – given no]

<u>7. Conditional operator (? :)</u>

- ❖ It is also called ternary operator
- ❖ **Syntax:**

  exp1 ? exp2 : exp3
- ❖ if exp1 is true exp2 is evaluated, otherwise exp3 is evaluated

- it can also be represented in if – else form

```
     if (exp1)
           exp2;
     else
           exp3;
```

Program

```
main ( )
{          int z;
           z = (5>3) ? 1:0;
           printf (“%d”,z);
}
```

Output

```
1
```

8) Special operations

Some of the special operations are comma, ampersand(&), size of operators.

a) Comma: (,)

It is used as separator for variables

eg; a=10, b=20

b) Address:(&)

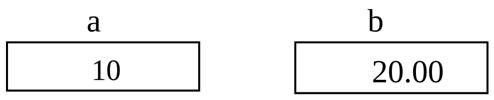
It is used to get the address of a variables.

c) Size of ( ) ;

It is used to get the size of a data type of a variable in bytes.

Program:

```
main (  )
{
     int   a=10;
      float  b=20 ;
      printf (“ a= %d     b=%f”, a,b );
      printf (“ Address  of a =%u “ , &a ) ;
      printf (“ Address of b =%u” ,&b ) ;
      printf (“Size of a = %d ” , sizeof (a) ) ;
      printf ( “Size of b = %d ”, sizeof (b) ) ;
}
```



Out Put :

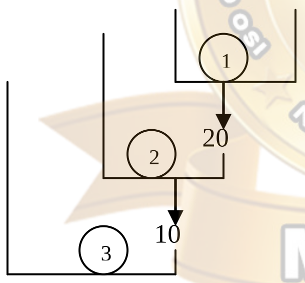
```
 a=10   b=20.00                                        1234                5678
Address  of a  =1 2 3 4 }
Address of b  = 5 6 7 8 }     Only for this example

 Size of a = 2  bytes
 Size of b = 4 bytes
```

EXPRESSION EVALUATION, PRECEDENCE AND ASSOCIATIVITY

- ❖ Expressions are evaluated by the 'C' compiler based on precedence and associativity rules.
- ❖ If an expression contains different priority operators then precedence rules are considered.
- ❖

Eg: C = 30 - 10 * 2

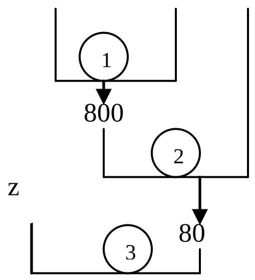


Here, 10*2 is evaluated first since '*' has more priority than '-' and '='

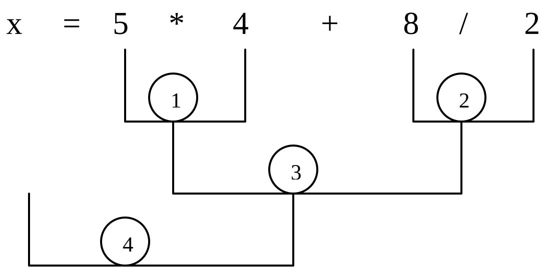
- ➢ If an expression contains same priority then assoiciativity rules are considered i.e. left right (or right to left)

eg: z= a*b/c

z = 40 * 20 / 10 Here '*' and '/' have same priority so, left to right associativity is considered



Eg: x =5 *4 + 8/2



x = 24

Parenthesis has highest priority and comma has least priority among operators

Type Conversions

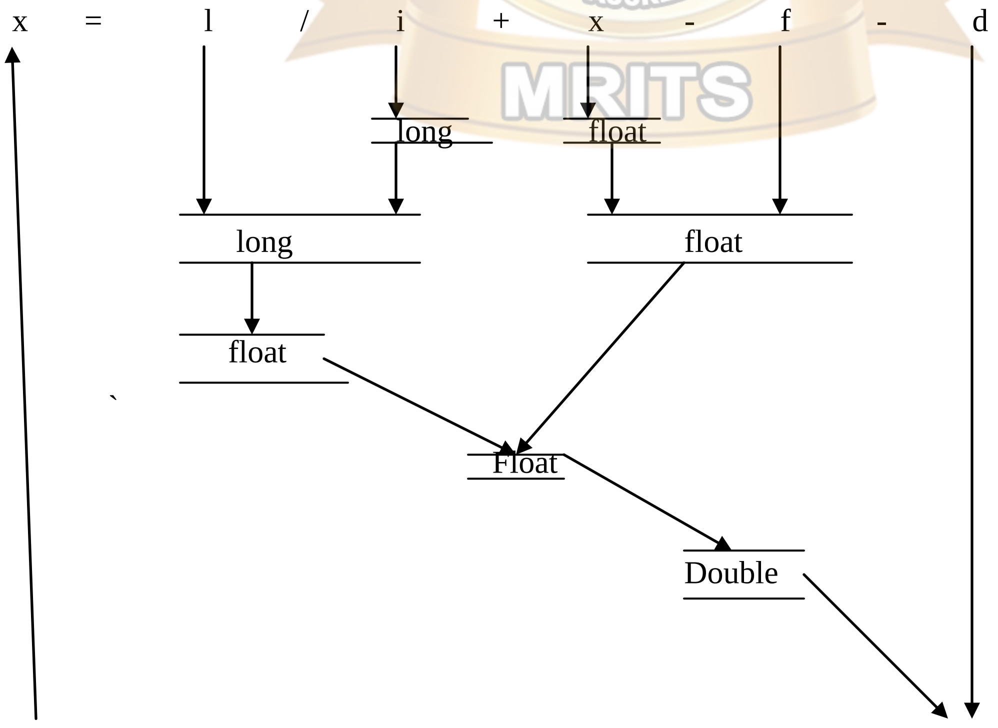
Converting one data type into another is the concept of type conversion

2 types

1. Implicit type conversion
2. Explicit type conversion

❖ Implicit type conversion is automatically done by the compiler by converting smaller data type into a larger data type.

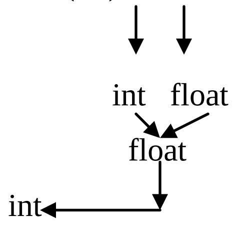
Eg:
```
int i,x;
float f;
double d;
long int l;
```



$$\begin{tikzcd} \text{int} & & & \text{double} \arrow[lll] \end{tikzcd}$$

Here, the above expression finally evaluates to a'double' value

Explicit type conversion is done by the user by using (type) operator

Eg:
```
int a,c;
float b;
c = (int) a + b
```


Here, the resultant of 'a+b' is converted into 'int' explicitty and then assigned to 'c'

Program:

```
main  ( )
{                                                    Output
        printf ("%d", 5/2);                            2
        printf ("%f", 5.5/2);                          2.75
        printf ("%f", (float) 5/2);                    2.5
}
```

STORAGE CLASSES

There are 4 storage classes (or) storage class specifiers supported in 'C' They are:

1) auto
2) extern
3) static
4) register

1. automatic variables / Local variables.

- Keyword : auto
- These are also called local variables
- Scope
    - Scope of a local variable is available within the block in which they are declared.

- These variables are declared inside a block

- Default value: garbage value

eg:

```
main ( )
{
        auto int i=1;
        {
                auto int i=2;
                {
                        auto int i=3;
                        printf ("%d",i)
                }
                printf("%d", i);
        }
        printf("%d", i);
}
```

Output

```
3
2
1
```

2. global Variables / external variables

- Keyword : extern
- These variables are declared outside the block and so they are also called global variables
- Scope:

  Scope of a global variable is available throughout the program.
- Default value: zero

eg:

```
    extern int i =1; /* this 'i' is available throughout program */
    main ( )
    {
            int i = 3; /* this 'i' available only in main */
            printf ("%d", i);
            fun ( );
    }
```

Output

```
3
1
```

```
fun ( )
{
        printf ("%d", i);
}
```

3) Static variables

- Keyword : static
- Scope

Scope of a static variable is that it retains its value throughout the program and in between function calls.

- Static variables are initialized only once.
- Default value: zero

eg:

```
main ( )
{
        inc ( );
        inc ( );
        inc ( );
}
inc ( )
{
        static int i =1;
        printf ("%d", i);
        i++;
}
```

Output

```
1       2       3
```

```
main ( )
{
        inc ( );
        inc ( );
        inc ( );
}
inc ( )
{
        auto int i=1;
        printf ("%d", i);
        i++;
}
```

Output

```
1       1       1
```

4. Register variables

- Keyword : register
  - Register variable values are stored in CPU registers rather than in memory where normal variables are stored.
  - Registers are temporary storage units in CPU
  - They allow faster access time for register variables than normal variables

eg:

```
main ( )
{
        register int i;
        for (i=1; i< =5; i++)
        printf (”%d  ”,i);
}
```

Output

```
1       2       3       4       5
```

Scope rules

- Scope rules relate to the accessibility, period of existence and boundary of usage of variables.

1) Scope rules related to statement Blocks :

- Block is set of statement enclosed in curly braces
- Variables declared in a block are accessible and usable within that block and doesnot exist outside it

```
eg:     main ( )
        {
                {
                        int i = 1;              ‘i' is available within this
                        printf (“%d”,i);        block only
                }
                {
                        int j=2;                ‘j' is available within this
                        printf(“%d”,j);         block only
                }
        }
```

output

```
1       2
```

- Even if the variables are redeclared in their respective blocks and with the same name, they are considered differently

```
main ( )
```

```
{
        {
                int i = 1;                  /* Both i's are declared in different
                printf ("%d",i);            blocks so, they are treated differently
        }                                   even though they have same name */
        {
                int i =2;
                printf ("%d",i);
         }
}
```

Output

```
1       2
```

- ❖ redeclaration of variables within the blocks bearing the same names as those in the outer block masks the outer block variables while executing the inner blocks.

eg:

```
main ( )
        {
                int i = 1;
        {                                   /* inner block variable dominates
                int i = 2;                  outer block variable with same name
                printf ("%d",i);            */
        }
        }
        Output : 2
```

- ❖ Variables declared outside the inner blocks are accessible to the nested blocks, provided these variable are not declared within the inner block

```
main ( )
{
                int i = 1;                  /*  'i' is available within the inner
        {                                   block also */
                int j = 2;
```

```
                    printf (“%d”,j);
                    printf (“%d”,i);
            }
    }
```

Output

```
    2       1
```

2. Scope rules related to functions

- Function is a self contained block that performs a particular task.
- Variables declared within the function body are called local variables
- These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main functions also
- The existence of local variables ends when the function completes its specific task and returns to the calling point.

eg:

```
main ( )
{
    int a=10, b = 20;
    printf (“before swapping a=%d, b=%d”, a,b);
    swap (a,b);
    printf (“after swapping a=%d, b=%d”, a,b);
}
swap (int a, int b)
{
    int c;
    c=a;
    a=b;
    b=c;
}
```

Output:

```
    Before swapping a=10, b=20
    After swapping a = 10, b=20
```

- Variables declared outside the function body are called global variables.
- These variables are accessible by any of the functions

eg:

```
include<stdio.h>
int  a=10, b = 20;
main()
{
        printf (“before swapping a=%d, b=%d”, a,b);
        swap ( );
        printf (“after swapping a=%d, b=%d”, a,b);
}
swap ( )
{
        int c;
        c=a;
        a=b;
        b=c;
}
```

Output

```
 Before swapping a = 10, b =20
After swapping a = 20, b = 10
```

CONTROL STATEMENTS :

Decision statements :

These are used to make a decision among the alternative paths

They are

1. simple – if statement
2. if – else statement

3. nested – if else statement
4. else – if ladder
5. switch statement

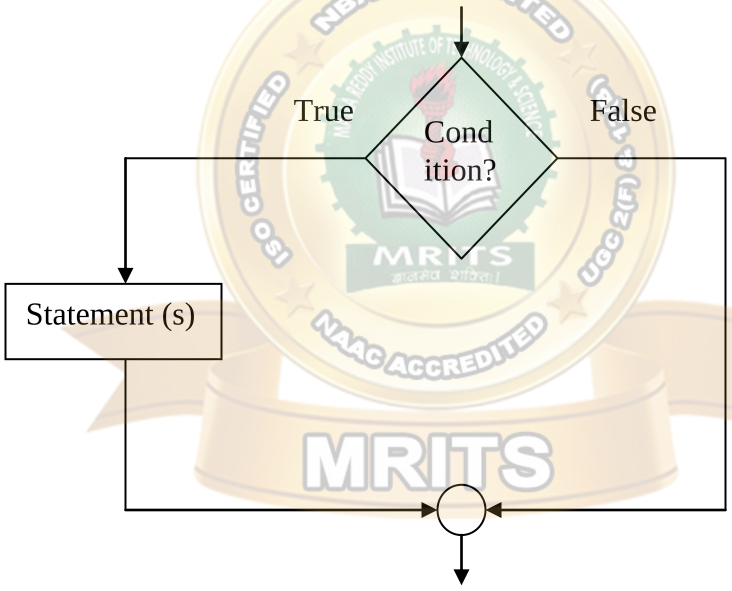
1. <u>Simple – if statement</u>

- ‘if’ keyword is used to execute a set of statements when the logical condition is true

<u>Syntax :</u>

```
if (condition)
{
        Statement (s)
}
```

<u>Flow chart</u>



Program /* checking whether a number is greater than 50 */

```
main ( )
{
        int a;
        printf (“enter a number”);
        scanf (“%d”, &a);
        if (a>50)
                printf (“%d is greater than 50”, a);
}
```

Output

```
1) enter a number 60                2) enter a number 20
60 is greater than 50                no output.
```

2. if else statement

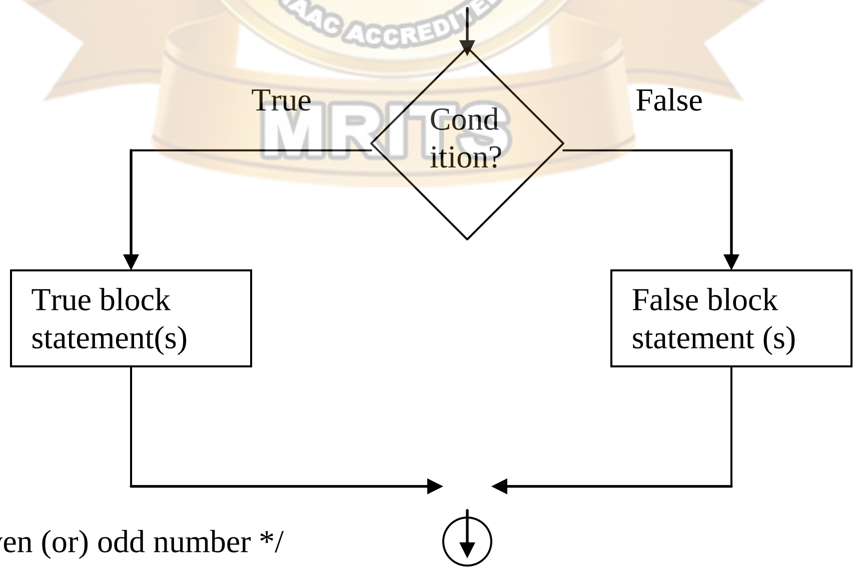
- If –else statement takes care of true as well as false conditions
- ‘true block’ is executed when the condition is true and ‘false block’ (or) ‘else block’ is executed when the condition is false.

Syntax:

```
    if (condition)
    {
            True block statement(s)
    }
    else

    {
            False block statement(s)
    }
```

Flow chart



Program

```
/* checking for even (or) odd number */
main ( )
{
        int n;
printf (“enter a number”);
scanf (“%d”, &n);
if (n%2 ==0)
        printf (“%d is even number”, n);
else
        printf( “%d is odd number”, n);
}
```

Output

```
1) enter a number 10              2) enter a number 5
10 is even number                    5 is odd number
```

3. Nested if - else statemen

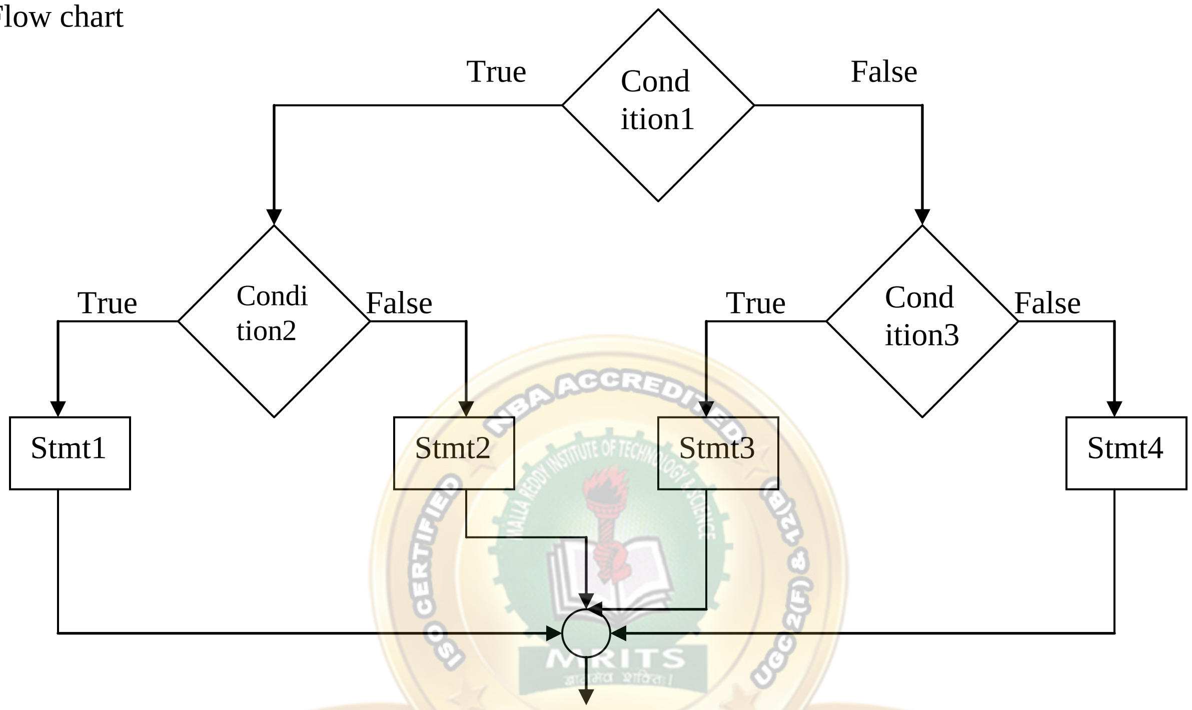
- A ‘nested if’ is an if statement that is the object of either if (or) an else
- ‘if’ is placed inside another if (or) else

Syntax:

```
if (condition1)
{
        if (condition2)
                stmt1;
        else
                stmt2;
}
else
{
        if (condition3)
                stmt3;
        else
                stmt4;
}
```

↓

Flow chart



Program /* largest of 3 numbers */

```
main ( )
{
        int a,b,c;
        printf ("enter 3 numbers");
        scanf ("%d%d%d", &a, &b, &c);
        if (a>b)
        {       if (a>c)
                printf ("%d is largest", a);
                else
                printf ("%d is largest", c);
        }
        else
        {
                if (b>c)
                printf ("%d is largest", b);
                else
                printf ("%d is largest", c);
        }
}
```

Output

```
enter 3 numbers = 10 20 30
30 is largest
```

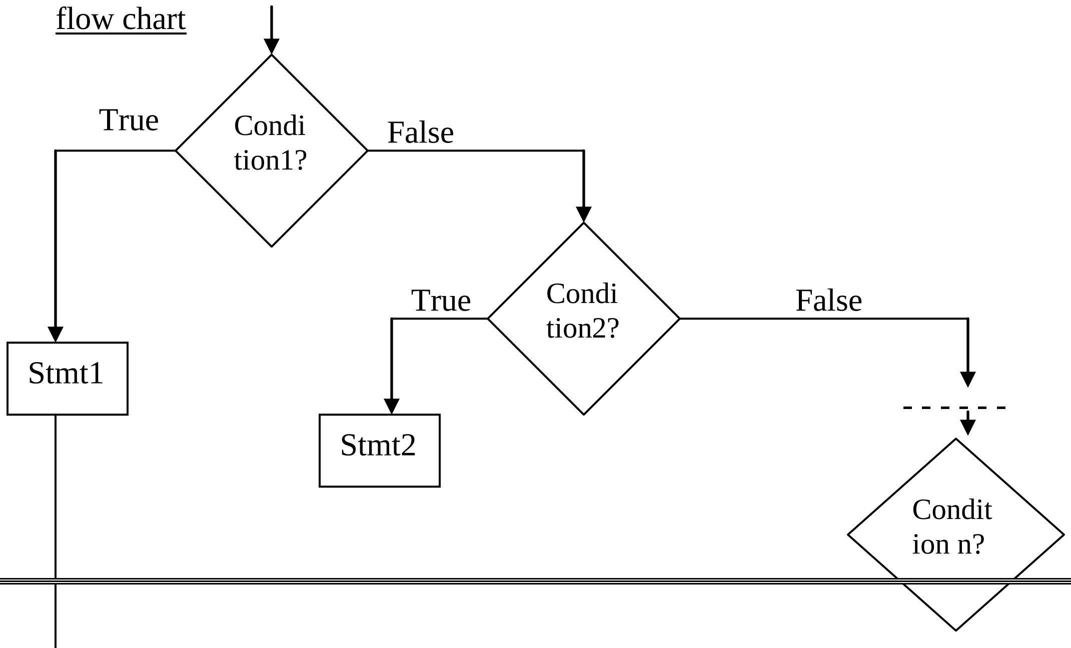
4. Else – if ladder

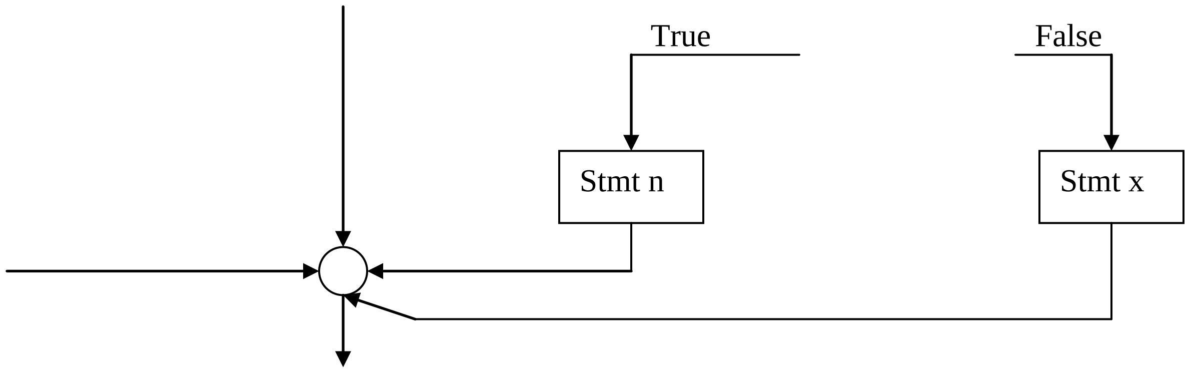
- This is the most general way of writing a multi-way decision

Syntax

```
if (condition1)
        stmt1;
else if (condition2)
              stmt2;
                - - - - -
                - - - - -
        else if (condition n)
                        stmtn;
                else
                        stmt x;
```

flow chart





Program /* finding roots of quadratic equation */

```
#include <math.h>
main ( )
{
        int a,b,c,d;
        float r1, r2
        printf (“enter a,b,c values”);
        scanf (“%d%d%d”, &a, &b, &c);
        d= b*b – 4*a*c ;
        if (d>0)
        {
                r1 = (-b+sqrt(d))  / (2*a);
                r2 = (-b-sqrt(d))  / (2*a);
                printf (“root1 = %f, root2 == %f”, r1, r2);
        }
        else  if (d= = 0)
        {
                r1 = -b  / (2*a);
                r2 = -b/ (2*a);
                printf (“root1 = %f, root2 = %f”, r1, r2);
        }
        else
```

```
                    printf ("roots are imaginary”);
}
```

Output

```
1) enter a,b, c values : 1 4 3
 Root 1 = -1
 Root 2 = -3
2)  enter a,b, c values : 1 2 1
Root 1 = -1
Root 2 = -1
3) enter a,b, c values : 1 2 3
 roots are imaginary
```

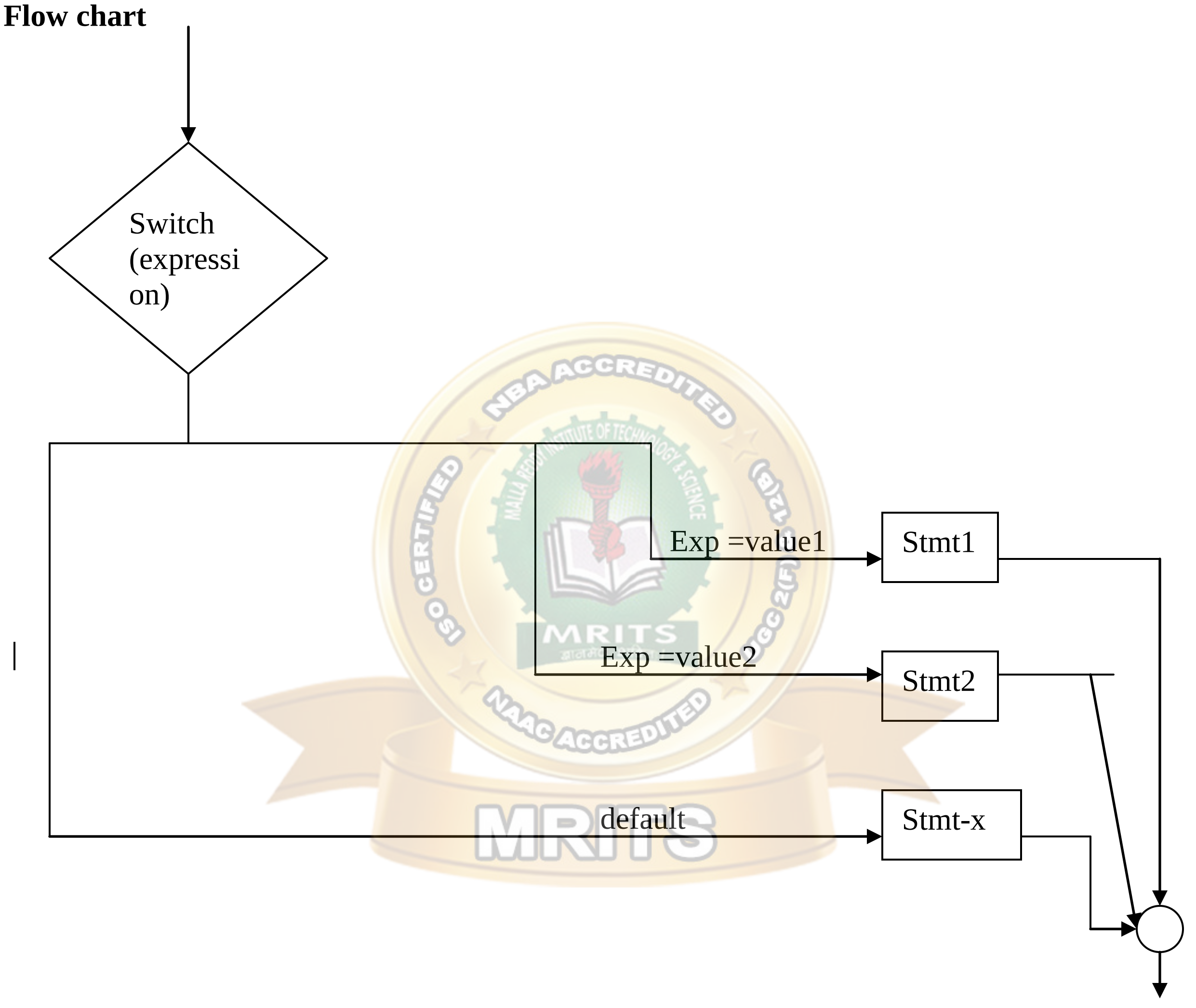
5. Switch statement

- It is used to select one among multiple decisions
- ‘switch’ successively tests a value against a list of integer (or) character constant.
- When a match is found, the statement (or) statements associated with that value are executed.

Syntax

```
switch (expression)
{
case value1 : stmt1;
                  break;
case value2 : stmt2;
                  break;
          - - - - - -
default : stmt – x;
}
```

Flow chart



Program

```
main ( )
{
        int n;
        printf (“enter a number”);
        scanf ("%d", &n);
        switch (n)
        {
                case 0 : printf ("zero")
                        break;
                case 1 : printf ('one");
                        break;
                default : printf ('wrong choice");
        }
}
```

Output

```
enter a number
1
one
```

<u>Loop control statements</u>

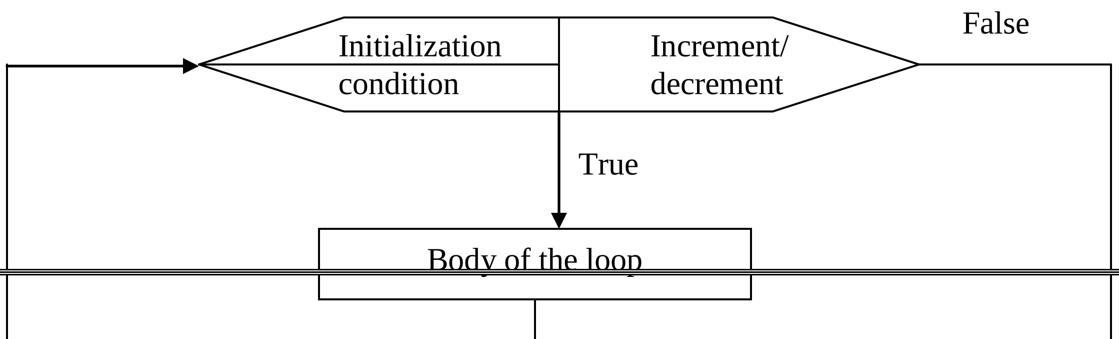
- These are used to repeat set of statements
- They are
  1) for loop
  2) while loop
  3) do-while loop

1) for loop

Syntax

```
        for (initialization ; condition ; increment / decrement)
        {
        body of the loop
        }
```

Flow chart



- for statement contains **3 parts**

i) **initialization** is usually an assignment statement that is used to set the loop control variable

ii) The **condition** is a relational expression that determines when the loop will exit.

iii) The **increment/decrement** part defines how the loop control variable will change each time loop is repeated.

iv) loop continues to execute as long as the condition is true.

v) Once the condition is false, program continues with the next statement after for loop.

Program

```
main( )
{
        int k;
        for (k = 1; k<=s; k++)
        {
                printf (”%d”,k);
        }
}
```

Output

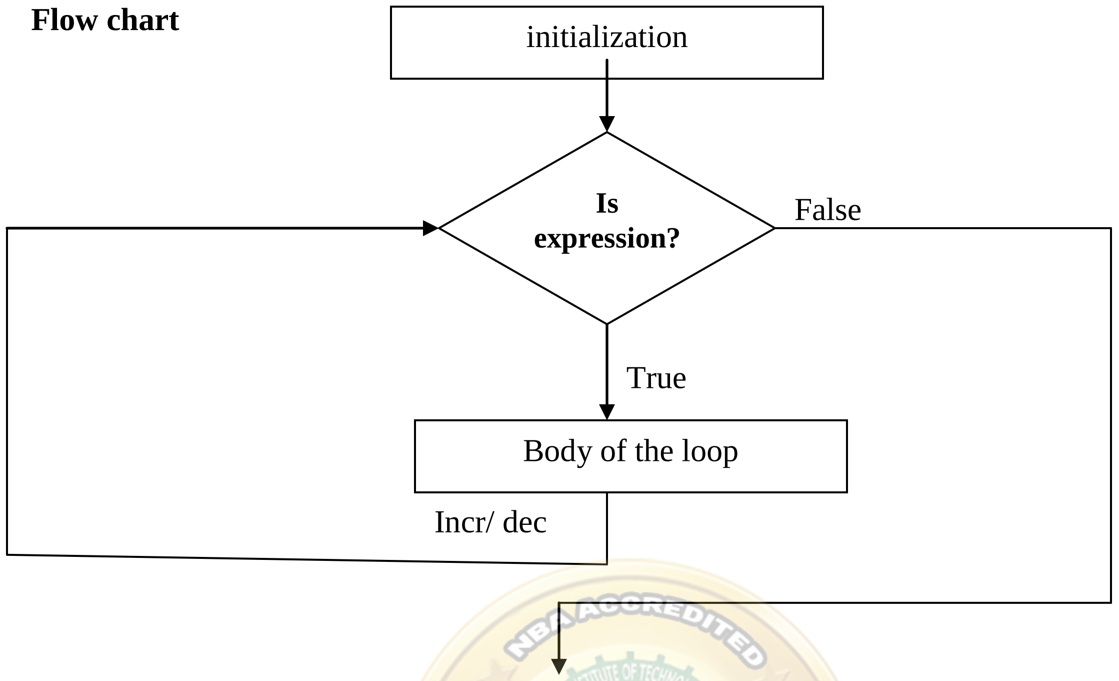
```
1
2
3
4
5
```

2) while loop

Syntax

```
        while (condition)
        {
                body of the loop
        }
```

Flow chart



- Intialization is done before the loop
- Loop continues as long as the condition is true
- Incrementation and decrementation part is done within the loop

Program

```
main( )
{
        int k;
        k = 1;
        while (k< = 5)
        {
                printf (”%d”,k);
                k++;
        }
}
```

Output

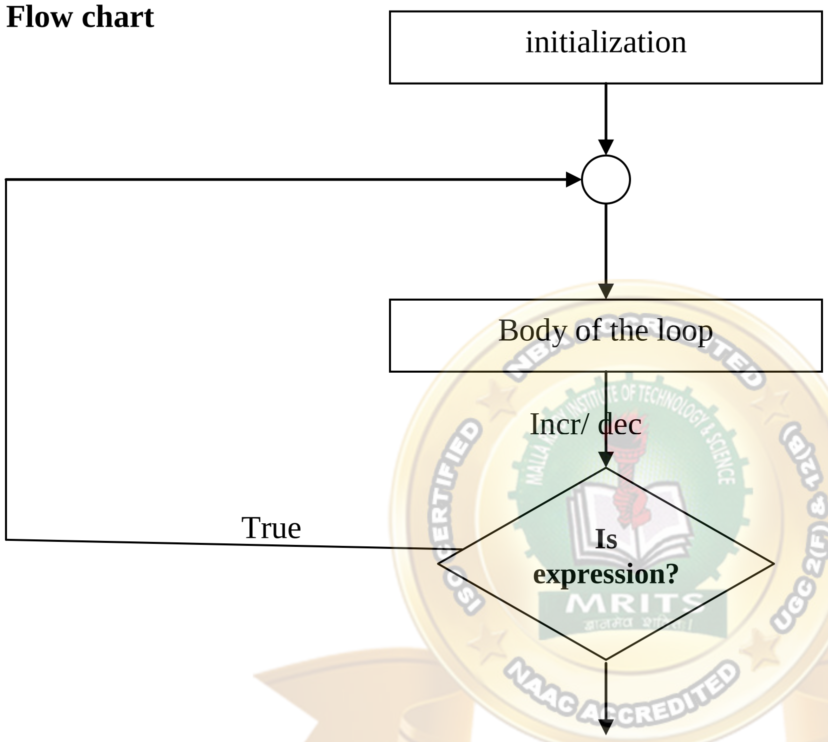
```
1
2
3
4
5
```

3) do-while loop

Syntax

```
        Initialization
        do
        {
                body of the loop
                inc/ dec
        } while (condition);
```

Flow chart



<u>Program</u>

```
main( )
    {
        int k;
        k = 1;
        do
        {
        printf ("%d",k);
        k++;
        } while (k <= 5);
    }
```

<u>Output</u>

```
1
2
3
4
5
```

<u>Nested for loops</u>

- In nested for loops one (or) more for statements are included in the body of the loop
- The number of iterations in this type of structure will be equal to number of iterations in the outer loop multiplied by the number of iterations in the inner loop

Program

```
main( )
{
        int i,j;
        for (i=1; i<=2; i++)
        {
                for (j=1;j<=2; j++)
                {
                printf (”%d”, i*j);
                }
        }
}
```

Output

```
1
2
2
4
3
6
```

Execution		i*j
i=1	j=1	1
	j=2	2
i=2	j=1	2
	j=2	4
i=3	j=1	3
	j=2	6

Other related statements

1) break
2) continue
3) goto

1) break

- It is a keyword used to terminate the loop (or) exit from the block
- The control jumps to next statement after the loop (or) block
- ‘break is used with for, while, do-while and switch statement
- When break is used in nested loops then only the innermost loop is terminated

Syntax

```
{       Stmt1;
        Stmt2;
```

```
                break;
                Stmt3;
                Stmt4;
        }
```

Program

```
main( )
{       int i;
        for (i=1; i<=5; i++)
        {
                printf (”%d”, i);
                if (i= =3)
                        break;
        }
}
```

Output

```
1
2
3
```

2) continue

- It is a keyword used for continuing the next iteration of the loop
- It skips the statements after the continue statement
- It is used with for, while and do-while

Syntax

```
        {
                Stmt1;
                Stmt2;
                continue;
                Stmt3;
                Stmt4;
        }
```

Program

```
main( )
{
        int i;
```

Output

```
1
3
4
5
```

```
        for (i=1; i<=5; i++)
        {
                if (i= =2)
                continue,
                printf("%d", i)
        }
}
```

3) goto

- It is used to after the normal sequence of program execution by transferring the control to some other part of program

Syntax

```
                Forward jump                              backward jump
goto label;  ──┐                          label :stmt  ◄──┐
      ----     │                          ----            │
      ----     │                          ----            │
      ----     │                          ----            │
label : stmt ◄─┘                          goto label;  ───┘
```

Program

```
main( )
{
      printf('Hello");
      goto l1;
      printf("How are");
      l1: printf("you");
}
```

Output

```
Hello

you
```

Command line arguments :

- ❖ An executable program that performs a specific task for operating system is called as command
- ❖ These commands are issued from the prompt of operating system.
- ❖ Some arguments are to be associated with the commands and hence these are called " command" line arguments. They are

1) argc ----- argument count
2) argv ----- argument vector

argc : it contains the total number of arguments passed from command prompt

argv : it is a pointer to an array of character strings which contains names of arguments. Each word is an argument

<u>for eg :</u>

c: |> sample. Exe $\underbrace{\text{hello how are you}}_{\text{arguments}}$

Here, argc = 5

argv[0] = sample.exe	argv[3] = are
argv[1] = hello	argv[4] = you
argv [2] = how	

<u>Program</u>

<u>Program for calling a function using pointer to function</u>

Program

```
main ( )
{
        int (*p) ( );
        clrscr ( );
        p = display;
        *(p) ( );
        getch ( );
}
display ( )
{
        printf(“Hello”);
}
```

<u>Output</u>

```
Hello
```

```
main ( )
{
        clrscr ( );
        display ( );
        getch( );
}
display ( )
{
        printf (“Hello”);
}
```

Programming For Problem Solving

ARRAYS

- **Array:** An array is a group of related data items that share a common name (or) Homogenous collection of data items that share a common name.
- A particular value in an array is identified using its "index number" or "subscript"

Advantage

- The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables the user to develop concise and efficient programs

Declaring for declaring array

Syntax : for declaring array:

```
datatype array_name [size];
```

Eg:

1. float height [50]

This declares 'height' to be an array containing 50 float elements

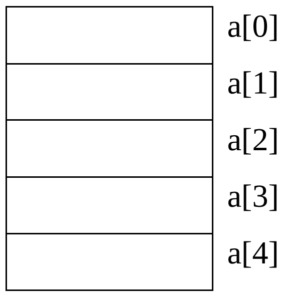
2. int group[10]

- This declares the 'group' as an array to contain a maximum of 10 integer constants
- Individual elements are identified using **" array subscripts"**
- While complete set of values are referred to as an array, individual values are called "elements"

Eg: To represent a set of 5 numbers by an array, it can be declared as follows

int a[5];

- Then computer reserves 5 storage locations each of 2 bytes.



- First element is identified by subscript 'zero' i.e., a[0] represents first element of the array.
- If there are 'n' elements in array then subscripts range from 0 to n-1

Initialization

To store values into an array it can be done as follows.

a[0] = 10;	10	a[0]
a[1] = 20;	20	a[1]
a[2] = 30;	30	a[2]
a[3] = 40;	40	a[3]
a[4] = 50;	50	a[4]

An array can also be initialized at the time of declaration as follows

int a[5] = { 10,20,30,40,50};

Types of arrays

Arrays are broadly classified into 3 types. They are

1) one – dimensional arrays
2) two – dimensional arrays
3) Multi – dimensional arrays

1. one – dimensional arrays

Syntax: datatype array name [size];

Eg: int a[5];

Initialization;

An array can be initialized in 2 ways.

a) compile time initialization
b) Runtime initialization

Program for compile time initialization and sequential access using for loop

main ( )

storing

a[0]	10
a[1]	20
a[2]	30
a[3]	40

a[4]	50

```
{
        int a[5] = {10,20,30,40,50};
        int i;
        clrscr ( );
        printf (“elements of the array are”);
        for ( i=0; i<5; i++)
                printf (“%d, a[i]);    ←
        getch ( );                     accessing
}
```

Output: Elements of the array are

```
        10      20      30      40      50
```

<u>Program for runtime initialization and sequential access using for loop</u>

```
main ( )
{
        int a[5],i;
        clrscr ( );
        printf (“enter 5 elements”);
        for ( i=0; i<5; i++)
                scanf(”%d”, &a[i]);   ───→ Storing / assigning values to element of an
                                                              array
        printf(“elements of the array are”);
        for (i=0; i<5; i++)
                printf(”%d ”, a[i]);  ───→ Accessing the elements of the aray
        getch ( );
}
```

<u>output</u>

```
        enter 5 elements 10     20      30      40      50
        elements of the array are :     10      20      30      40      50
```

<u>Note :</u>

❖ The output of compile time initialized program will not change during different runs of the program

- The output of run time initialized program will change for different runs because user is given a chance of accepting different values during execution.

2. <u>Two – dimensional arrays</u>

- These are used in situations where a table of values have to be stored (or) in matrices applications
- <u>Syntax :</u>

  datatype array_ name [rowsize] [column size];
- <u>Eg</u>: int a[5] [5];
- No of elements in array = rowsize *columnsize = 5*5 = 25

<u>Initialization :</u>

<u>Program for compile time initialization and sequential access using nested for loop</u>

```
main ( )
{
        int a[3][3] = {10,20,30,40,50,60,70,80,90};
        int i,j;
        clrscr ( );
        printf (“elements of the array are”);
        for ( i=0; i<3; i++)
        {
                for (j=0;j<3; j++)
                {
                        printf(“%d \t”, a[i] [j]);
                }
                printf(“\n”);
        }
        getch ( );
}
```

a[0] [0] 10	a[0] [1] 20	a[0] [2] 30
a[1] [0] 40	a[1] [1] 50	a[1] [2] 60
a[2] [0] 70	a[2] [1] 80	a[2] [2] 90

<u>output</u>

```
        elements of the array are:
                10      20      30
                40      50      60
```

```
          70      80      90
```

Program for runtime initialization and sequential access using nested for loop

```
main ( )
{
        int a[3][3] ,i,j;
        clrscr ( );
        printf ("enter elements of array");
        for ( i=0; i<3; i++)
        {
                for (j=0;j<3; j++)
                {
                        scanf("%d ", &a[i] [j]);
                }
        }
        printf("elements of the array are");
        for ( i=0; i<3; i++)
        {
                for (j=0;j<3; j++)
                {
                        printf("%d\t ", a[i] [j]);
                }
                printf("\n")
        }
        getch( );
}
```

a[0] [0] 10	a[0] [1] 20	a[0] [2] 30
a[1] [0] 40	a[1] [1] 50	a[1] [2] 60
a[2] [0] 70	a[2] [1] 80	a[2] [2] 90

output

```
        Enter elements of array : 1   2   3   4   5   6   7   8   9
        Elements of the array are
                1       2       3
                4       5       6
                7       8       9
```

3. Multi –dimensional arrays

- ‘C’ allows arrays of 3 (or) more dimensions
- The exact limit is determined by compiler

Syntax:

- datatype arrayname [size1] [size2] ----- [sizen];
- eg: for 3 – dimensional array:

  int a[3] [3] [3];
- No of elements = 3*3*3 = 27 elements

Program

```
main ( )
{
        int a[2][2] [2] = {1,2,3,4,5,6,7,8};
        int i,j,k;
        clrscr ( );
        printf (“elements of the array are”);
        for ( i=0; i<2; i++)
        {
                for (j=0;j<2; j++)
                {
                        for (k=0;k<2; k++)
                        {
                                printf(“%d  ”, a[i] [j] [k]);
                        }
                }
        }
        getch( );
}
```

Output : Elements of the array are :

```
                1       2       3       4       5       6       7       8
```

Arrays of pointers: (to strings)

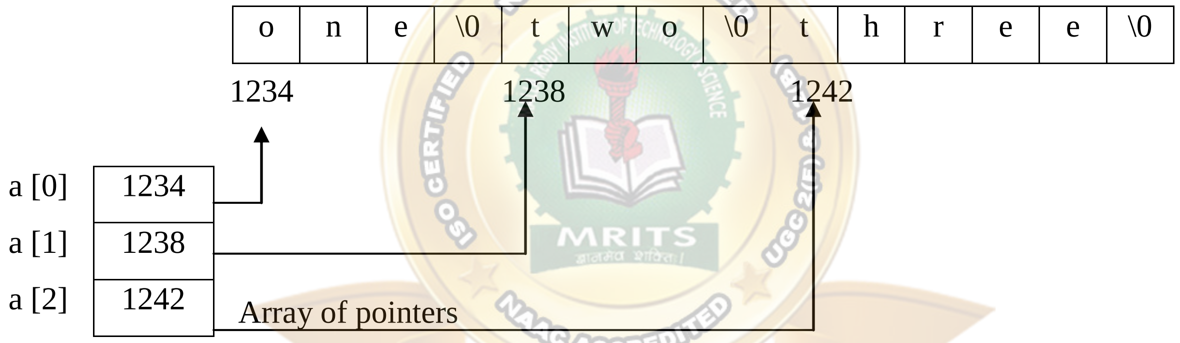
- It is an array whose elements are pointers to the base address of the string
- It is declared and initialized as follows

  char *a[ ] = {“one”, “two”, “three”};

Here, a[0] is a pointer to the base address of the string “one”

a[1] is a pointer to the base address of the string “two”

a[2] is a pointer to the base address of the string “three”



Advantage :

- Unlink the two dimensional array of characters. In (array of strings), in array of pointers to strings there is no fixed memory size for storage.
- The strings occupy only as many bytes as required hence, there is no wastage of space.

Program

```
main ( )
{
        char *a[5] = {“one”, “two”, “three”, “four”, “five”};
        int i;
        clrscr ( );
        printf ( “the strings are”)
for (i=0; i<5; i++)
```

```
                printf ("%s", a[i]);
        getch ( );
}
```

Output

```
The strings are : one   two     three   four    five
```

STRINGS

Strings basics:

String : array of characters (or) collection of characters is called a string

Declaration :

```
        char stringname [size];
        eg: char a[50]; ———→ string of length 50 characters
```

Initialization

a) using single character constant:

```
  char a[10] = { 'H', 'e', 'l', 'l', 'o' ,'\0'}
```

'H'	'e'	'l'	'l'	'o'	'\0'	

b) using string constants :

```
 char a[10] = "Hello":;
```

'H'	'e'	'l'	'l'	'o'	'\0'

'\0' is called null character.

It marks the end of the string

'\0' is automatically placed by the compiler if a string constant is given as input.

User must take care of placing '\0' at the end if single character constants are given.

Accessing:

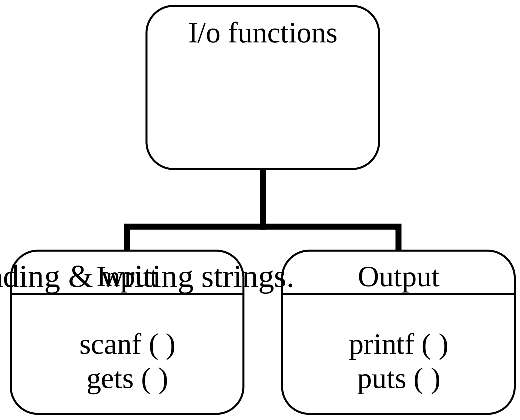
- There is a control string "%s" used for accessing the string till it encounters '\0'

Program

```
main ( )
{
        char a[10] = "Hello";
        clrscr ( );
        printf ( " given string is %s",a)
        getch ( );
}
```

Output : Given string is Hello

Input and output for strings



program : using printf ( ) and scanf ( ) for reading & writing strings.

```
main ( )
{
        char a[30];
        printf("enter your name");
        scanf ( "%s",a);
        printf ("your name is %s",a);
        getch ( );
}
```

Output

```
1. Enter your name : Ramu                2. Enter your name : Ram kumar
Your name is Ramu                           Your name is Ram
```

Note :

1. '&' is not used for accepting string because name of the string itself specifies the base address of the string
2. space is not accepted as a character by scanf( )
3. '\0' is automatically placed by the compiler at the end.

Program : Using gets ( ) and puts ( ) for reading and writing strings.

```
main ( )
{
        char a[30];
        printf ( "enter your name");
        gets (a);
        printf("Your name is");
        puts (a);
}
```

Out put

1. Enter your Name : Ramu 2) Enter your name : Ram kumar
   Your name is Ramu Your name is Ram kumar

Note : Space is also accepted as a character by gets ( )

String Library functions

❖ There are some predefined functions designed for handling strings which are available in the library "string.h"

They are :

1) strlen ( ) 6. strcmp ( )
2) strcpy ( ) 7. strncmp ( )
3) strncpy ( ) 8. strrev ( )
4) strcat ( ) 9.strstr()
5) strncat ( )

1). strlen ( )

❖ This function gives the length of the string i.e. the number of characters in a string.

Syntax:

```
        int strlen (string name)
```

program

```
#include <string.h>
main ( )
{
        char a[30] =   "Hello";
        int l;
        l = strlen (a);
        printf ("length of the string = %d", l);
        getch ( );
}
```

Output

length of the string = 5

Note : "\0" will not be counted as a character.

2). strcpy ( )

- ❖ This function is used for copying source string into destination string
- ❖ The length of the destination string must be greater than (or) equal to that of the source string

Syntax: strcpy (Destination string, Source String);

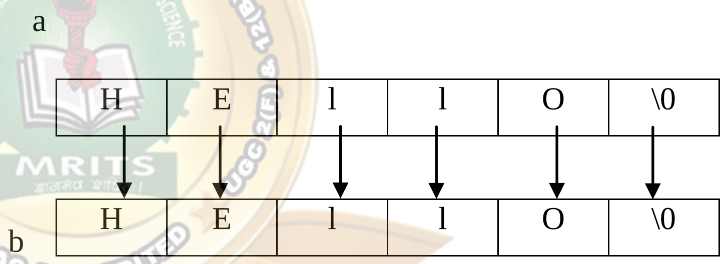
Eg:

1) char a[50];	2) char a[50];
strcpy ("Hello",a);	strcpy (a,"hello");
o/p: error	o/p: a= "Hello"

program

```
#include <string.h>
main ( )
{
        char a[50], b[50];
        clrscr ( )

        printf ("enter a source string");
        scanf("%s", a);
        strcpy ( b,a);
        printf ("copied string = %s",b);
        getch ( );
}
```



Output

```
Enter a source string : Hello
Copied string = Hello
```

3) strncpy ( )

- ❖ This function is used for copying 'n' characters of source string into destination string

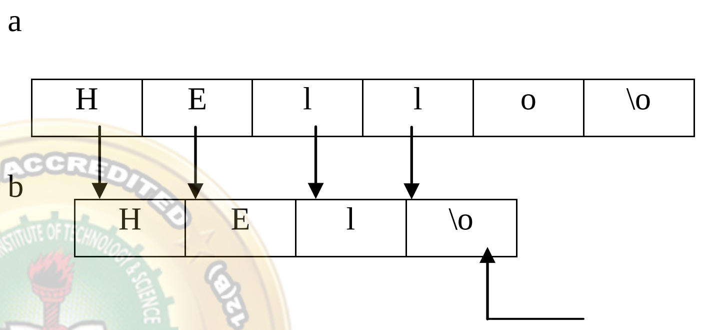
- The length of the destination string must be greater than (or) equal to that of the source string

<u>Syntax:</u>

strncpy (Destination string, Source String, n);

<u>program</u>

```
#include <string.h>
main ( )
{
        char a[50], b[50];
        clrscr ( )
        printf ("enter a string");
        gets (a);
        strncpy (b,a,3);
        b[3] = '\0';
        printf ("copied string = %s",b);
        getch ( );
}
```



Output

```
Enter a string : Hello
Copied string = Hel
```

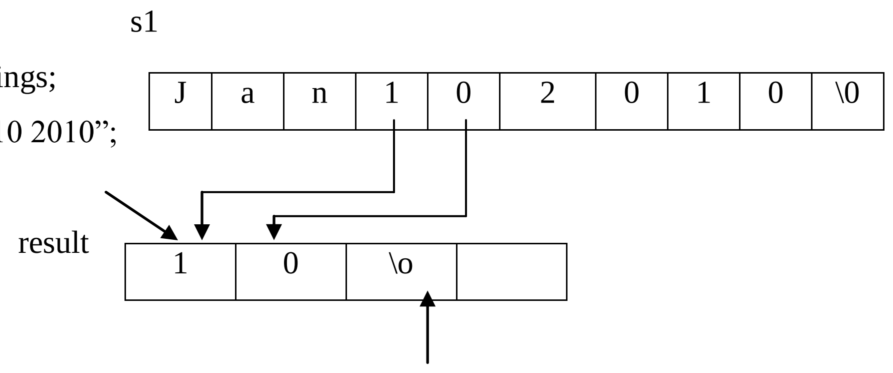
It is also used for extracting substrings;

Eg: char result[10], s1[15] = "Jan 10 2010";

strncpy (result, &s1[4], 2);

result[2] = '\0'

o/p :Result = 10



4) strcat ( ):

- This is used for combining or concatenating two strings.
- The length of the destination string must be greater than the source string
- The resultant concatenated string will be in the source string.

Syntax:

```
strcat  (Destination String, Source string);
```

program

```
#include <string.h>
main()
{
        char a[50] = “Hello”;
        char b[20] = “Good Morning”;
        clrscr ( );
        strcat (a,b);
        printf(“concatenated string = %s”, a);
        getch ( );
}
```

Output

```
        Concatenated string = Hello Good Morning
```

5) strncat ( ):

- This is used for combining or concatenating n characters of one string into another.
- The length of the destination string must be greater than the source string
- The resultant concatenated string will be in the source string.

Syntax:

```
strncat  (Destination String, Source string,n);
```

program

```
#include <string.h>
main ( )
{
        char a [30] = “Hello”;

char b [20] = “Good Morning”;
```

```
        clrscr ( );
        strncat (a,b,4);
        a [9] = '\0';
        printf("concatenated string = %s", a);
        getch ( );
}
```

Output

```
        Concatenated string = Hello Good.
```

String comparison

6) strcmp

- This function compares 2 strings
- It returns the ASCII difference of the first two non – matching characters in both the strings.

Syntax

```
        int strcmp (string1, string2);
```

If the difference is equal to zero $\Rightarrow$ string1 = string2

If the difference is positive $\Rightarrow$ string1> string2

If the difference is negative $\Rightarrow$ string1 <string2

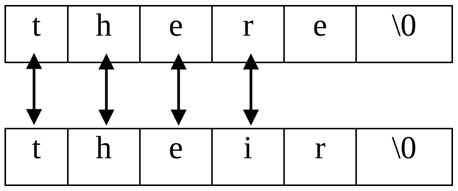
eg:

1) char a[10]= "there"

   char b[10] = "their"

   strcmp (a,b);

   Output: string1 >string2

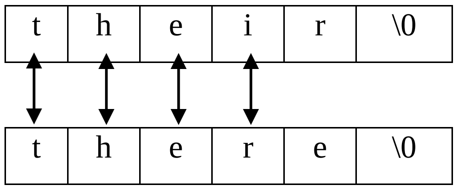


'r' > 'i'

2) char a[10]= "their"

   char b[10] = "there"

   strcmp (a,b);



Output: string1 <string2

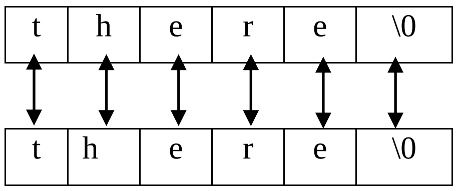
‘i’< ‘r’

3) char a[10]= “there”

char b[10] = “there”

strcmp (a,b);

Output: string1 =string2

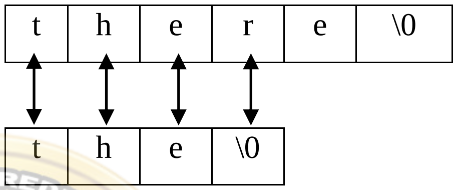


4) char a[10]= “there”

char b[10] = “the”

strcmp (a,b)

Output: string1 >string2



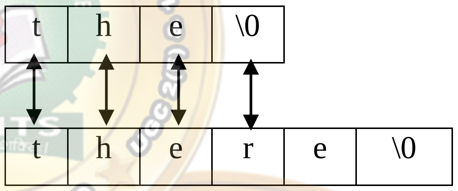
‘r’ > ‘\0’

5) char a[10]= “the”

char b[10] = “there”

strcmp (a,b);

Output: string1 <string2



‘\0’ < ‘r’

Program

```
main ( )
{
        char a[50] b [50];
        int d;
        clrscr( );
        printf (“enter 2 strings”);
        scanf (“%s %s”, a,b);
        d = strcmp (a,b);
        if (d==0)
                printf(“%s is equal to %s”, a,b);
        else if (d>0)
```

```
                printf("%s is greater than %s",a,b);
        else if (d<0)
                printf("%s is less than %s", a,b);
        getch ( );
}
```

7. strncmp ( )

- ❖ This function is used for comparing first 'n' characters of 2 strings

Syntax :

```
        strncmp ( string1, string2, n)
```

Eg:

```
        char a[10] = "the";
        char b[10] = "there"
strncmp (a,b,3);
```

Output : Both strings are equal

8. strrev( )

- ❖ The function is used for reversing a string
- ❖ The reversed string will be stored in the same string

Syntax :   strrev (string)

Program

```
main ( )
{
        char a[50] ;
        clrscr( );
        printf ("enter a string");
        gets (a);
        strrev (a);
        printf("reversed string = %s",a)
        getch ( );
}
```

Output : enter a string   Hello

Reverse string = olleH

9.strstr():

- It is used to search whether a substring is present in the main string or not.
- It returns pointer to first occurrence of s2 in s1

Syntax : strstr(mainsring,substring);

Program

```
void main()
{
        char a[30],b[30];
        char *found;

        clrscr();

        printf("Enter a string:\t");
        gets(a);

        printf("Enter the string to be searched for:\t");
        gets(b);

        found=strstr(a,b);

if(found)
                printf("%s is found in %s in %d position",a,b,found-a);
        else
                printf("-1 since the string is not found");
        getch();
}
```

Output:

```
Enter a string: how are you
Enter the string to be searched for:    you
you is found in 8 position
```

STRUCTURES AND UNIONS

Introduction :

- Structure : It is a collection of different datatype variables, grouped together under a single name. (or) It is heterogenous collection of data items that share a common name

Features of structure

1. It is possible to copy the contents of all structure elements of different datatypes to another structure variable of its type using assignment operator
2. To handle complex datatypes, it is possible to create structure within another structure, which is called nested structures.
3. It is possible to pass entire structure, individual elements of structure and address of structure to a function
4. It is possible to create structure pointers

Declaration and initialization of structures.

General form of structure declaration

```
struct tagname
{
        datatype member1;
        datatype member2;
        datatype member n;
};
```

Here,	struct	-	keyword
	tagname	-	specifies name of structure
	member1, member2 -	-	specifies the data items that make up structure.

Eg:

```
        struct book
        {
                int pages;
                char author [30];
                float price;
        };
```

Structure variables

There are 3 ways of declaring structure variables

1) struct book
```
{
        int pages;
        char author[30];
        float price;
}b;
```
2) struct
```
{
        int pages;
        char author[30];
        float price;
}b;
```

Note : Tagname can be ignored if the variable is declared of the time of defining structure

3) struct book
```
{
        int pages;
        char author[30];
        float price;
};
struct book b;
```

Initialization and accessing of structures

- ❖ The link between a member and a structure variable is established using member operator (or) dot operator
- ❖ Initialization can be done in the following ways

1. struct book
```
{
        int pages;
        char author[30];
        float price;
} b = {100, “balu”, 325.75};
```

2. struct book

```
{
        int pages;
        char author[30];
        float price;
};
struct book b = {100, “balu”, 325.75};
```

3. using member operator

```
struct book
{
        int pages;
        char author[30];
        float price;
} ;
struct book b;
        b. pages = 100;
        strcpy (b.author, “balu”);
        b.price = 325.75;
```

4. **using scanf ( )**

```
  struct book
  {
        int pages;
        char author[30];
        float price;
   } ;
struct book b;
        scanf (“%d”, &b.pages);
        scanf (“%s”, b.author);
        scanf (“%f”, &b. price);
 main ( )
```

```
{
        struct book b;
        clrscr ( );
        printf ( “enter no of pages, author, price of book”);
        scanf (“%d%s%f”, &b.pages, b.author, &b.price);
        printf(“ Details of book are”);
        printf(“pages =%d, author = %s, price = %f”, b.pages, b.author, b.price);
        getch();
}
```

Structure within structure (or) Nested structures

- Creating a structure inside another structure is called nested structure
- Consider the following example

```
struct emp
{
        int eno;
        char ename[30];
        float sal;
        float da;
        float  hra;
        float ea;
}e;
```

- This is structure defines eno, ename, sal and 3 kinds of allowances. All the items related to allowances can be grouped together and declared under a sub – structure as shown below.

```
stuct emp
{
        int eno;
        char ename[30];
        float sal;

struct allowance
```

```
                {
                        float da;
                        float  hra;
                        float ea;
                }a;
        }e;
```

- ❖ The inner most member in a nested structure can be accessed by changing all the concerned structure variables (from outer most to inner most) with the member using dot operator

Eg :

```
        e.eno;          e.ename         e.sal;
        e.a.da;         e.a.hra;        e.a.ea;
```

Program

```
struct emp
{
        int eno;
        char ename[30];
        float sal;
        struct allowance
        {
                float da;
                float  hra;
                float ea;
        }a;
}e;
main ( )
{
        clrscr ( );
        printf("enter eno, ename, salary");
        scanf ("%d%s%f", &e.eno, e.ename, &e.sal);
        printf (“enter da, hra, ea, values”);
        scanf (“%f%f%f’’, &e.a.da, &e.a.hra, &e.a.ea);
        printf(“employee details are”)
        printf (“number = %d”, e.eno);
        printf (“name = %s”, e.ename);
        printf(“salary = %f’, e.sal);
        printf(“Dearness Allowance = %f’, e.a.da);
        printf (“House Rent Allowance = %f’, e.a.hra);
        printf(“City Allowance = %f’, e.a.ea);
        getch ( )
}
```

Array of structures:

- ❖ The most common use of structure is array of structures
- ❖ To declare an array of structures, first the structure must be defined and then an array variable of that type.

Eg: struct book b[10]; $\longrightarrow$ 10 elements in an array of structures of type ‘book’

Program for accepting and printing details of 10 students

```
struct student
{
        int sno;
        char sname[30];
        float marks;
};
main ( )
{
        struct student s[10];
        int i;
        clrscr ( );
        for (i=0; i<10; i++)
        {
                printf("enter details of students%d", i+1);
                scanf ("%d%s%f", & s[i]. sno, s[i]. sname, &s[i].marks);
        }
        for (i=0; i<10; i++)
        {
                printf ("the details of student %d are", i+1);
                printf ("Number = %d", s[i]. sno);
                printf ("name = %s", s[i]. sname);
                printf ("marks =%f", s[i]. marks);
        }
        getch ( );
}
```

Pointer to structure:

- It holds the address of the entire structure .
- Mainly these are used to create complex data structures such as linked lists, trees, graphs and so on.
- The members of the structure can be accessed using a special operator called arrow operator (→) .

Declaration

```
        struct tagname *ptr;
        eg; struct student *s;
```

Accessing ;

```
        ptr→    membername;
eg: s→sno,   s→sname,  s→marks;
struct student
{
        int sno;
        char sname[30];
        float marks;
};
main ( )
```

```
{
        struct student s;
        struct student *st;
        clrscr ( );
        printf(“enter sno, sname, marks”);
        scanf (“%d%s%f”, & s.sno, s.sname, &s. marks);
        st = &s;
        printf (“details of the student are”);
        printf (“Number = %d”,  st →sno);
        printf (“name = %s”, st→sname);
        printf (“marks =%f”, st →marks);
        getch ( );
}
```

Union

Def : A union is a memory location that is shared by several variables of different data types.

Syntax:

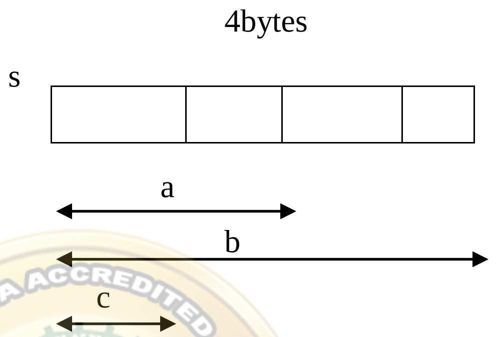
```
        union uniontag
        {
        datatype member 1;
        datatype member 2;
        ----
        ----
        datatype member n;
        };
```

Eg:

```
        union sample
        {
                int a;
                float b;
        char c;
    };
```

Declaration of union variable

```
1) union sample
    {
        int a;
        float b;
        char c;
    }s;
```



```
2) union
    {
        int a;
        float b;
        char c;
    }s;
3) union sample
    {
        int a;
        float b;
        char c;
    };
    union sample  s;
```

- ❖ when a union is declared, the compiler automatically creates a variable large enough to hold the largest variable type in the union.
- ❖ At any time only one variable can be referred.

Initialization and accessing

- ❖ To access a union member, the same syntax as that of the structure is used
- ❖ The dot operator is used for accessing members normally

❖ The arrow operator ($\longrightarrow$) is used for accessing the members using pointer

program

```
union sample
{
        int a;
        float b;
        char c;
}
main ( )
{
        union sample s = {10, 20.5, 'A'};
        clrscr( );
        printf("a=%d",s.a);
        printf("b=%f",s.b);
        printf("c=%c",s.c);
getch ( );
}
```

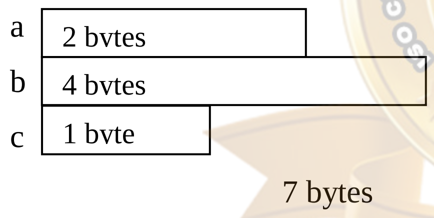
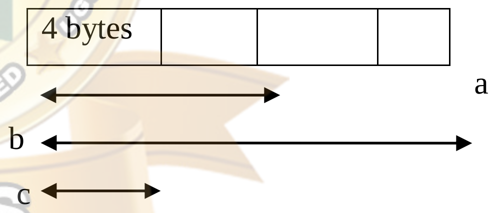
Output

```
a = garbage value
b = garbage value
c = A
```

Only the variable that is stored at last will retain its value

Differences between structures and Unions

Structure	Union
1. Definition Structure is heterogenous collection of data items grouped together under a single name **2. syntax;** struct tagname {	**1. Definition** A union is a memory location that is shared by several variables of different datatypes. **2. syntax;** union tagname {

datatype member1;	datatype member1;
datatype member2;	datatype member2;
-----	-----
-----	-----
-----	-----
};	};
3. Eg:	**3. Eg:**
struct sample	union sample
{	{
int a;	int a;
float b;	float b;
char c;	char c;
};	};
4. Keyword : struct	**4. Keyword** : union
5. Memory allocation	**5. Memory allocation**
	
6. Memory allocated is the sum of sizes of all the datatypes in structure (Here, 7bytes)	**6**. Memory allocated is the maximum size allocated among all the datatypes in union (Here, 4bytes)
7. Memory is allocated for all the members of the structure differently	**7**. Only one member will be residing in the memory at any particular instance

Union of structures

- A structure can be nested inside a union and it is called union of structures
- It is also possible to create a union inside a structure

Program

```
struct x
{
        int a;
        float b;
};
union z
{
        struct x s;
};
main ( )
{
        union z u;
        clrscr ( );
        u.s.a = 10;
        u.s.b = 30.5;
        printf("a=%d", u.s.a);
        printf("b=%f", u.s.b);
        getch ( );
}
```

Output

```
        a= 10
        b = 30.5
```

Typedef ;

❖ 'C' allows to define new datatype names using the 'typedef' keyword

❖ Using 'typedef', user will not actually create a new datatype but define a new name for an existing type.

Syntax :

typedef datatype newname;

eg :

```
typedef   int     num;
num a;                        ⇒        int a;
```

- This statement tells the compiler to recognize ‘num’ as another name for ‘int’.
- ‘num’ is used to create another variable ‘a’ .
- **‘num a’** declares ‘a’ as a variable of type ‘int’.

Program

```
main ( )
{
        typedef int hours;
        hours h;
        clrscr ( );
        printf(“enter hours”);
        scanf (“%d”, &h);
        printf(“Minutes =%d”, h*60);
        printf(“Seconds = %d”, h*60*60);
        getch ( );
}
```

Output :

```
Enter hours =1
Minutes = 60
Seconds = 360
```

Example for typedefining a structure

```
typedef employee
{
        int eno;
        char ename[30];
        float sal;
} emp;
main ( )
{
        emp e = {10, “ramu”, 5000};
        clrscr( );
        printf(“number = %d”, e.eno);
        printf(“name = %d”, e.ename);
        printf(“salary = %d”, e.sal);
        getch ( );
}
```

Bit Fields

- These are used to change the order of allocation of memory from bytes to bits
- A bit field is a set of adjacent bits whose size can be from 1 to 16 bits in length
- There are occasions where data items require much less than 16 bits of space. In such cases memory will be wasted. Bit fields can pack several data items in a word of memory

Syntax

```
        datatype name : bit – length;
```

- The datatype can be either int (or) unsigned int (or) signed int.
- Bit length specifies the number of bits
- The largest value that can be stored is $2^n - 1$, where ‘n’ is bit length

NOTE :

1) Bit fields cannot be arrayed
2) scanf ( ) cannot be used to read values into bit fields
3) cannot use pointer to access the bit fields
4) Bit fields should be assigned values within the range of their size

Bit Length	**Range of values**
1	0 to 1
2	0 to 3 (2^2-1)
3	0 to 7 (2^3-1)
n	0 to 2^n-1

Eg:

```
1) struct pack
  {
        int count;
        unsigned a : 2;
        unsigned b : 3;
  };
```

Here, count will be in 2 bytes. ‘a’ and ‘b’ will be packed into next 1 byte

2) struct pack

```
{
        unsigned a : 2;
        int count;
        unsigned b : 3;
};
```

Here, ‘a’ will be in 1 byte, ‘count’ in 2 bytes and ‘b’ in 1 bytes.

Note ;

1. Bit Fields are packed into words as they appear in the definition
2. All unsigned bit fields must be placed in order for effectively using the memory

Program

```
struct vehicle
{
        unsigned type : 3;
        unsigned fuel : 2;
        unsigned model : 3;
};
main ( )
{
        struct vehicle v;
        v.type = 4;
        v. fuel = 2;
        v. model = 5;
        printf (“type of vehicle =%d”, v.type);
        printf (“fuel =%d”, v.fuel);
```

Note: Instead of 6 bytes only 1 byte of memory will be allocated

```
        printf ("model =%d", v.model);
}
```

Enumerated Data type

- These are used by the programmers to create their own data types and define what values the variables of these datatypes can hold.

Keyword : enum

Syntax :

```
        enum tagname
        {
                identifier1, identifier2,…….,identifier n
        };
```

eg :

```
        enum week
        {
        mon,tue, wed, thu, fri, sat, sun
        };
```

- Here, with identifier values are constant unsigned integers and start from 0.
- Mon refers 0, tue refers 1 and so on.

Program :

```
main ( )
{
        enum week {mon, tue, wed, thu, fri, sat, sun};
        clrscr ( );
        printf ("Monday = %d", mon);
        printf ("Thursday = %d", thu);
        printf ("Sunday = %d", sun);
}
```

```
Output :  Monday = 0
          Thursday =3
          Sunday =6
```

- enum identifiers can also be assigned initial value.

Program

```
main ( )
{
        enum week {mon=1, tue, wed, thu, fri, sat, sun};
        clrscr ( );
        printf ("Monday = %d", mon);
        printf ("Thursday = %d", thu);
        printf ("Sunday = %d", sun);
}
```

```
Output : Monday = 1
         Thursday =4
         Sunday =7
```

POINTERS

Pointer : Pointer is a variable that stores the address of another variable.

Features of Pointers

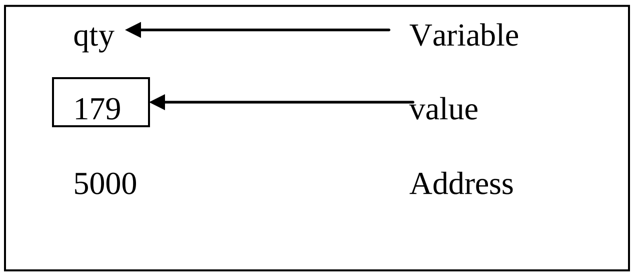
- Pointer saves the memory space
- Execution time with pointer is faster because data is manipulated with the address i.e. direct access to memory location
- The memory is accessed efficiently with the pointer i.e. dynamically memory is allocated and deallocated
- Pointers are used with data structures

Pointer declaration, initialization and accessing.

Consider the following statement :

```
int qty = 179;
```

The representation of the variable in memory is as follows



Declaring a pointer

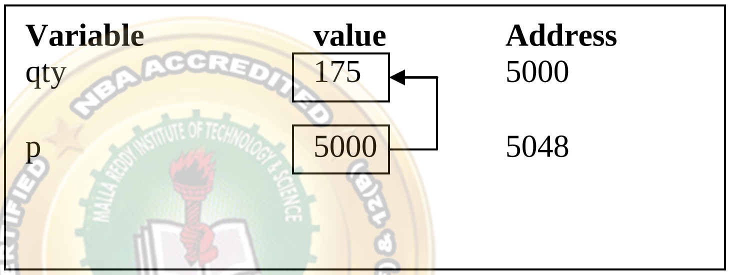
```
int *p;
```

It means ‘p’ is a pointer variable that holds the address of another integer variable.

Initialization of a pointer

❖ Address operator (&) is used to initialize a pointer variable.

Eg:
```
int qty = 175;
int *p;
p= &qty;
```



Accessing a variable through its pointer

❖ To access the value of the variable, indirection operator (*) is used.

eg :
```
int qty = 175, *p,n;

p = &qty;

n = *p;
```

‘*’ can be treated as value at address

❖ The 2 statements are equivalent to the following statement

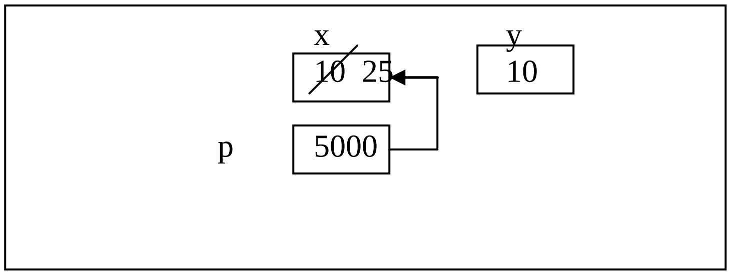
p = &qty;

n = *p; $\Longleftrightarrow$ n =qty

Program

main ( )

```
{
        int x,y;
        int *p;
        clrscr ( );
        x= 10;
        p = &x;
        y= *p;
        printf (“Value of x = %d”, x);
        printf (“x is stored at address %u”, &x);
        printf (“Value of x using pointer =  %d”, *p);
        printf (“address of x using pointer = %u”, p);
        printf (“value of x in y = %d”, y);
        *p = 25;
        printf (“now x = %d”, x)
        getch ( );
}
```



Output

```
Value of x = 10
x is stored at address = 5000
Address of x using pointer = 10
Address of x using pointer = 5000
Value of x in y = 10
Now x = 25
```

<u>**Pointers and arrays**</u>

- ❖ Continuous memory locations are allocated for all the elements of the array by the compiler
- ❖ The base address is the location of the first element (index 0) of the array.

Eg : int a [5] = {10, 20,30,40,50};

The five elements are stored as follows

Elements	**a[0]**	**a[1]**	**a[2]**	**a[3]**	**a[4]**
Value	10	20	30	40	50

Address	1000	1002	1004	1006	1008

base address

a= &a[0]=1000

if ‘p’ is declared as integer pointer, then the array ‘a’ can be pointed by the following assignment

```
        p = a;

    (or) p = &a[0];
```

- Every value of ‘a’ can be accessed by using p++ to move from one element to another. When a pointer is incremented, its value is increased by the size of the datatype that it points to. This length is called the “scale factor”
- The relationship between ‘p’ and ‘a’ is shown below

P	= &a[0]	=	1000
P+1	= &a[1]	=	1002
P+2	= &a[2]	=	1004
P+3	= &a[3]	=	1006
P+4	= &a[4]	=	1008

- Address of an element is calculated using its index and the scale factor of the datatype.

For eg:

```
  Address of a[3] = base address + (3* scale factor of int)
                  =  1000 + (3*2)
                  =  1000 +6
                  =  1006
```

- instead of using array indexing, pointers can be used to access array elements.
- *(p+3) gives the value of a[3]

a[i] = *(p+i)

Program

```
main ( )
{
        int a[5];
        int *p,i;
        clrscr ( );
        printf (”Enter 5 lements”);
        for (i=0; i<5; i++)
                scanf (“%d”, &a[i]);
        p = &a[0];
        printf (“Elements of the array are”);
        for (i=0; i<5; i++)
                printf(“%d”, *(p+i));
        getch( );
}
```

Output

```
Enter 5 elements : 10 20 30 40 50
Elements of the array are : 10        20      30      40      50
```

Array of pointers

- ❖ It is collection of addresses (or) collection of pointers

Declaration

    datatype *pointername [size];

eg: int *p[5]; $\longrightarrow$ It represents an array of pointers that can hold 5 integer element addresses

p[0]	p[1]	p[2]	p[3]	p[4]

Initialization

‘&’ is used for initialization

Eg :

```
int a[3] = {10,20,30};
int *p[3], i;
for (i=0; i<3; i++)          (or)      for (i=0; i<3,i++)
     p[i] = &a[i];
```

```
p[i] =    a+i;
```

Accessing

Indirection operator (*) is used for accessing

Eg:
```
for (i=0, i<3; i++)
printf ("%d", *p[i]);
```

Program

```
main ( )
{
        int a[3] = {10,20,30};
        int *p[3],i;
        for (i=0; i<3; i++)
        p[i] = &a[i];
        printf (elements of the array are")
        for (i=0; i<3; i++)
                printf ("%d \t", *p[i]);
        getch();
}
```

Output elements at the array are : 10   20   30

PREPROCESSOR COMMANDS & FILES

Preprocessor commands

- ‘preprocessor’ is a program that processes the source code before it passes through the compiler
- It operates under the control of preprocessor directives which begin with the symbol #

3 types

1) Macro substitution directives
2) File inclusion directives
3) compiler control directives

1) Macro substitution directives

- It replaces every occurence of the identifier by a predefined string.

Syntax for defining a macro

```
        # define identifier string
```

```
Eg: #define    PI   3.1415
        #define   f(x)     x *x
        #undef   PI
```

Program

```
#define wait getch( )
main ( )
{
clrscr ( );
printf ("Hello");
wait ;
}
```

Output:

```
Hello
```

Program

```
#define wait getch( )
main ( )
{
#undef wait;
clrscr ( );
printf ("Hello");
wait ;
}
```

Output

Error since wait is undefined before using it

2. File inclusion directives:

- An external file containing functions (or) macro definitions can be included using #include directive

Syntax

```
# include <filename> (or) #include "filename"
```

Eg:

```
#include <stdio.h>
main ( )
{
        printf ("hello");
}
```

Output

```
Hello
```

The definition of the function printf ( ) is present in <stdio.h>header file.

3. Compiler control directives

- C pre processor offers a feature known as conditional compilation, which can be used to switch ON (or) OFF a particular line (or) group of lines in a program.

Eg: #if, #else, #endif etc.

```
#define LINE 1
main ( )
{
#ifdef   LINE
        printf ("this is line number one");
#else
        printf('This is line number two");
#endif
}
```

Output

```
This is line number one
```

FILE :

Definition : It is collection of records (or) It is a place on hard disk where data is stored permanently.

Types of Files: (1)Text file

(2)Binary File

1. Text File : It contains alphabets and numbers which are easily understood by human beings.

2. Binary file : It contains 1's and 0's which are easily understood by computers.

- Based on the data that is accessed, files are classified in to

(1) Sequential files

(2) Random access files

(1) Sequential files: Data is stored and retained in a sequential manner.

(2) Random access Files : Data is stored and retrieved in a random way.

Operations on files : 1. Naming the file

2. Opening the file

3. Reading from the file

4. Writing into the file

5. Closing the file

Syntax for opening and naming file.

1) FILE *File pointer;

Eg : FILE * fp;

2) File pointer = fopen (“File name”, “mode”);

Eg : fp = fopen (“sample.txt”, “w”);

```
FILE *fp;
fp = fopen (“sample.txt”, “w”);
```

<u>Modes of the opening the file :</u>

r	-	File is opened for reading
w	-	File is opened for writing
a	-	File is opened for appending (adding)
r+	-	File is opened for both reading & writing
w+	-	File is opened for both writing & reading
a+	-	File is opened for appending & reading
rt	-	text file is opened for reading
wt	-	text file is opened for writing
at	-	text file is opened for appending
r+t	-	text file is opened for reading & writing
w+t	-	text file is opened for both writing & reading

a+t	-	text file is opened for both appending & reading
rb	-	binary file is opened for reading
wb	-	binary file is opened for writing
ab	-	binary file is opened for appending
r+b	-	binary file is opened for both reading & writing
w+b	-	binary file is opened for both writing & reading
a+b	-	binary file is opened for both appending & reading.

1) Write mode of opening the file

```
FILE *fp;
fp =fopen (“sample.txt”, “w”);
```

a) If the file does not exist then a new file will be created

b) If the file exists then old content gets erased & current content will be stored.

2. Read mode of opening the file:

```
FILE *fp
fp =fopen (“sample.txt”, “r”);
```

a) If the file does not exists, then fopen function returns NULL value.

b) If the file exists then data is read from the file successfullly

3. Append mode of opening a file

```
FILE *fp;
fp =fopen (“sample.txt”, “a”);
```

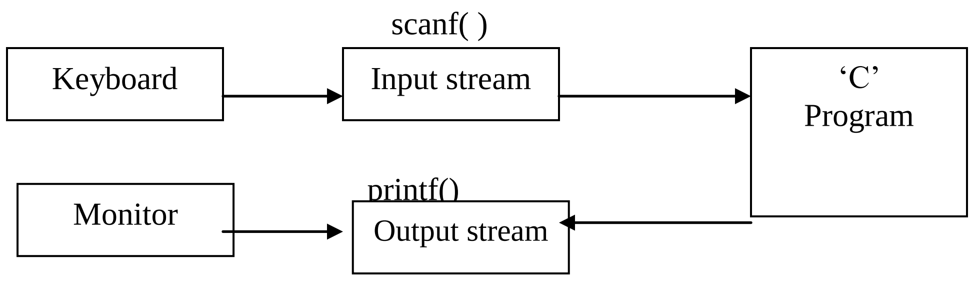
a) If the file doesn’t exists, then a new file will be created.

b) If the file exists, the current content will be appended to the old content

Mode	Exist	Not exist
r	Read	fp = “NULL”
w	Current content	New file will be created
a	Old content Current content	New file will be created

I/O STREAMS:

Stream : flow of data



I/0 functions:

1) high level I/o

- These are easily understood by human beings
- Advantage: portability.

2) Low level I/o

- These are easily understood by computer
- Advantages. Execution time is less
- Disadvantage: Non protability

High level I/o Functions

1) fprintf ( )	-	to write data into a file
2) fscanf ( )	-	To read data from a file
3) putc ( )/ fputc()	-	to write a character into a file
4) getc ( ) /fgetc()	-	to read a character from a file
5) putw ( )	-	To write a number into a file
6) getw ( )	-	To read number from a file
7) fputs ( )	-	To write a string into a file
8) fgets ( )	-	To read a string from a file
9)fread()	-	To read an entire record from a file

10)fwrite() - To write an entire record into a file

<u>fprint ( ) & fscanf ( ) functions</u>

1) fprint ( )

<u>Syntax</u> : fprintf (file pointer, " control string", variable list)

<u>Eg:</u>
```
FILE *fp;
fprintf (fp, "%d%c", a,b);
```

2) fscanf ( )

<u>Syntax :</u> fscanf(file pointer, "control string", & variable list);

<u>Eg:</u>
```
FILE *fp;
fscanf (fp, "%d%c", &a,&b);
```

Program for storing the details of an employee in a file and print the same

```
main ( )
{
        FILE *fp;
        int eno;
        char ename [30];
        float sal;
        clrscr ( );
        fp =fopen ("emp.txt", "w");
        printf ("enter the details of eno, ename, sal");
        scanf ("%d%s%f", &eno, ename, &sal);
        fprintf (fp, "%d%s%f", eno, ename, sal);
        fclose (fp);
        fp = fopen ("emp.txt", "r");
        fscanf (fp, "%d%s%f", &eno, ename, &sal);
        printf (“employee no: = %d”, eno);
        printf (“employee name = %s”, ename);
        printf (“salary = %f”, sal);
        fclose (fp);
        getch( );
}
```

Program for storing the details of 60 employers in a file and print the same

```
main ( )
{
        FILE *fp;
         int eno, i;
        char ename [80];
        float sal;
        clrscr ( );
        fp = fopen (“emp1. txt”, “w”);
        for (i=1; i<60; i++)
        {
        printf (“enter the eno, ename, sal of emp%d”, i);
        scanf (“%d%s%f”, &eno, ename, &sal);
        fprintf (fp, “%d %s %f”, eno, ename, sal);
        }
        fclose (fp);
        fp = fopen (“emp1.txt”, “r”);
        for (i=1; i<60; i++)
        {
        fscanf(fp, “%d %s %f”, &eno, ename, &sal);
        printf (“details of employee %d are \n”, i);
        printf (“eno = %d, ename = %s, sal = %f”, eno, ename, sal);
        }
        fclose (fp);
        getch ( );
```

}

putc( ) and getc( ) functions:

1) putc ( ): It is used for writing a character into a file

Syntax :

```
putc (char ch, FILE *fp);
```

Eg :

```
FILE *fp;
char ch;
putc(ch, fp);
```

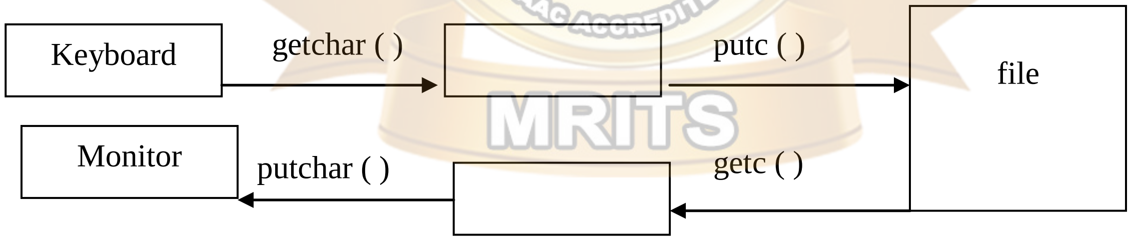
2) get c ( ) : It is used to read a character from file

Syntax :

```
char getc (FILE *fp);
```

Eg:

```
FILE *fp;
char ch;
ch = getc(fp);
```



Program :

```
main ( )
{
        FILE *fp;
        char ch;
        clrscr ( );
        fp = fopen ("characters.txt", "w");
        printf (“enter text. press ctrl+z at the end”);
        while ((ch = getchar ( ))! = EOF)
        {
                putc(ch, fp);
        }
        fclose (fp);
        fp =open (“characters. txt”, “r”);
        printf (“file content is \n”);
        while ((ch = getc (fp))! = EOF)
        {
                putchar (ch);
        }
        fclose (fp);
        getch ();
}
```

Output:

```
Enter text press ctrl+z at the end.
Hello how r u ^z
File Content is
Hello How r u
```

putw ( ) and getw ( ) functions:

1. putw( ) : It is used for writing a number into file.

Syntax: putw (int num, FILE *fp);

Eg:

```
FILE *fp;
int num;
putw(num, fp);
```

2. getw ( ): It is used for reading a number from a file

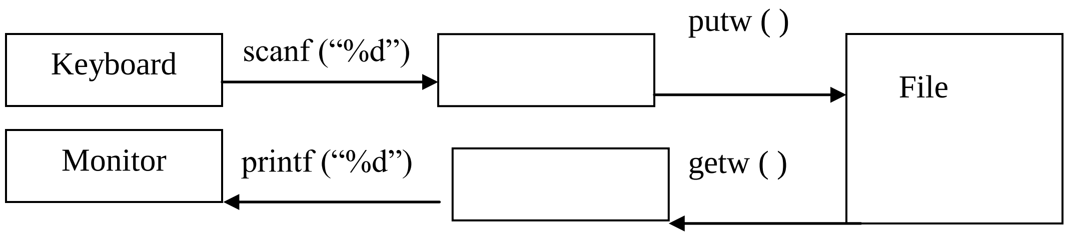
Syntax :

```
int getw (FILE *fp);
```

Eg :

```
FILE *fp;
int num;
```

```
        num = getw(fp);
```



Program for storing no's from 1 to 10 and print the same

```
main ( )
{
        FILE *fp;
        int i;
        clrscr ( );
        fp = fopen ("number. txt", "w");
        for (i =1; i< = 10; i++)
        {
                putw (i, fp);
        }
        fclose (fp);
        fp =fopen ("number. txt", "r");
        printf ("file content is ");
        for (i =1; i< = 10; i++)
        {
                i= getw(fp);
                printf ("%d",i);
         }
        fclose (fp);
        getch ( );
        }
```

Program for copying the contents of one file into another file

```
main ( )
{
        FILE *fp1, *fp2;
        char ch;
        clrscr ( );
        fp1 = fopen (“file1.txt”, “w”);
        printf (“enter text press ctrl+z at the end”);
        while ((ch = getchar ( ))! = EOF)
        {
                putc(ch, fp1);
        }
        fclose (fp1);
        fp1 =fopen (“file1. txt”, “r”);
        fp2 =fopen (“file2. txt”, “w”);
        while ((ch = getc (fp1))! = EOF)
        {
                putc(ch,fp2);
        }
        fclose (fp1);
        fclose (fp2);
        fp2 = fopen (“file2.txt”, “r”);
        printf (“File2 contents are”);
        while ((ch = getc(fp2))! = EOF)
                putchar (ch);
        fclose (fp2);
        getch ();
}
```

Program for displaying the contents of a file

```
main ( )
{
        FILE *fp;
        char ch ;
        clrscr ( );
        fp = fopen (“file1.txt”,”r”);
        if (fp = = NULL)
        {
                printf (“File does not exist”);
        }
        else
        {
                printf (“file content is”)
                while ((ch = getc(fp))! = EOF)
                putchar (ch);
        }
        fclose (fp);
        getch ( );
}
```

Program to merge two files into a third file. (the contents of file1, file2 are placed in file3)

```
main ( )
{
        FILE *fp1, *fp2, *fp3;
        char ch;
        clrscr ( );
        fp1 = fopen (“file1.txt”, “w”);
        printf (“enter text into file1”);
        while ((ch = getchar ( ))! = EOF)
        {
                putc(ch, fp1);
        }
        fclose (fp1);
        fp2 = fopen (“file2.txt”, “r”);
        printf (“enter text into file2”);
        while ((ch = getchar ( ))! = EOF)
                putc(ch, fp2);
        fclose (fp2);
```

```
            fp1 =fopen (“file1. txt”, “r”);
            fp2 =fopen (“file2. txt”, “r”);
            fp3 =fopen (“file3. txt”, “w”);
            while ((ch = getc (fp1))! = EOF)
                        putc(ch,fp3);
             while ((ch = getc (fp2))! = EOF)
                        putc(ch,fp3);
            fclose(fp1);
            fclose (fp2);
            fclose (fp3);
            fp3 = fopen (“file3.tx”, “r”);
            printf (“File3 contents is”);
            while ((ch = getc(fp3))! = EOF)
                        purchar (ch);
            fclose (fp3);
            getch ();
}
```

fput c ( ) and fgetc ( ) functions :

1) fputc( ) : It is used for writing a character in to a file .

Syntax :

fputc (char ch, FILE *fp);

```
Eg :    FILE *fp;
        char ch;
        fputc (ch.fp);
```

2. fgetc( ) : This is used for reading a character from a file

Syntax :

fputc (char ch, FILE *fp);

```
Eg :    FILE *fp;
        char ch;
```

ch = fgetc(fp);

<u>fgets ( ) and fputs ( ) functions :</u>

1) fgets ( ) : It is used for reading a string from a file

Syntax :

fgets (string variable, No. of characters, File pointer);

Eg : FILE *fp;

char str [30];

fgets (str,30,fp);

2) fputs ( ) : It is used for writing a string into a file

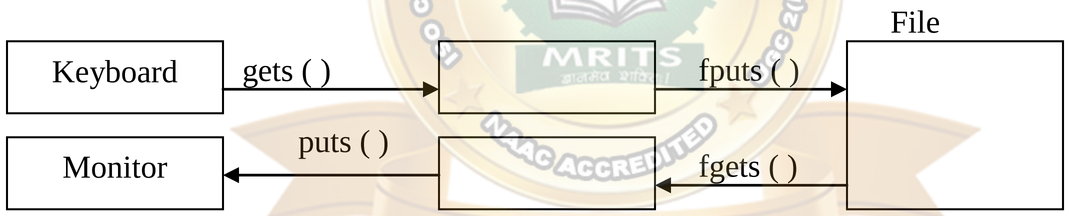
Syntax :

fputs (string variable, file pointer);

Eg : FILE *fp;

char str[30];

fputs (str,fp);



Program :

main ( )

{

FILE *fp;

char str [30];

int i,n;

clrscr ( );

printf (“enter no of strings”);

scanf (“%d”, & n);

fp = fopen (‘strings.txt”, “w”);

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

{

```
            printf (”enter string %d”,i);
            gets (str);
            fputs (str, fp);
        }
        fclose (fp);
        fp = fopen (”strings.txt”, ”r”);
        for (i=1; i<=n; i++)
        {
            fgets (str, 30, fp);
            printf (”string %d =”, i);
            puts (str);
        }
        fclose (fp);
        getch ( );
    }
```

<u>fread ( ) and fwrite ( ) functions</u>

1. fread ( ) : It is used for reading entire record at a time.

Syntax : fread(& structure variable, size of (structure variable), no of records, file pointer);

Eg : struct emp

```
        {
            int eno;
            char ename [30];
            float sal;
        } e;
        FILE *fp;
        fread (&e, sizeof (e), 1, fp);
```

2. fwrite ( ) : It is used for writing an entire record at a time.

Syntax : fwrite(& structure variable , size of structure variable, no of records, file pointer);

Eg : struct emp

```
        {
            int eno:
                char ename [30];
                float sal;
        } e;
        FILE *fp;
        fwrite (&e, sizeof(e), 1, fp);
```

program for storing the details of 60 students into a file and print the same using fread ( ) and fwrite ( )

```
struct student
{
        int sno;
        char sname [30];
        float marks;
};
main ( )
{
        struct student s[60];
        int i;
        FILE *fp;
        clrscr ( );
        fp = fopen (“student1. txt”, “w”);
        for (i=0; i<60; i++)
        {
                printf (“enter details of student %d”, i+1);
                scanf (“%d%s%f”. &s[i].sno,s[i].sname, &s[i].marks);
                fwrite (&s[i], sizeof (s[i]), 1, fp);
        }
        fclose (fp);
        fp = fopen (“student1. txt”, “r”);
        for (i=0; i<60; i++)
        {
                printf (“details of student %d are”, i+1);
                fread (&s[i], sizeof (s[i]) ,1,fp);
                printf("student number = %d", s[i]. sno.);
                printf("student name  = %s", s[i]. sname.);
                printf("marks = %f", s[i]. marks);
        }
        fclose (fp)
        getch( );
}
```

ERROR HANDLING IN FILES:-

- Some of the errors in files are
    1. Trying to read beyond end of file
    2. Device over flow
    3. Trying to open an invalid file
    4. Performing a invalid operation by opening a file in a different mode.

Functions for error handling.

1) ferror ( )
2) perror ( )
3) feof ( )

1. ferror ( )

It is used for detecting an error while performing read / write operations.

Syntax :

```
        int ferror (file pointer);
```

```
eg :    FILE *fp;
        if (ferror (fp))
        printf ("error has occurred");
```

→ it returns zero if success and a non- zero otherwise.

2. perror ( )

→ It is used for printing an error.

Syntax :

```
        perror (string variable);
```

```
Eg :    FILE *fp;
        char str[30] = ”Error is”;
        perror (str);
O/P : Error is : error 0
```

Program :

```
main ( )
{
        FILE *fp;
        char str[30] = “error is”;
        int i = 20;
        clrscr ( );
        fp = fopen (“sample. txt”, “r”);
        if (fp = = NULL)
        {
                printf (“file doesnot exist”);
        }
        else
        {
        fprintf (fp, “%d”, i);
        if (ferror (fp))
        {
                perror (str);
                printf (“error since file is opened for reading only”);
        }
fclose (fp);
getch ( );
}
```

O/P: Error is : Error1 □ □ compiler generated.

Error since file is opened for reading □ □ by us.

3. feof ( )

It is used for checking whether end of the file has been reached (or) not.

Syntax :

```
int feof ( file pointer);
```

```
Eg :    FILE *fp;
        if (feof (fp))
        printf ("reached end of the file");
```

→ If returns a non zero if success and zero otherwise.

Program:

```
main ( )
{
        FILE *fp;
        int i,n;
        clrscr ( );
        fp = fopen ("number. txt", "w");
        for (i=0; i<=100;i= i+10)
        {
                putw (i, fp);
        }
        fclose (fp);
        fp = fopen ("number. txt", "r");
        printf ("file content is");
        for (i=0; i<=100; i++)
        {
                n = getw (fp);
                if (feof (fp))
                {
                printf ("reached end of file");
                break;
                }
                else
                {
```

```
                printf ("%d", n);
                }
        }
        fclose (fp);
getch ( );
}
```

Outpute : File content is

```
        10      20      30      40      50
        60      70      80      90      100
        Reached end of the file.
```

<u>Other file functions</u>

Random accessing of files

1. ftell ( )
2. rewind ( )
3. fseek ( )

1. ftell ( ) : It returns the current postion of the file pointer

Syntax : int n = ftell (file pointer)

```
Eg :    FILE *fp;
        int n;
        _____
        _____
        _____
        n = ftell (fp);
```

Note : ftell ( ) is used for counting the no of characters entered into a file.

2. rewind ( )

It makes the file pointer move to the beginning of the file.

Syntax: rewind (file pointer);

```
Eg : FILE *fp;
        -----
```

```
-----
rewind (fp);
n = ftell (fp);
printf ("%d", n);
```

o/p: 0 (always).

3. fseek ( )

It is used to make the file pointer point to a particular location in a file.

Syntax: fseek(file pointer,offset,position);

offset :

- The no of positions to be moved while reading or writing.
- If can be either negative (or) positive.

→ Positive - forward direction.

→ Negative – backward direction .

position :

- it can have 3 values.

0 – Beginning of the file

1 – Current position

2 – End of the file

Eg :

1. fseek (fp,0,2) - fp is moved 0 bytes forward from the end of the file.
2. fseek (fp, 0, 0) – fp is moved 0 bytes forward from beginning of the file
3. fseek (fp, m, 0) – fp is moved m bytes forward from the beginning of the file.
4. fseek (fp, -m, 2) – fp is moved m bytes backward from the end of the file.

Errors :

1. fseek (fp, -m, 0);
2. fseek(fp, +m, 2);

Write a program for printing some content in to the file and print the following ?

1. Number of characters entered into the file.
2. Reverse the characters entered into the file.

```
main ( )
    {
    FILE *fp;
    char ch;
    int n;
    clrscr ( );
    fp = fopen (“reverse. txt”, “w”);
    printf (“enter text press ctrl+z of the end”);
    while ((ch = getchar( ) ) ! EOF)
            {
                    putc (ch, fp);
            }
    n = ftell (fp)
    printf ( “No. of characters entered = %d”, n);
    rewind (fp);
    n = ftell (fp);
    printf (“fp value after rewind = %d”,n);
    fclose (fp);
    fp = fopen (“reverse.txt”, “r”);
    fseek (fp, -1, 2);
    printf (“reversed content is”);
    do
    {
            ch = getc (fp);
            printf (“%c”, ch);
    } while (!fseek (fp, -2, 1);
    fclose (fp);
    getch ( );
    }
```

Output : Enter text press ctrl z at the end.

```
        How are you ^z
```

No. of characters entered = 11

fp value after rewind =0

Reversed content is **uoy era woh**.

UNIT-IV

FUNCTIONS

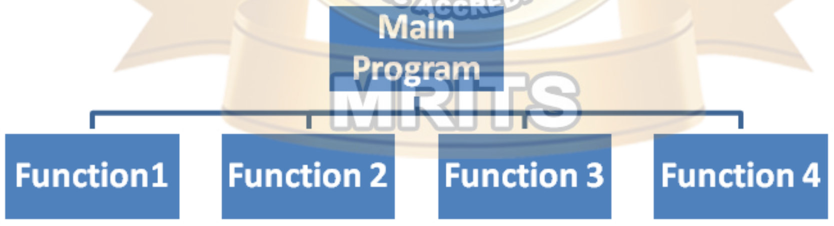
Def : A function is a self contained block that carries out a specific well defined task.

Advantages

1. Reusability i.e. a function may be used by many other programs.
2. The length of the source program can be reduced.
3. It is easy to locate and isolate a faulty function.
4. It facilitates top-down modular programming.

Top down design and structure charts

- **"Top down design"** is a problem solving method in which a complex problem is solved by breaking up into sub problems.
- It proceeds from the original problem at the top level to the sub problems at each lower level
- "structure chart" is a documentation tool that shows the relationships among the sub problems of a problem.



- The splitting of a problem into its related sub problems is analogous to the process of refining an algorithm.

e.g. Performing arithmetic operations on 2 numbers

1. find sum
2. find difference
3. find product
4. find quotient

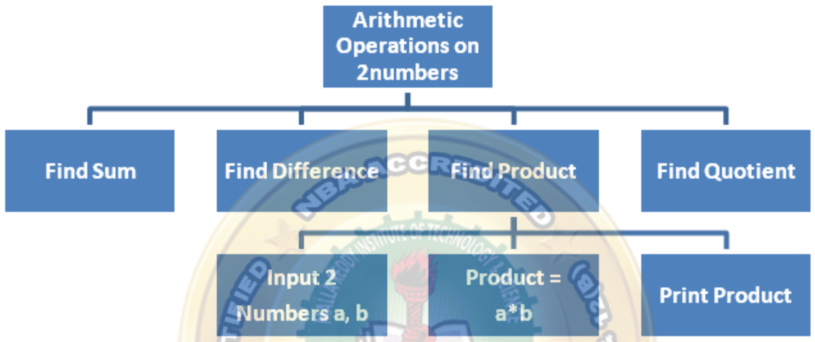
Refined algorithm for 1st step

1.1 take 2 numbers a, b

1.2 sum, c = a + b

1.3 print sum

Structure chart



Types of functions

Functions are broadly classified into 2 types

They are

1) predefined functions
2) user defined functions

1) predefined (or) library functions

❖ These functions are already defined in the system libraries

❖ Programmer can reuse the existing code in the system libraries to write error free code.

❖ But to use the library functions, user must be aware of syntax of the function.

eg: 1) sqrt() function is available in **math.h** library and its usage is :

y= sqrt (x)

$\xrightarrow{}$ number must be positive

eg: y = sqrt (25)

then ‘y’ = 5

2 printf ( ) function is available in **stdio.h** library

3) clrscr ( ) function is available in **conio.h** library

Program

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main ( )
{
        int x,y;
        clrscr ( );
        printf (“enter a positive number”);
        scanf (“ %d”, &x)
        y = sqrt(x);
        printf(“squareroot = %d”, y);
        getch();
}
```

Output

```
        Enter a positive number  25
        Squareroot = 5
```

2) user defined functions

These functions must be defined by the programmer (or) user

Programmer has to write the coding for such functions and test them properly before using them

The syntax of the function is also given by the user and therefore need not include any header files.

Eg: main ( ), swap ( ), sum ( ) etc

Program

```
#include<stdio.h>
#include<conio.h>
main ( )
{
        int sum (int, int);
        int a, b, c;
        clrscr ( );
        printf ("enter 2 numbers");
        scanf (" %d %d", &a ,&b)
        c = sum (a,b);
        printf("sum = %d", c);
        getch();
}
int sum (int a, int b)
{
        int c;
        c=a+b;
        return c;
}
```

Output

```
Enter 2 numbers  10    20
Sum = 30
```

Communications Among functions

Functions communicate among themselves using arguments and return value.

Farm of 'C' function

```
return-datatype function name (argument list)
{
        local variable declarations;
        executable statements(s);
        return (expression);
}
eg: void mul (int x, int y)
        {
                int p;
                p=x*y;
                printf("product = %d",p);
}
```

Return values and their types

- ❖ A function may (or) may not send back any value to the calling function
- ❖ If it does, it is done through the return statement
- ❖ A called function can only return one value per call
- ❖ The return types are void, int, float, char and double.
- ❖ If a function is not returning any value then its return type is 'void'

Function name

- ❖ A function must follow the same rules of formation as other variables name in 'C'
- ❖ A function name must not duplicate library routine names (or) predefined function names.

Argument list

- ❖ The argument list contains valid variable names separated by commas
- ❖ The argument variables receive values from the calling function, thus providing a means for data communication from the calling function to the called function.

Calling a function

- ❖ A function can be called by simply using the function name in a statement

Function definition

- ❖ When the compiler encounters a function call, the control is transferred to the function definition.
- ❖ All the statements ,in the called function, are together called as function definition

Function header

- ❖ The first line in the function definition is called function header.

Actual parameter

- ❖ All the variables inside the function call are called actual parameters.

Formal parameters

- ❖ All the variables inside the function header are called formal parameters

Program

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

```
main ( )
{
        int mul (int, int);    ——→   function prototype
        int a,b,c;
        clrscr( );
        printf (“enter 2 numbers”);
        scanf(“%d %d”, &a, &b);
        c = mul (a,b);   ————→function call
        printf(“product =%d”,c);     →Actual parameters
        getch ( );
}
        int mul (int a, int b)       →
                                     Formal parameters
        {                            →function header
                int c;        ⎫
                c = a *b;     ⎬——→Function definition
                return c;     ⎭
        }
```

<u>Output</u>

```
Enter 2 numbers:    10      20
Product = 200
```

<u>Categories of functions:</u>

❖ Depending on whether arguments are present (or) not and whether a value is returned (or) not, functions are categorized into:

1) functions without arguments and without return values
2) functions without arguments and with return values
3) Functions with arguments and without return values
4) Functions with arguments and with return values.

<u>1) functions without arguments and without return values</u>

Calling function	Analysis	Called function

main ( )
{
    --- 	
    --- 	
    fun ( );
}
	→ No arguments are passed
	
	No values are sent back ←
		fun ( )
		{
		    ---- 	
		    ---- 	
		}

eg:

```
main ( )
{
        void sum ( );
        clrscr ( );
        sum ( );
        getch ( );
}
void sum ( )
{
        int a,b,c;
        printf("enter 2 numbers");
        scanf ("%d%d", &a, &b);
        c = a+b;
        printf("sum = %d",c);
}
```

<u>Output</u>

```
Enter 2 numbers
     10    20
   Sum=30
```

<u>2) Functions without arguments and with return values</u>

Calling function	Analysis	Called function

main ( )
{
    int c;
    --- 	
    c= fun ( );
    ----- 	
    ----- 	
}
	→ No arguments are passed
	
	values are sent back ←
		fun ( )
		{
		    ---- 	
		    ---- 	
		  return c;
		
		}

eg:

```
main ( )
{
        int sum ( );
        int c;
        clrscr ( );
        c= sum ( );
        printf("sum = %d",c);
        getch ( );
}
int sum ( )
{
        int a,b,c;
        printf("enter 2 numbers");
        scanf ("%d%d", &a, &b);
        c = a+b;
        return c;
}
```

Output

```
enter 2 numbers
  10    20
Sum =  30
```

3) **Functions with arguments and without return values**

Calling function	Analysis	Called function

main ( )
{
--- 	
--- 	
fun (a,b);
--- 	
--- 	
}
	Arguments are passed
	
	No values are sent back
		fun (int a, int b)
		{
		---- 	
		---- 	
		}

eg:

```
main ( )
{
        void sum (int, int );
        int a,b;
        clrscr ( );
        printf(“enter 2 numbers”);
        scanf(“%d%d”, &a,&b);
        sum (a,b);
        getch ( );
}
void sum ( int a, int b)
{
        int c;
        c= a+b;
        printf (“sum=%d”, c);
}
```

Output

```
enter 2 numbers
        10 20
        sum  =  30
```

4) Functions with arguments and with return values.

Calling function	Analysis	Called function

main ( )
{
    int c;
    --- 	
    --- 	
    c= fun (a,b);
    --- 	
    --- 	
}
	→ Arguments are passed
	
	value are sent back ←
		fun (int a, int b)
		{
		    ---- 	
		    ---- 	
		
		    return c;
		}

eg:

```
main ( )
{
        int sum ( int,int);
        int a,b,c;
        clrscr ( );
        printf(“enter 2 numbers”);
        scanf(“%d%d”, &a,&b);
        c= sum (a,b);
        printf (“sum=%d”, c);
        getch ( );
}
int sum ( int a, int b )
{
        int c;
        c= a+b;
        return c;
}
```

Output

```
enter 2 numbers
10     20
Sum =  30
```

Scope

❖ "scope" of a variable determines the part of the program where it is visible

2 types

1. local scope   2. global scope

1. local scope

❖ Local scope specifies that variables defined within the block are visible only in that block and invisible outside the block.

2. global scope

❖ Global scope specifies that variables defined outside the block are visible upto end of the program.

eg:

```
        int c= 30;        /* global area */
        main ( )
        {
                int a = 10;
                printf ("a=%d, c=%d" a,c);
                fun ( );
        }
        fun ( )
        {
                printf ("c=%d",c);
        }
```

Local function area

output

```
        a =10, c = 30
        c = 30
```

Recursive Functions

- "recursion" is the process of defining something in terms of it self.
- "recursive function" is a function that calls itself again in the body of the function

Eg:

- A function fact ( ), which computes the factorial of an integer 'N' ,which is the product of all whole numbers from 1 to N
- When fact ( ) is called with an argument of 1 (or) 0, the function returns 1. otherwise, it returns the product of n*fact (n-1), this happens until 'n' equals 1.

```
Fact (5)  =5* fact (4)
          =5*4*3* fact (3)
          =5*4*3*2* fact (2)
          =5*4*3*2*1 fact (1)
          =5*4*3*2*1
          = 120.
```

```
main ( )
{
        int n,f;
        int fact (int);
        clrscr ( );
        printf ("enter a number");
        scanf ("%d", &n);
        f= fact (n);
        printf (factorial value = %d",f);
}
int fact (int n)
{
        int f;
        if ( ( n==1) || (n==0))
                return 1;
        else
```

```
                f= n*fact (n-1);
        return f;
}
```

Output

```
Enter a number  5
Factorial value  = 120
```

Preprocessor commands

- ‘preprocessor’ is a program that processes the source code before it passes through the compiler
- It operates under the control of preprocessor directives which begin with the symbol #

3 types

1) Macro substitution directives
2) File inclusion directives
3) compiler control directives

1) Macro substitution directives

- It replaces every occurence of the identifier by a predefined string.

Syntax for defining a macro

```
        # define identifier string
```

```
Eg: #define    PI   3.1415
        #define   f(x)     x *x
        #undef   PI
```

Program

```
#define wait getch( )
main ( )
{
clrscr ( );
printf (“Hello”);
wait ;
}
```

Output:

```
Hello
```

Program

```
#define wait getch( )
main ( )
{
#undef wait;
clrscr ( );
printf (“Hello”);
wait ;
}
```

Output

Error since wait is undefined before using it

2. File inclusion directives:

- An external file containing functions (or) macro definitions can be included using #include directive

Syntax

# include <filename> (or) #include "filename"

Eg:

```
#include <stdio.h>
main ( )
{
        printf ("hello");
}
```

Output

```
Hello
```

The definition of the function printf ( ) is present in <stdio.h>header file.

3. Compiler control directives

- C pre processor offers a feature known as conditional compilation, which can be used to switch ON (or) OFF a particular line (or) group of lines in a program.

Eg: #if, #else, #endif etc.

```
#define LINE 1
main ( )
{
#ifdef   LINE
        printf (”this is line number one”);
#else
        printf(‘This is line number two”);
#endif
}
```

Output

```
This is line number one
```

Structure and functions

- There are 3 ways by which the values of structure can be transferred from one function to another.

1) passing individual members as arguments to function

- Each member is passed as an argument in the function call.
- They are collected independently in ordinary variables in function header.

Eg:

```
struct date
{
        int day;
        int mon;
        int yr;
};
main ( )
{
        struct date d= {02,01,2010};
        clrscr ( );
        display (d.day, d.mon, d.yr);
        getch ( );
}
display (int a, int b, int c)
{
        printf("day = %d", a);
        printf("month = %d",b);
        printf("year = %d",c);
}
```

2. Passing entire structure as an argument to function

- Name of the structure variable is given as argument in function call
- It is collected in another structure variable in function header

Disadvantage : A copy of the entire structure is created again wasting memory

Program

```
struct date
{
        int day;
        int mon;
        int yr;
};
main ( )
{
        struct date d= {02,01,2010};
        display (d);
        getch ( );
}

display (struct date dt)
{
        printf("day = %d", dt.day);
        printf("month = %d",dt.mon);
        printf("Year = %d",dt.yr);
}
```

3. Passing the address of structure as an argument to function

- The Address of the structure is passed as an argument to the function
- It is collected in a pointer to structure in function header

Advantages:

1. No wastage of memory as there is no need of creating a copy again
2. No need of returning the values back as the function can access indirectly the entire structure and work on it.

Program

```
struct date
{
        int day;
        int mon;
        int yr;
};
main ( )
{
        struct date d= {02,01,2010};
        display (&d);
        getch ( );
}
display (struct date *dt)
{
        printf("day = %d", dt→day);
        printf("month = %d",dt→mon);
        printf("Year = %d",dt →yr);
}
```

Pointers and functions;

pass by value: Here values are sent as arguments

```
void main()
{
        void swap(int,int);
        int a,b;
        clrscr();
        printf("enter 2 numbers");
        scanf("%d%d",&a,&b);
        printf("Before swapping a=%d b=%d",a,b);
        swap(a,b);
        printf("after swapping a=%d, b=%d",a,b);
        getch();
}
void swap(int a,int b)
{
        int t;  ⎫        all these statements is equivalent to
        t=a;    ⎪        a = (a+b) – (b =a);
        a=b;    ⎬        or
        b=t;    ⎭        a = a + b;
}                          b = a – b;
                          a = a – b;
```

o/p:

```
enter 2 numbers 10 20
Before swapping a=10 b=20
After swapping a=10 b=20
```

<u>pass by reference:</u> Here addresses are sent as arguments

```
void main()
{
        void swap(int *,int *);
        int a,b;
        clrscr();
        printf("enter 2 numbers");
        scanf("%d%d",&a,&b);
        printf("Before swapping a=%d b=%d",a,b);
        swap(&a, &b);
        printf("after swapping a=%d, b=%d",a,b);
        getch();
}


void swap(int *a,int *b)
{
        int t;     ⎫
        t=*a;      ⎪
        *a=*b;     ⎬     *a = (*a + *b) – (*b = * a);
        *b=t;      ⎭
}
```

o/p:

```
enter 2 numbers 10 20
```

Before swapping a=10 b=20

After swapping a=20 b=10

Pointer to functions:

- ❖ It holds the base address of function definition in memory

Declaration

- ❖ datatype (*pointername) ( );
- ❖ The name of the function itself specifies the base address of the function. So, initialization is done using function name.
- ❖ Eg: int (*p) ( );

  p = display; if display ( ) is a function that is defined.

Program for calling a function using pointer to function

Program

```
main ( )
{
        int (*p) ( );
        clrscr ( );
        p = display;
        *(p) ( );
        getch ( );
}
display ( )
{
        printf("Hello");
}
```

```
main ( )
{
        clrscr ( );
        display ( );
        getch( );
}
display ( )
{
        printf ("Hello");
}
```

Output

```
Hello
```

ALGORITHM

ALGORITHM:

- It is a step – by – step procedure for solving a problem
- If algorithm is written in English like sentences then it is called as 'PSEUDO CODE'

Properties of an Algorithm

An algorithm must posses the following 5 properties. They are

6. Input
7. Output
8. Finiteness
9. Definiteness
10. Effectiveness

6. **Input :** An algorithm must have zero (or) more number of inputs
7. **Output:** Algorithm must produce one (or) more number of outputs
8. **Finiteness :** An algorithm must terminate in countable number of steps
9. **Definiteness:** Each step of the algorithm must be stated clearly
10. **Effectiveness:** Each step of the algorithm must be easily convertible into program 1. statements

1. **Example**

Algorithm for finding the average of 3 numbers

```
7.  start
8.  Read 3 numbers a,b,c
9.  Compute sum = a+b+c
10. compute avg = sum/3
11. Print avg value
12. Stop
```

2. **EXAMPLE:** Finding the roots of a quadratic equation, ax^2+bx+c

- There will be 2 roots for such quadratic equation

Input : a,b,c values

Output: r_1, r_2 values

Procedure : $r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$

$$r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Algorithm:

```
1. start
2. Read a,b,c values
3. Compute $d = b^2 - 4ac$
4. if d > 0 then
        a) $r_1 = -b + sqrt(d)/(2*a)$
        b) $r_2 = -b - sqrt(d)/(2*a)$
5. otherwise if d = 0 then
        a) compute r1 = -b/2a
                    r2=-b/2a
        b) print r1,r2 values
6. otherwise if d < 0 then print roots are imaginary
7. Stop
```

3. **EXAMPLE :** To print the Fibonacci series of n terms as follows.
   0,1,1,2,3,5,8,13, . . . , n terms

ALGORITHM:

```
Step 1: start
Step 2: initialize the a=0, b=1
Step 3: read n
Step 4: if n== 1 print a go to step 7. else goto step 5
Step 5: if n== 2 print a, b go to step 7 else print a,b
Step 6: initialize i=3
Step 7: if i<= n do as follows. If not goto step 7
c=a+b
print c
a=b
b=c
increment i value
goto step 7
Step 8: stop
```

4. **EXAMPLE :** To print a prime numbers up to 1 to n(i.e., with in the range from 1 to n)

ALGORITHM:

```
Step 1: start
Step 2: read n
Step 3: initialize i=1,c=0
Step 4:if i<=n goto step 5
If not goto step
10 Step 5: initialize j=1
Step 6: if j<=1 do as the follow. If no goto step 7
i)if i%j==0 increment c
ii) increment j
iii) goto Step 6
Step 7: if c== 2 print i
Step 8: increment i
Step 9: goto step 4
Step 10: stop
```

5.EXAMPLE : The total distance travelled by vehicle in 't' seconds is given by distance = ut+1/2at2 where 'u' and 'a' are the initial velocity (m/sec.) and acceleration (m/sec2).

ALGORITHM:

```
Step 1:Start
Step2 : Read t ,dt
Step 3: Set i to 1
Step 4:Set k to dt
Step 5: Read u,a
Step 6: set s to u*k+0.5*d*k*k
Step 7: Write s
Step 8: If(k<=t) and i=1 then
Begin
go to step 6
Else
Begin
read j
if(j=0)
then Begin
Set I to 0
End
Else
Begin
Set I to 1
go to step 4
End
End
Step 9: Stop
```

11. **EXAMPLE :**To perform the binary search operation

ALGORITHM:
Main Program:

```
Step 1: Start
Step 2:Read the value of n
Step 3:for i=1 to n increment in steps of 1
STEP 4 :Read the value of ith element into array
STEP5 :  Read the element(x) to be search<--binary(a,n,x)
Step 6: if search equal to 0 goto step 7 otherwise goto step 8
Step 7: print unsuccessful search
Step 8: print successful search
Step 9: stop
```

Binary Search Function:

```
Step 1: start
Step 2: initialise low to 1 ,high to n, test to 0
Step 3: if low<= high repeat through steps 4 to 9 otherwise goto step 10
Step 4: assign (low+high)/2 to mid
Step 5: if m<k[mid] goto step 6 otherwise goto step 7
Step 6: assign mid-1 to high goto step 3
Step 7: if m>k[mid] goto step 8 otherwise goto step 9
Step 8: assign mid+1 to low
Step 9: return mid
Step 10: return 0
Step 11:stop
```

12. **EXAMPLE :** Write a c program for selection sort

ALGORITHM:

```
Step 1:Start
Step 2:Initiliaze the variables I,j,temp and arr[]
Step 4:Read the loop and check the condition. If the condition is true print the
array elements and increment the I value. Else goto step 4
Step 4:Read the loop and check the condition. If the condition true then
goto next loop.
Step 5:Read the loop and check the condition. If the condition true then
goto if condition
```

Step 6: If the condition if(arr[i]>arr[j]) is true then do the following
steps temp=arr[i]
arr[i]=arr[j]
arr[j]=temp **Step**
7:increment the j value
Step 8:perform the loop operation for the displaying the sorted elements.
Step 9:print the sorted elements
Step 10:stop

13. **EXAMPLE :** Program that implements the bubble sort method

ALGORITHM:

Main Program:
1. start
2. read the value of n
3. for i= 1 to n increment in steps of 1
Read the value of ith element into array
4. call function to sort (bubble_sort(a,n))
5. for i= 1 to n increment in steps of 1
print the value of ith element in the array
    5. stop

Bubble Sort Function:
1. start
2. initialise last to n
3. for i= 1 to n increment in steps of 1
begin
4. initialise ex to 0
5. for i= 1 to last-1 increment in steps of
1 begin
6. if k[i]>k[i+1] goto step 7 otherwise goto step 5
begin
7. assign k[i] to temp assign k[i+1]
to k[i] assign temp to
k[i+1] increment ex
by 1 end-if
end inner for loop
11. if ex!=0
assign last-1 to last
end for loop
12. stop