I B.Sc (CS/IT) – Semester I
Problem Solving in C

UNIT I

General Fundamentals: Introduction to computers:

Block diagram of a computer, characteristics and limitations of computers, applications of computers, types of computers, computer generations.

Introduction to Algorithms and Programming Languages:

Algorithm–Key features of Algorithms, Flow Charts, Programming Languages – Generations of Programming Languages – Structured Programming Language- Design and Implementation of Correct, Efficient and Maintainable Programs.

UNIT II

Introduction to C:

Introduction–Structure of C Program–Writing the first C Program–File used in C Program – Compiling and Executing C Programs – Using Comments – Keywords – Identifiers – Basic Data Types in C – Variables – Constants – I/O Statements in C- Operators in C- Programming Examples.

Decision Control and Looping Statements:

Introduction to Decision Control Statements–Conditional Branching Statements – Iterative Statements – Nested Loops – Break and Continue Statement – Goto Statement

UNIT III

Arrays: Introduction–Declaration of Arrays–Accessing elements of the Array–Storing Values in Array– Operations on Arrays – one dimensional, two dimensional and multi-dimensional arrays, character handling and strings.

UNIT IV

Functions:

Introduction–using functions–Function declaration/ prototype–Function definition – function call – return statement – Passing parameters – Scope of variables – Storage Classes – Recursive functions.

Structure, Union, and Enumerated Data Types:

Introduction–Nested Structures–Arrays of Structures – Structures and Functions– Union – Arrays of Unions Variables – Unions inside Structures – Enumerated Data Types.

UNIT V

Pointers: Understanding Computer Memory–Introduction to Pointers–declaring Pointer Variables – Pointer Expressions and Pointer Arithmetic – Null Pointers - Passing Arguments to Functions using Pointer – Pointer and Arrays – Memory Allocation in C Programs – Memory Usage – Dynamic Memory Allocation – Drawbacks of Pointers

Files: Introduction to Files–Using Files in C–Reading Data from Files–Writing Data to Files – Detecting the End-of-file – Error Handling during File Operations – Accepting Command Line Arguments.

UNIT - I
Chapter – I – Introduction to Computers

COMPUTER

The word computer is derived from the Latin word "computere"means "to Calculate". Computer is an advanced electronic device that takes data and instructions as input from the user and processes these data (or information) and gives the result accurately, consistently atvery high speed as output and stores the output for future use.

Charles Babbage is known as the father of the computer, developed prototypes for the first mechanical and programmable computing machines.

CHARACTERISTICS AND LIMITATIONS OF A COMPUTER

Characteristics of a Computer:

Computer is an electronic device that takes data and instructions as input from the user and processes these data and gives the result as output. The main and most important characteristics of a computer are:

- Speed
- Accuracy
- Diligence
- Versatility
- Automation
- Storage

a) **Speed:**
The Speed of the computer is defined as the time taken by computer to perform a task. Computer is a very fast calculating device. It takes only few seconds for calculations that a human being can do it in an entire year.
The speed of computer is determined in terms of micro second (10^{-6}) , nano seconds (10^{-9}) and Pico seconds (10^{-12}).

b) **Accuracy:**
Computer is a very fast, reliable and robust machine. Each and every calculation is performed with same accuracy if we give the correct data. Otherwise it gives wrong data. The errors in computer are due to only human beings.

c) **Diligence:**
The capacity of performing repeated tasks without getting tired is called as Diligence. A computer is free from tiredness, lack of concentration, etc. It can work for hours without creating any errors.

d) **Versatility:**
The capacity of performing more than one task at the same time is called as versatility. Computer can perform different types of jobs at the same time.

e) **Automation:**
Computers are automatic devices that can perform tasks without user interaction.

f) **Storage or Memory:**
A computer has large storage to store large amount of data for future use. A computer contains internaland externalmemory.

Limitations of computers:

a) **No thinking and decision making power:**
Computer cannot think itself. It cannot take any decision without user interaction.

b) **No feelings:**
Computers have no feelings because they are machines. It cannot feel like us. It does not have any emotions, feeling, knowledge etc.

c) **No self-Intelligence:**
A computer does not have intelligence of its own to complete the task. It can do any work without user instructions.

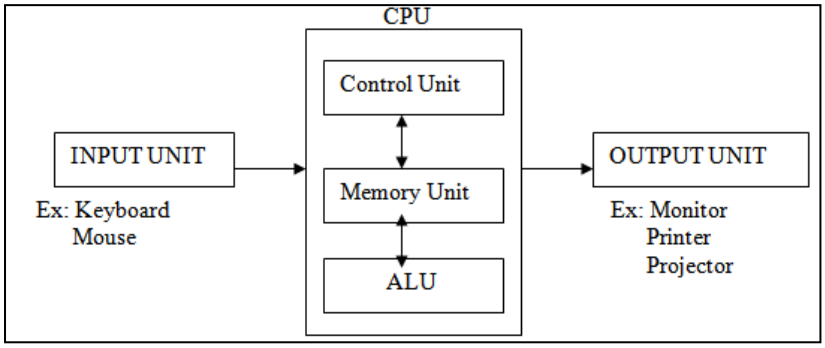
d) **No learning power:**
A computer has no learning power. It cannot do any work without user interaction.

e) **No IQ (Intelligent Quotient):**
Computer is a dumb machine and it cannot take its own decisions. Its IQ is zero till today

BLOCK DIAGRAM OF A COMPUTER (or) BASIC COMPONENTS OF PC

Basically, a computer performs five major operations (or) functions. Those are

1) It accepts data (or instructions) by input
2) It stores data
3) It can process data as required by the user
4) It gives results by output
5) It controls all the operations inside a computer



a) **Input Unit:**
Input Unit is used to send data or programs into the computer using input devices like Keyboard, mouse, scanner, joystick etc.

b) **Central Processing Unit:**
CPU stands for Central Processing Unit. The combination of ALU and CU is also known as "central processing unit". CPU is brain of any computer system.
CPU contains 3 parts – Arithmetic and Logical Unit, Control Unit and Memory Unit.

- **Arithmetic & Logic Unit (ALU):**
  ALU stands for Arithmetic & logical Unit. It is used to perform Arithmetic operations like addition, subtraction, multiplication, division and logical operations.
- **Control Unit (CU):**
  CU stands for Control Unit. This unit co-ordinates and controls the activities of all parts either internally or externally connected to the computer.

- **Memory Unit (MU) or Storage:**
  The process of saving the data permanently is known as storage. The function of memory unit is to store data or information permanently in the form of ON or OFF states. A Computer contains two types of memories - Primary memory and secondary memory.

Units of Computer Memory		
0 or 1	= 1 bit	1 Giga Byte =1024 Mega Bytes
4 bits	= 1 nibble	1 Tera Byte =1024 Giga Bytes
2 nibbles	= 8 bits= 1 byte	1 Peta byte = 1024 Tera bytes
1 word	= 32 bits = 4 bytes	1 Exa byte = 1024 Peta bytes
= 8 nibbles		1 Zetta byte = 1024 Exa bytes
1 Kilo Byte	= 1024 bytes	1 Yotta byte = 1024 Zetta bytes
1 Mega Byte	=1024 Kilo Bytes	1 Bronto byte= 1024 Yotta bytes
		1 Geop byte = 1024 Bronto bytes

c) Output Unit:
Output unit is used to produces final results by using output devices like monitor, printer, plotter, speaker etc. There are two types of output devices – soft copy devices and Hard copy devices.

CLASSIFICATION (or) TYPES OF COMPUTERS

Computers are classified into three types based on input/functionality, purpose and size.

1. Based on Input / Functionality:

Based on input, computers can be divided in to three types: Analog, Digital, and Hybrid

a. **Analog Computer:** Analog computer is used to process analog data. These types of computers works with continuous and physical quantities like temperature, pressure, volume, force, velocity and converts them into analog quantities. It produces output in the form of graph.These are used for engineering and scientific applications
   Examples: Thermometer, analog clock, speedometer, a blood pressure monitoring machine is a type of analog computer.
b. **Digital Computer:** A digital computer operates directly on digits that represent either data or symbols. It converts the data into binary digits (0's or 1's). These are used for business and scientific applications.
   Examples: IBM PC, Apple Macintosh, Calculators, Digital watches, etc.
c. **Hybrid Computers:** Hybrid computer is combination of both analog and digital computers.
   Examples: computers used in hospitals to measure the heartbeat of the patient, Devices used in petrol pump

2. Based on Purpose:

Based on purpose, computers are divided into two types – General purpose and Special purpose computers.

General purpose Computers: These computers store different programs and used in many applications. It can perform any kind of work with same accuracy.

Examples: personal computers, smart phones

Special purpose computers: These are designed to perform only specific tasks and it cannot use for any other purpose.
Examples: traffic light control system, ATM machines, Washing machines

3. Based on Size, Performance, and design:
Based on size, computers can be divided into – super, mainframe, mini and macro computers
1. Super Computer:

- Supercomputers are the fastest, most powerful and very expensive computers introduced in 1980s.
- These are used to process large amount of data and designed for specialized applications to solve complex mathematical and scientific calculations.
- A super computer supports thousands of users at the same time.
- Super computers uses parallel technology and perform more than one trillion calculations per second.
- These are mainly used for weather forecasting, scientific simulations, graphics, nuclear energy research, electronic design, aircraft design, automotive design, weapon design and analysis of geological data.
- **Examples:** CRAY-1, CRAY-2, IBM Summit

2. Main frame Computer:

- Mainframe computers are bigger, faster and more expensive than mini computers.
- These are introduced in 1975.
- A main frame computer supports hundreds or even thousands of users simultaneously.
- These are mainly used in large organizations like banks, airlines, universities etc. to store large amount of data
- **Examples**: IBM Z-series, IBM S/390

3. Mini Computer:

- Mini computers are larger, more powerful and more expensive than microcomputers.
- Mini computers are also called as Mid-range computers introduced in 1960s.
- The first minicomputer was introduced by Digital Equipment Corporation (DEC) in 1960.
- Minicomputer is a multiprocessing system that supports 200 users simultaneously
- These are mainly used in business, education, hospitals, government organizations etc.
- **Examples:** DEC, PDP-8, PDP-11 and Prime Computer

4. Micro Computer:

- Microcomputers, commonly known as Personal computer (or) Desktop computer introduced in 1970s.
- The first microcomputer was designed by IBM in 1981 named as IBM-PC
- These are very small and cheap.
- A Microcomputer is a digital computer contains one or more microprocessors, input / output units and memory.
- They are used in schools, homes, office etc.
- **Examples:** IBM-PC, Apple, Laptop, Notebook, Personal Digital Assistant (PDA)

GENERATIONS OF COMPUTERS

In computer technology, Generation means a change in technology that a computer was being used. The main generations of computers are:

1. First Generation (1946-1959):

- The period of first generation was 1946-1959.
- The first generation computers used vacuum tubes as the basic components for memory and circuitry for Central Processing Unit.
- These tubes, like electric bulbs, produced a lot of heat, were very expensive.
- Punched cards, paper tape, and magnetic tape were used as input and output devices.
- The computers in this generation used machine code as programming language.
- Examples: UNIVAC (Universal Automatic Computer), ENIAC (Electronic Numerical Integrator and Calculator)

2. Second Generation (1959-1965):

- The period of second generation was 1959-1965.
- In this generation transistors were used instead of vacuum tubes.
- These are cheaper, consumed less power, more compact in size, more reliable and faster than the first generation machines.
- In this generation, magnetic cores were used as primary memory and magnetic tape and magnetic disks as secondary storage devices.
- In this generation assembly language and high-level programming languages like FORTRAN, COBOL was used.
- Examples: UNIVAC 1108, IBM 1620, IBM 7094 series, IBM 1400 series and CDC 164 etc

Transistors

3. Third Generation (1965-1971):

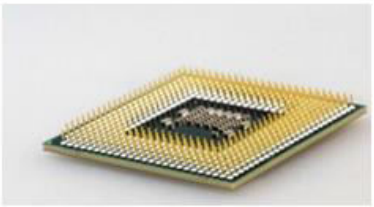
- The period of third generation was 1965-1971.
- The computers of third generation used integrated circuits (IC s) instead of transistors.
- A single IC is smaller in size, reliable and efficient.

- In this generation remote processing, time-sharing, multi-programming operating system were used.
- High-level languages (FORTRAN-II TO IV, COBOL, PASCAL PL/1, BASIC, ALGOL-68 etc.) were used during this generation.
- Examples: IBM 370, IBM System/360, UNIVAC 1108 and UNIVAC AC 9000



4. Fourth Generation (1971-1980):

- The period of fourth generation was 1971-1980.
- The fourth generation computers used Very Large Scale Integrated (VLSI) circuits.
- Fourth generation computers became more powerful, compact, reliable.
- The size, cost, power requirement, heat generation decreased compared to previous generations.
- New operating systems like DOS, windows, Linux and Graphical User Interface (GUI), mouse, handheld devices developed.
- Keyboard, Mouse is used for input and video displays, printouts are used for output devices.
- RAM, ROM, cache memory is used for primary memory and magnetic disk, floppy disk, optical disk are used for secondary memory.
- All high-level languages C, C++ are used in this generation.
- Examples: Apple Macintosh & IBM PC

5. Fifth Generation (1980 to till date):

- The period of fifth generation is 1980 till date.
- In the fifth generation, the ULSI (Ultra Large Scale Integration) technology is used.
- This generation is based on parallel processing hardware and AI (Artificial Intelligence) software.
- All the high-level languages like C and C++, Java, .Net etc., are used in this generation.
- **IBM's Watson** is example of computers used AI, whichwas featured as a contestant on the TV show Jeopardy.
  Microsoft's Cortana on Windows 8 and Windows 10 computers,**Apple's Siri** on the iPhone, **Google** search engine are examples of computers thatused AI.

Generations Duration	1st (1946-1959)	2nd (1959-1965)	3rd (1965-1970)	4th (1970-1981)	5th (1981-onwards)
Major Innovation	Vacuum Tubes	Transistors as Main component	Integrated Circuit (ICs) as basic electronic component	LSIC & VLSIC (Microprocessor)	ULSIC (Ultra Large Scale Integrated Circuit)
Main Memory	Magnetic Drums	RAM and ROM	PROM & DRAM	EPROM & SRAM	EEPROM, SIMM & DIMM
External Storage	Punched cards	Magnetic capes and magnetic disk	Improve disk (Floppy disk)	Floppy disk & Hard disk	Modified magnetic and optical disks
Input/Output devices	Punched cards & paper	Magnetic tape, Punched card, paper for output	Keyboard for input and Monitor for output	Monitor for Output	Keyboard, Pointing device, Scanner as input and Monitor as main output
Languages	Low level machine language	Assembly-language, some high level languages for example BASIC, COLOL, FORTRAN	More high level languages	Languages and application software	Al (Artificial intelligence) Expert systems
Operating system	No operating system, human operators to set switches	Human handles punched card	Complete operating system were introduced	MS-DOS & PC-DOS	GUI based e.g. Windows 95 and Windows NI
size	Main frame for example ENLAC, EDVAC, UNIVAC	Main frame for example IBM-1401, NCR-300, IBM-600 etc	Mini, for example IBM system/360 ICH-360, Honey well 316 etc	Micro computer e.g. IBM-PC, Apple, Macintosh etc	Very small in size e.g. Laptop, Note book, Digital diary, Palm top and Pocket PC

UNIT - I
Chapter – II - Algorithm

Algorithm:
An algorithm is a step-by-step process to solve a particular problem. An algorithm provides a blueprint for writing a program to solve a particular problem.

Features of an Algorithm:
The features of an algorithm is as follows
1. It should be simple.
2. It should be clear with no ambiguity.
3. It should lead to unique solution of the problem.
4. It should involve finite number of steps.
5. It should have the capability to handle some unexpected situations that may arise during thesolution of a problem.

Properties (or) Characteristics of an Algorithm:
All algorithms must satisfy the following criteria:
1. Finiteness: An algorithm must terminate after a finite number of steps and further each step must be executable in a finite amount of time.
2. Each step of an algorithm must be precisely defined.
3. Input: An algorithm has zero or more, but only finite number of inputs.
4. Output: An algorithm has one or more outputs.
5. Effectiveness:

Implementation of Algorithm:
Thealgorithm contains 4 steps. They are
(a) Declaration
(b) Input
(c) Logical processing
(d) Output

(a) Declaration:
The data after the **start** keyword with using a **declare** keyword is knownas declaration part.
(b) Input:
To define a data name at declaration part, it must be assigned some value. In this input, we require the **"READ" or "GET"** statement.
(c) Logical processing:
To define a data name and assign the value to be performed some operation between the data name by using operators like *"+, -, *, /, %, >, <, >/, /<, #"*. These are logical processing operators
(d) Output processing:
To solve the desired output format,we require "PRINT" or "GIVE" statements.

Control structures used in Algorithms: An algorithm uses three control structures

1. **Sequence:** Sequence means that each step of the algorithm is executed in the specific order.

   Example: An algorithm to add two numbers

   Step 1: Input the first_number as A
   Step 2: Input the second_number as B
   Step 3: Set sum=A+B
   Step 4: Print sum
   Step 5: End

2. **Decision:** Decision statements are used when the execution of a statements are based on condition. It can be implemented using if, if-else, switch statements

   For example, *if*$x=y$ then print "*equal*".

   Example: An algorithm to check the equality of two numbers

   Step 1: Input the first_number as A
   Step 2: Input the second_number as B
   Step 3: if A= =B
   Then print "Equal"
   else
   print "not equal"
   Step 4: End

3. **Repetition:** Repetition statements are used to execute one or more statements for a specific number of times. It can be implemented using *while, do-while,* and *for* loops.

   Example: Algorithm to first 10 natural numbers

   Step 1: initialize I=1, N=10
   Step 2: Repeat steps 3 and 4 while I<=N
   Step 3: print I
   Step 4: set I=I+1
   Step 5: End

Examples:

1. Write an algorithm to find the sum of three given numbers

Name: Sum of three numbers
Inputs: A, B, C
Output: sum
Method
Get A
Get B
Get C
Let sum = A + B + C
Give sum

2. Write algorithm to find largest value

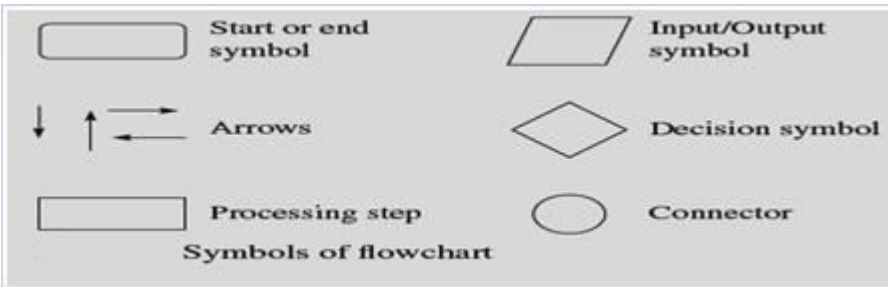
Step 1: Input first_number as A
Step 2: Input second_number as B
Step 3: if A > B
then print A
else if A < B
then print B
else
print " the numbers are equal"
Step 5: End

FLOWCHART:

A flowchart is a graphical or symbolic representation of a given problem.(ora flowchart is a pictorial representation of an algorithm)

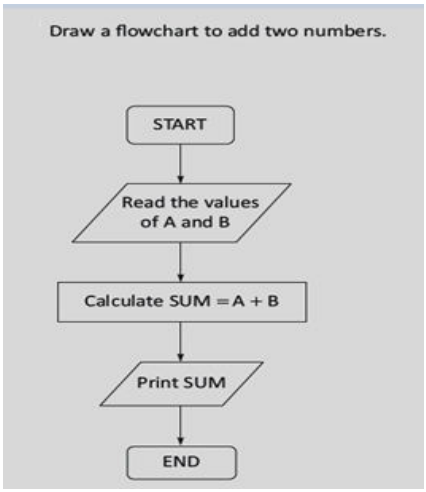
When designing a flowchart, each step in the process is shown by different symbol with a description.

Different Symbols used in Flow chart:



SNO	Notation	Purpose	Represented by
1	**Terminal** symbol	It is used to indicate the **Start** and **end**of thelogic flow	It is represented by **circles, ovals symbols**.
2	**Flow lines**	It is used to show the flow of control of the program.	It is represented by**Arrow heads**
3	**Processing** symbol	It is used to indicate arithmetic and data movement instructions	It is represented by **rectangle** symbol.
4	**Input/output** symbol	It is used to get inputs from the user or display the results to the users	It is represented by parallelogram symbol
5	**conditional** or **decision** symbol	It is used to indicate decisions	It is represented by **diamond** symbol
6	**connector** symbol	It is used to connect or join various parts of the flow lines	It is represented by **circle** symbol.

Examples:



Advantages of Flowchart:

- **Communication:** Flowcharts are a better way of communicating the logic of a system to all concerned.
- **Effective analysis**: With the help of a flowchart, a problem can be analyzed in a moreeffective way.
- **Proper documentation**: Program flowcharts serve as a good program documentation, which is needed for various purposes.
- **Efficient Coding**: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
- **Proper Debugging**: The flowchart helps in debugging process.

Limitations of Flow chart:

- **Complex logic:** Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex.
- **Alterations and Modifications**: If alterations are required, the flowchart may requireredrawing completely.

COMPUTER LANGUAGES:

A Computer language is the communication media between a user and the computer. A programming language is a set of commands, instructions, and other syntax use to create and solve a program. Computer language contains well-definedcharacter set and syntax rules.

REQUIREMENTS OF PROGRAMMING LANGUAGES:

The following are requirements of the programming language

1. Expressivity: It is the ability of a language to clearly reflect the meaning intended by thealgorithm designer (the programmer). Moreover, an expressive language embodies notationswhich are consistent with those that are commonly used in the field for which the languagewas designed.

2. Well-definiteness: this means that the language's syntax and semantics are free of ambiguity, are internally consistent, and complete.

3. Data Types and structures: This is the ability for a language to support for a variety of datavalues (Integers, rules, strings, pointers etc.) and non-elementary collections of these.

4. Modularity: It has two aspects, the language's support for sub-programming and thelanguage's extensibility in the sense of allowing programmer-defined operators and data types.

5. Input-Output Facilities: This is the support for sequential indexed and random access files,as well as its support for database and information retrieval functions

6. Portability: This is the ability of a language to be implemented on a variety of computers.

7. Efficiency: An efficient Language is one which permits fast computation and execution onthe machines where it is implemented.

8. Pedagogy: Some Languages have better pedagogy than others. That is, they are intrinsicallyeasier to teach and to learn.

9. Generality: Generality means that a language is useful in a wide range of programmingapplications.

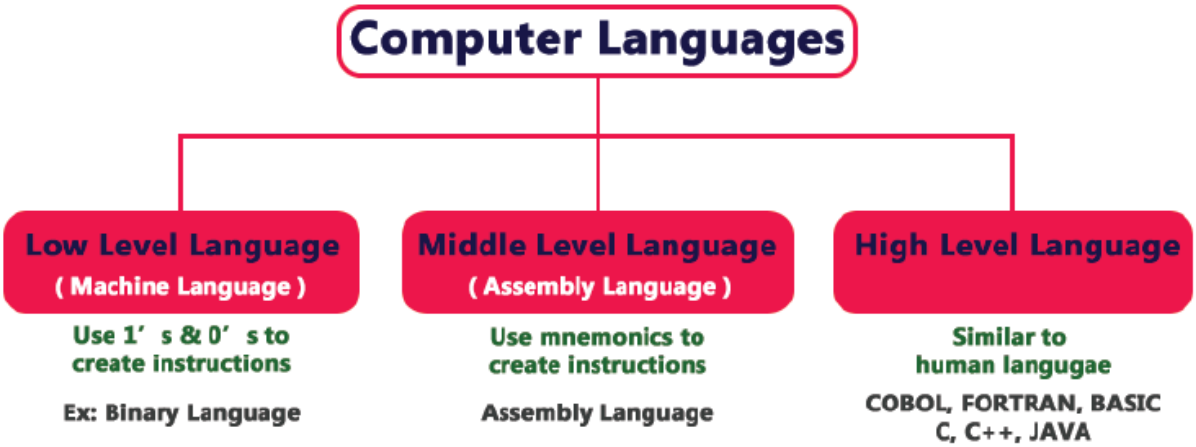
PROGRAMMING LANGUAGES and ITS TYPES

Computer languages are the languages used to communicate between a user and computer by writing program instructions. Every Computer language contains a set of pre-defined words and syntax rules to create instructions of a program.

The languages in which programs are written are called programming languages. Computer programs are mainly divided into three types of languages.

- Machine language
- Assembly language
- High level language

In general, the programs written in high level or assembly languages are called 'source programs'.The source program to be converted into the machine language is called as 'object program'. Theconvention mechanism is done by the translators.



Low-Level Language (or) Machine Language

- Low-Level language is the only language understood by the computer. It is also known as **Machine Language**
- Binary Language is an example of a low-level language. The binary language contains only two symbols 1 & 0.
- A computer can directly understand the binary language.
- As the CPU directly understands the binary language instructions, it does not require any translator.
- The Machine Language program is referred to as an 'object program'
- **Example:** the letter A is represented as 1000001

Middle-Level Language (Assembly Language)

- Middle-level language is a computer language in which the instructions are created using symbols such as letters, digits and special characters.
- This language uses mnemonics code (ADD, SUB, MUL, LDA, STA, etc) in place of 0s and 1s.
- Assembly language is an example of middle-level language.
- The computer cannot understand mnemonics, so we use a translator called Assembler to translate mnemonics into binary language.
- Assembler is a translator used to translate middle-level language into low-level language.
- The Assembly language program is referred to as a 'source program'.

High Level Language

- A high-level language is a computer language which is understood by the users.
- The high-level language is very similar to human languages or English-like words.
- The high level language is easier to understand but the computer cannot understand it.
- We use Compiler or interpreter to convert high-level language to low-level language
- *Basic, C, C++, Java, Perl and COBOL are the example of high level language (HLL)*
- High level language further categorized as:
    A. Procedural-oriented or third generation (3GL)
    B. Problem-oriented or fourth generation (4GL)
    C. Natural or fifth generation (5GL)

(A) Procedural-oriented Language (3GL)

These languages are designed to express the logic and theprocedure of a problem to be solved. It includes languages such as Pascal, COBOL, C, FORTAN, etc.

(B) Problem-oriented Language (4GL)

It allows the users to specify what the output should be, without describing all the details of howthe data should be manipulated to produce the result. These areresult-oriented and include database query language. e.g.: Visual Basic, C#, PHP, etc.

The objectives of 4GL are to:

- Increase the speed of developing programs.
- Minimize user's effort to obtain information from computer.
- Reduce errors while writing programs.

(C) Natural Language (5GL)

Natural languages are still in developing stage where we could write statements that would looklike normal sentences.

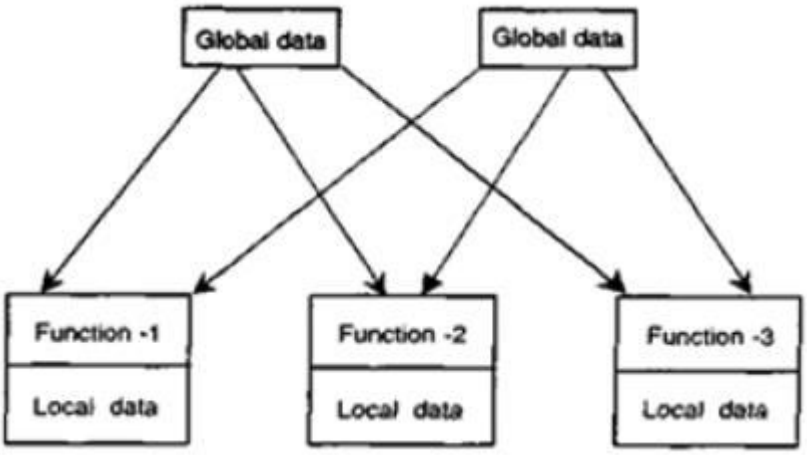
STRUCTURED PROGRAMMING LANGUAGE

The structured programminglanguages allow the program to be divided into small block called as "modules", so that program becomes easy to understand.

In order to accomplish any task, C language divides the problem into smaller modules called functions or procedures. That is, C language is called as Structured programming language.

Features:

- These languages are emphasis on algorithm rather than the data.
- Large programs are divided into smaller elements known as modules.
- Every module is independent of one another.
- The concept of control structures and data types is introduced
- Every function contains its own data and also access global data.
- It follows top-down approach.



Advantages of structured programming

1. Easy to read and understand
2. User friendly
3. Easier to maintain
4. Development is easier as it requires less effort and time
5. Easier to debug, error finding, testing and maintenance
6. Machine independent
7. C programs are efficient, fast and highly portable
8.

Dis-Advantages of structured programming:

1. Since it is machine independent, so it's take time to convert into machine code.
2. Development in this method takes longer time as it is language dependent.

DESIGN AND IMPLEMENTATION OF CORRECT, EFFICIENT AND MAINTAINABLE PROGRAMS

Programming is the process of solving a problem with computer. The entire program or softwaredevelopment process is divided into a number of phases, where each phaseperforms a well-defined task. The **five** phases in the programming process are:

1. Statement of problem: Definition of a problem can be achieved by writing down theproblem in a clear statement. Better problem definition results in faster and accurate solutions.

2. Solution development: This is the creative part of the programming process. At this stage wehave to construct a step-by-step procedure of the solution. The useful tools in development ofa solution are

1. Algorithm
2. Flowchart.

An algorithm is a descriptive step-by-step problem solving procedure and a flowchart provides visual representation of the solution.

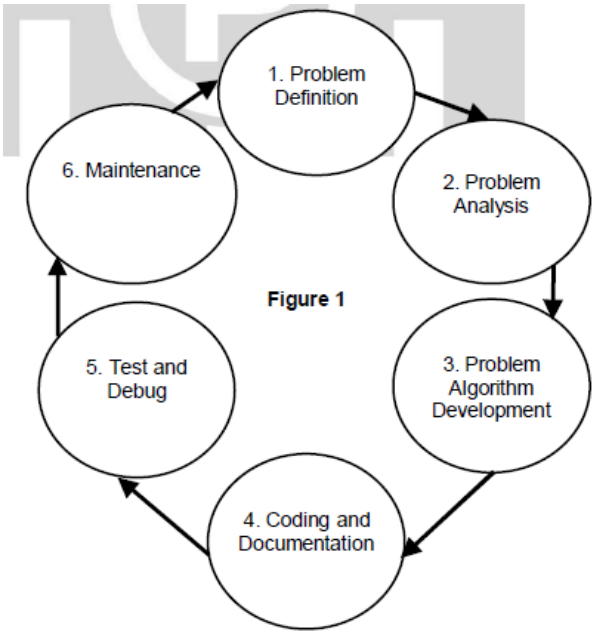


Figure 1

3. Coding: The next step in programming process is coding. This is the process of converting asolution into an actual computer program. Here, the programmer has to choose a particularprogramming language.

4. Testing & Debugging: Once coding is complete, we will enter the program into the computerto test it. Thetesting of a program involves debugging. Debugging is the process of finding out andremoving errors.

5. Documentation: The final step in the programming process is called documentation. Documentation of a program contains a detailed procedure of operation of a program.

Review Questions

1. Define Algorithm. What are the properties of an algorithm?
2. How to implement an algorithm? Give example.
3. Define flowchart. What are the various symbols used to draw a flowchart?
4. What are the advantages and limitations of flowcharts?
5. What are the requirements for programming languages?
6. Write about various programming languages and their merits and demerits.
7. How to design and implement efficient program?

UNIT - II
Chapter – I – Introduction to C

Introduction to C Language:

C is a programming language is developed by **Dennis Ritchie** at AT& T Bell Laboratories of USA in 1972. In the late seventies C began to replace the more familiar languages like PL/I, ALGOL etc., of that time. C is popular because, it is reliable, simple and easy to use.

FEATURES OF C

1. **Simple:**
   C is a simple language. It provides **structured approach** (to break the problem into parts), **rich set of library functions**, **data types** etc.
2. **Portable:**
   This feature refers to use of C language program on different platforms without any change in configuration.
3. **Structured Programming Language:**
   C is a structured programming language that is we can break the program into parts using functions. So, it is easy to understand and modify.
4. **Speed:**
   The compilation and execution time of C language is fast.
5. **Extensible:**
   C language is extensible because it can easily adopt new features
6. **Case Sensitive:**
   C is a case sensitive that is upper case and lower case characters are different.
7. **Pointer:**
   C provides the feature of pointers. We can directly interact with the memory by using the pointers.
8. **Recursion:**
   In C, we can call the function within the function.
9. **Rich Library:**
   C provides a lot of in-built functions that makes the development fast.
10. **Memory Management:**
    It supports the feature of dynamic memory allocation.

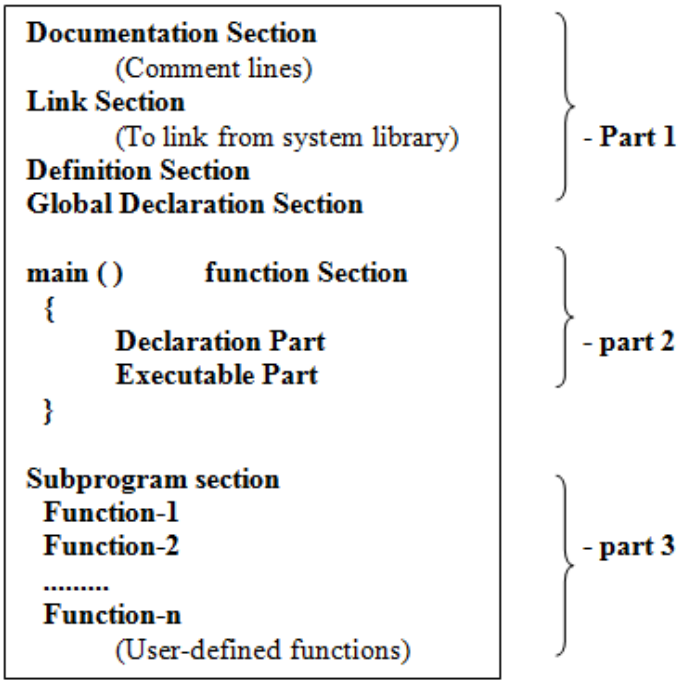
STRUCTURE OF C PROGRAM

A C program is created as a group of building blocks called as "functions". A C function contains set of statements to perform specific tasks. The basic structure of C program is as follows:

1. **Documentation section**– It contains a set of comments lines giving the author name, program name and some other details. These are ignored by C compiler. C supports tow types of comment lines: Single line comment and Multi line comment.
   Example: // single line comment
   Example: /* Sample
   C
   Program ***/**

2. **Link section**– It contains header files and symbolic constants and instructs the compiler before executing the program. It is started with **"#"** symbol followed by the **pre-processor directive** "include".

   Example: **#include<stdio.h>**
   #include<conio.h>
   #include<math.h>



3. **Definition section**– This section defines all symbolic constants in the C program.

   Example: #define PI 3.1428
   #include MAX 100

4. **Global declaration section**– It contains variables (also called as global variables) that are used in more than one function and scope is valid throughout the program.

   Example: int c;
   void main( )
   {
   ….
   }

5. **main( ) function section**- Every C program contains one **main( )** function section. This section contains two parts, **declaration part** and **executable part**. These parts placed in between the opening and closing braces.

   The declaration part declares all variable used in execution part. The program execution begins at the opening brace and ends at the closing brace. All the statements in the declaration and executable parts end with semicolon (;).

   Ex: void main( )
   {
   int a=10;
   printf ("%d", &a);
   }

6. **Subprogram section**– It contains all the user-defined functions that are called in the main( ) function. User defined functions are placed after the main( ) function

Ex:
```
void print( );
void main( )
{
print( );
}
```
```
void print( )
{
printf("user defined function");
}
```

Example:
```
//First program in C
#include<stdio.h>
#include<conio.h>
 void main()
{
 printf("Hello C Language");
 getch();
}
```

COMPILING AND EXECUTING C PROGRAM

In case of Turbo C package, the program can be typed using built-in editor. If the operating system is UNIX, the program can be typed using ED or Vi editor.

Compiling and executing C program:

Creating and Executing a C program contains following steps:

1. Creating the program
2. Compiling the program
3. Linking the program
4. Executing the program

1. Creating the Program: create a text file with its name ending with a "**.c** "extension. In case of Turbo C package, the program can be typed using built-in editor.

2. Compiling the Program: Then the source file is processed by a special program called compiler. The compiler translates the source code into an object code.

3. Linking the Program: The object code is processed by the Linker and translates object code into an executable code.

4. Executing the Program: During execution, the program may request for some data, if the program does not produce the desired result. Then it is necessary to correct the source code.

In case, the source code is modified, the entire process of compiling and execution of the program is repeated.

Follow the steps to compile and execute C program:

- Start the compiler at **C :>** prompt.
- Type as C:\tc (The compiler (TC.EXE is usually present in **C:\TC\BIN** directory).
- Select **New** from the **File** menu.
- Type the program.
- Save the program using **F2** under a proper name (say Program1.c).
- Use **Alt + F9** for Compile and **Ctrl + F9** to execute (or Run) the program.
- Use **Alt + F5** to view the output
- Use **Alt + X** to exit from C Window

CHARACTER SET

A character set denotes any alphabet, digit or special symbol used to represent information. C compiler allows us to use no. of characters:

- Lower case letters : a b c … z
- Upper case letters : A B C….Z
- Digits : 0 1 2 3…9
- Other characters: + - * ( ) & % $ # { } [ ]' " : ; etc.
- White space characters: blank, new line, tab space etc.

Backslash Character Constants:

C supports some special backslash characters constants that are used in output functions.

Constant	Meaning
'\b'	Back space
'\f'	Form feed
'\n'	New line
'\r'	Carriage return
'\t'	Horizontal tab
'\''	Single quote
'\"'	Double quote
'\\'	Black slash
'\a'	Audible alert
'\v'	Vertical tab

Delimiters in C

A delimiter is a unique character or series of characters indicates the beginning or end of a specific statement, string or function body.

Delimiter	Purpose
:	Useful for labeling
;	Statement termination symbol
{ }	Justifies the scope of the variable
( )	To define function and function call
[ ]	Used for array declaration.
#	Preprocessor Directive.
,	Variable Separator.

COMMENTS

Comments are like helping text in the C program and they are ignored by the compiler.

There are two types of comments.

1. Single Line Comments: Single line comments are represented by double slash //.
   Example: // single line comment
2. Multi line comments: Multi line comments are represented by slash asterisk /*…*/. It occupies of many lines of code.
   Example:

```
/*
Multi line
comment
*/
```

C TOKENS

The smallest individual units in a program are known as tokens. C has the following tokens.

1. Keywords
2. Identifiers
3. variables
4. Constants
5. Data types
6. Operators

1. Keywords

Keywords (or reserved words) are defined by C compiler and each word has different meaning. Keywords cannot use as names of variables, functions or labels.

Example:

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

2. Identifiers

The names of variables, functions, labels and various other user defined items are called as identifiers. Identifier contains alphabets, digits and underscore. It can't include spaces. It starts with alphabet.

Example: lcm( ), ab_cd, pi,r etc.

Rules for identifiers:

1. First character must be an alphabet (or underscore).
2. An Identifier must contain only letters, digits, or underscore.
3. Keywords are not used as identifier.
4. Commas, white spaces, tabs are not allowed
5. Allows only 31 characters of maximum length

3. Constant:

A constant refers to fixed values and its value cannot change during the execution of a program.

Syntax: **const <data type> <variable_name> = <value>;**

Example: const int a=10;

4. Variable:

A variable is an identifier used to store values and its value can be changed during the program execution. A variable takes different values at different times during execution.

Syntax: **Datatype variable_Name;**

Example:
```
int a;
float b;
char c;
```

Rules for defining variables:

- A variable contain alphabets, digits and underscore.
- A variable name must start with alphabet and underscore only.
- It can't start with digit.
- Commas, white space, tabs are not allowed within variable name.
- Keywords are not used as variable names
- Lower case and upper case letters are different.

5. Operators:

Operators are symbols which are used to perform mathematical and logical operations.
C Supports following types of operators:

1. Arithmetic operators
2. Logical operators
3. Relational operators
4. Conditional operators
5. Assignment operators
6. Bitwise operators
7. increment/decrement operators

6. Data types:

In C programming, data types are declarations for variables. Data type specifies the type and size of data associated with variables. C language supports following types of data types

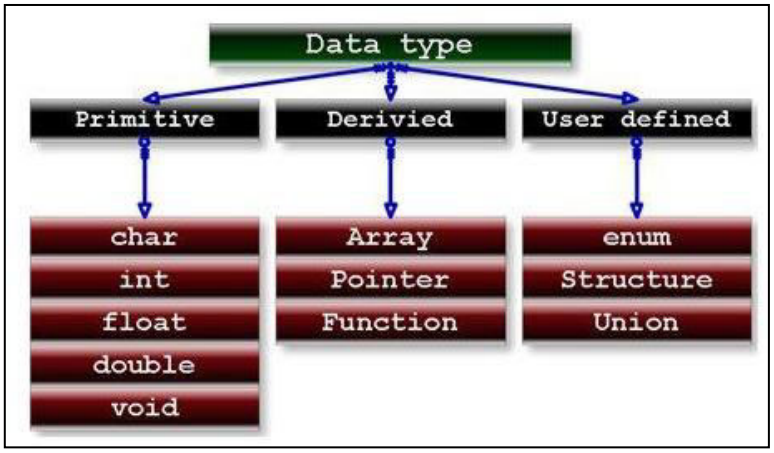
1. **Primary data types:** int, float, char, void
2. **Derived data types:**
   a) Arrays
   b) function
   c) Pointers
3. **User-Defined data types:**
   a) Structures
   b) Unions
   c) Enumerations

DATA TYPES IN C

Data types:

In C programming, data types are declarations for variables. Data type specifies the type and size of data associated with variables. C language supports following types of data types

1. **Primary data types:** The primary data types are also called as scalar (or) fundamental data types defined by C compiler.
   Example: - integer (int), floating (float), character (char) , void
2. **Derived data types:** These data types are derived from the primary (or built-in) data types.
   a) Arrays
   b) function
   c) Pointers
3. **User-Defined data types:** These data types are defined by user.
   a) Structures
   b) Unions
   c) Enumerations



1. Primitive data types: The primary data types are also called as scalar (or) fundamental data types defined by C compiler.

Example: - integer (int), floating (float), character (char), void

a) **Int:** It is used to store integer values with the range of -32,768 to 32767 and it occupies 2 bytes of memory.

b) **Float:** It is used to store a numeric values with a 6 digit precision. It occupies 4 bytes of memory.

c) **Double**: It is used to store a numeric values with a 14 digit precision. It occupies 8 bytes of memory

d) **Char:** It is used to store a single character or group of characters with the range of -128 to +127. It occupies 1 byte of memory.

e) **Void type:** Void type means no value.

Data type	Size	Min Value	Max Value	Formatted Specifier
int/short int	2	-32768	+32767	%d or %i
long/long int	4	-2147483647	+2147483647	%ld
unsigned int	2	0	65535	%u
unsigned long int	4	0	4294967295	%lu
char	1	-128	+127	%c
unsigned char	1	0	255	%c
float	4	$3.4\ e^{-38}$	$3.4\ e^{+38}$	%f
double	8	$1.7\ e^{-308}$	$1.7\ e^{+308}$	%lf
long double	10	$3.4\ e^{-4932}$	$3.4\ e^{+4932}$	%lf

2. Derived data types: These data types are derived from the primary (or built-in) data types.

a. **Arrays :** It is a collection of similar data elements
Ex: int a[10], char name[20];

b. **Pointers:** pointer is a variable which holds the address of similar datavariable.
Ex: int a=5,*p;

3. User-Defined data types: These data types are defined by user.

a. **Structures:** It is a collection of different data items combined together. All the data items stored in individual memory locations.
b. **Unions:** It is a collection of different data items combined together. All the data items can share unique memory location.
c. **Enumerations:** These data types allow declaring string constants with integer equivalent values.

TYPES OF CONSTANTS:

Constants refer to fixed values and its value cannot change during the execution of a program. C supports following types of constants.

1. Integer Constants:

An integer constant refers to a sequence of digits. There are three types of integer constants-

a) Decimal integer: It contains set of digits from 0 to 9. It may be either positive or negative.
Ex: 3445, 123, -123, -56, 0

b) Octal integer: It contains any combination of digits from the set 0 to 7 with a leading 0. Octal values have sign
Ex: 034, 037, 0551

c) Hexadecimal integer: It contains set of digits 0 to 9 and alphabets A to F to represent the values of 10 to 15. Each Hexadecimal value begins with 0x.
Ex: 0x20xF5, 0x2, 0x9F

2. Real Constants:

A real (or floating point) constant refers to numbers containing fractional parts.
Ex: 3.14, -1.2

3. Single Character Constants:

Any single character or symbol or digits placed in between single quotes is called single character constant.
Ex: 'B', '3', '!'

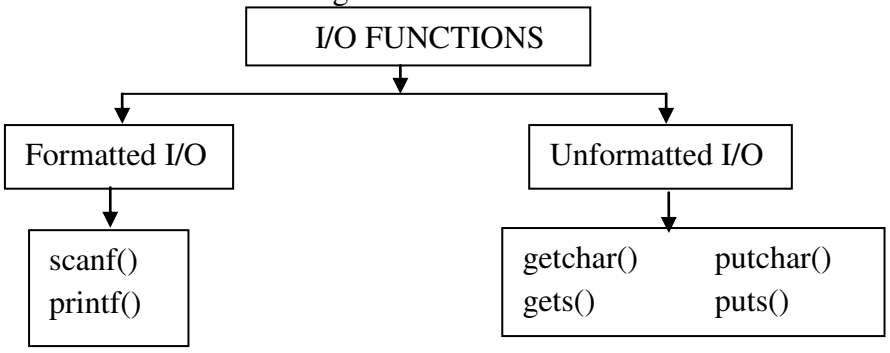
4. String Constants:

A string constant is a sequence of characters placed in between double quotes.
Ex: "vimmu", "2004", "September"

I/O STATEMENTS IN C

C programming language provides number of the built-in functions to read given input and write data on screen, printer or in any file. These built-in functions are available in "stdio.h" and "conio.h" header files.

The I/O functions are classified into 2 categories as shown below:



1. scanf(): It is a formatted input function used to accept data from the user at run time.
Syntax: scanf ("format-string", &varl ,&var2, ……..);

2. printf(): The printf ( ) function is a formatted output function used to display the given information on the standard output device.
Syntax-1: printf ("format-string");
Syntax-2: printf ("format-string", values-list);

3. getchar (): This is an unformatted I/O function used to accept a character from the keyboard and stores in a variable.
Syntax: variable=getchar();

4. putchar (): This is an unformatted I/O function used to print a character on the standard output device.
Syntax: putchar(arg)

5. gets(): This is an unformatted I/O function used to accept a string from the keyboard and stores in a variable. It stores null character '\0' at the end of the string.
Syntax: gets(variable);

6. puts(): This is an unformatted I/O function used to print a string on the standard output device.
Syntax: puts(variable);

All these input and output functions are declared in <stdio.h>

Function	Type	meaning
printf	Output	Writes formatted text on the screen
scanf	Input	Reads formatted text from keyboard
getc	Input	Reads a character from stream
putc	Output	Writes a character into a stream
gets	Input	Reads a string from keyboard
puts	output	Writes string on to the screen

Example:

```
#include<stdio.h>
#include<conio.h>
voidmain()
{
int x;
printf(" Enter the value ");
scanf("%d",&x);
printf(" The value of x is %d ",x);
char i;
i=getchar();
putchar(i);
char a[8];
gets(a);
puts(a);
getch();
}
```

OPERATORS

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types,

1. Arithmetic operators
2. Relation operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

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

1. Arithmetic operators: Arithmetic operators are used to perform arithmetic calculations.

Operator	Description	Example
+	Addition	a+b
-	Subtraction	a-b
*	Multiplication	a*b
/	Division	a/b
%	Modulo Division	a%b

Example:

```
#include<stdio.h>
#include<conio.h>
voidmain()
{
        inta,b,c;
        a=20,b=3,c=2;
        printf(" \n The sum of a, b, and c is %d ",(a+b+c));
        printf(" \n The subtraction of a and b is %d ",(a-b));
        printf(" \n Multiplication of a and b and c is %d ",(a*b*c));
        printf(" \n Division of a and b is %d ",a/b);
        printf(" \n Modulo of a and b is %d ",a%b);
        getch();
}
```

Types of Arithmetic Operators:

a) Integer Arithmetic: When both the operands in a single arithmetic expression are integers, the expression is called an integer expression, and the operation is called integer arithmetic.

Example: a=14 and b =4 then a-b=10

b) Real Arithmetic: An arithmetic operation contains only real operands is called real arithmetic.

Example: X=6.0/7.0 = 0.857143

c) Mixed-mode Arithmetic: When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression, and the operation is called Mixed arithmetic.

Example: 15/10.0=1.5

2. Relational operators: The relational operators are used to compare two values and gives either true (1)or false (0) result. The following are the relational operators.

Operator	Description	Example (a=10,b=20)
==	Equal to	(A == B) is not true.
!=	Not equal to	(A!= B) is true.
>	Greater than	(A > B) is not true.
<	Less than	(A < B) is true.
>=	Greater than or equal to	(A >= B) is not true.
<=	Less than or equal to	(A <= B) is true.

Example:

```
#include<stdio.h>
#include<conio.h>
voidmain()
{
        inta,b;
        a=20,b=3;
        printf(" \n a>b %d ",a>b);
        printf(" \n a<b %d ",a<b);
        printf(" \n a>=b %d ",a>=b);
        printf("\n a!=b %d ",a!=b);
        getch();
}
```

3. Logical operators: These are used to combine two or more relational expressions and give the result either true or false.

Operator	Description	Example(a=1,b=0)
&&	Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Logical NOT Operator. It is used to reverses the logical state of its operand. If condition is true then Logical NOT Operator will FLASE	!(A && B) is true.

Ex: if (age>30 && salary>15000), if(number<0 || number>100)

4. Assignment operators: Assignment operators are used to assign the result of an expression to a variable.

Operators		Example	Explanation
Simple assignment operator	=	sum=10	10 is assigned to variable sum
Compound assignment	+=	sum+=10	This is same as sum=sum+10
	-=	sum-=10	This is same as sum = sum-10

operators	*=	sum*=10	This is same as sum = sum*10
	/+	sum/=10	This is same as sum = sum/10
	%=	sum%=10	This is same as sum = sum%10
	&=	sum&=10	This is same as sum = sum&10
	^=	sum^=10	This is same as sum = sum^10

5. Increment / Decrement operator: (++ /--)

The increment operator is a unary operator. It is used to increase the value of an operand by 1. The decrement operator is a unary operator. It is used to decrease the value of an operand by 1.

It is used again two ways. They are pre-increment, post-increment and pre-decrement, post-decrement.

Pre-increment: ++m; Post-increment: m++;
Pre-decrement: --m; Post-decrement: m--;

Example:

```
#include<stdio.h>
#include<conio.h>
voidmain()
{
        int a=1,b=2;
        printf(" \n Post increment  of a is %d ",a++);
        printf(" \n post decrement of b is %d ",b--);
        printf(" \n pre increment of a  is %d ",++a);
        printf(" \n pre decrement of b is %d ",--b);
        getch();
}
```

6. Conditional operators:

It is also known as Ternary Operator. Conditional operator contains a condition followed by two statements or values. If the condition is true the first statement is executed otherwise the second statement is executed.

Syntax: Exp=condition ?exp1 : exp2;

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num;
printf("Enter the Number : ");
scanf("%d",&num);
(num%2==0)?printf("Even"):printf("Odd");
}
```

8. Special operators:

The following are the special operators used in 'C' language.

- Comma operator (,)
- Sizeof operator : This operator returns the size of an operand
- Pointer operator (& and *)
- Member selection operators (. And ->)

9. Bitwise Operators: C supports special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing bits or shifting them to right or left. Bitwise operators may not be applied to float or double data type.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR (Exclusive OR)
~	1's complement
>>	Right shift
<<	Left shift

P	Q	P&Q
1	1	1
1	0	0
0	1	0
0	0	0

P	Q	P|Q
1	1	1
1	0	1
0	1	1
0	0	0

P	Q	P^Q
1	1	0
1	0	1
0	1	1
0	0	0

P	~P
1	0
0	1

Left shift

A	0100101
A<<3	0101000
A<<5	0100000

Right shift

A	0100101
A>>3	0000100
A>>5	0000001

Bitwise AND(&)

The output of logical AND is 1 if both the corresponding bits of operand is 1. If either of bit is 0 or both bits are 0, the output will be 0.

Example:

12 = 00001100 (In Binary)
25 = 00011001 (In Binary) Bit
Operation of 12 and 25
00001100 & 00011001 = 00001000 = 8 (In decimal)

Bitwise OR(|)

The output of bitwise OR is 1 if either of the bit is 1 or both the bits are 1. In C Programming, bitwise OR operator is denoted by |.

12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise OR Operation of 12 and 25 00001100 | 00011001 = 00011101 = 29 (In decimal)

Bitwise XOR(exclusive OR) operator

The output of bitwise XOR operator is 1 if the corresponding bits of two operators are opposite.

12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise XOR Operation of 12 and 25 00001100 ^ 00011001 = 00010101 = 21 (In decimal)

Bitwise compliment operator

Bitwise compliment operator is an unary operator(works on one operand only). It changes the corresponding bit of the operand to opposite bit,i.e., 0 to 1 and 1 to 0. It is denoted by ~.

35=00100011 (In Binary)
Bitwise complement Operation of 35 ~ 00100011 = 11011100 = 220 (In decimal)

Right Shift Operator

Right shift operator moves the all bits towards the right by certain number of bits which can be specified. It is denoted by >>.

Left Shift Operator

Left shift operator moves the all bits towards the left by certain number of bits which can be specified. It is denoted by <<.

FILES USED IN C PROGRAM:

UNIT - II
Chapter – II – Decision Control and Looping Statements

CONTROL STRUCTURES

Normally, statements in C program are executed sequentially. This is called sequential execution.

When a program breaks the sequential flow and jumps to another part of the code, it is called branching. When the branching is based on particular condition, it is known as conditional branching.

The control statements are classified into 2 types:

1. Decision making and branching statements
2. Decision making and looping statements

1. **Decision making and branching statements**: These are used to execute the statements in sequential order. C supports following types of decision making and branching statements
   a. IF statement
   b. IF-ELSE statement
   c. NESTED IF-ELSE statement
   d. SWITCH statement

2. **Decision making and looping statements**: These are used to execute the statements for a specific number of times repeatedly. C supports following types of loop statements
   a. WHILE
   b. DO-WHILE
   c. FOR

CONDITIONAL BRANCHING STATEMENTS:

The conditional branching statements are used to execute group of statements based on condition. C supports following types of branching statements

1. If statement
2. If-else statement
3. Nesting of if - else
4. Else if ladder
5. Switch statement
6. Conditional operator

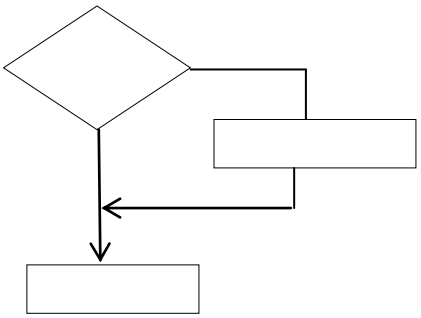
1. IF statement: It is a powerful decision making statement used to control the flow of execution of statements.

Syntax:

```
if(condition)
{
Statement-1;
}
Statement-x;
```

During the process, first the **condition** is evaluated. If the condition is true, then the statements inside **if block** is executed, otherwise the "if block" is skipped and the execution will jump to outside of the "if block".

Flowchart



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int age;
printf(" Enter the age value:");
scanf("%d",&age);
if(age>18)
{
printf(" age is greater than 18” );
}
printf(" age is below 18 “ );
getch();
}
```

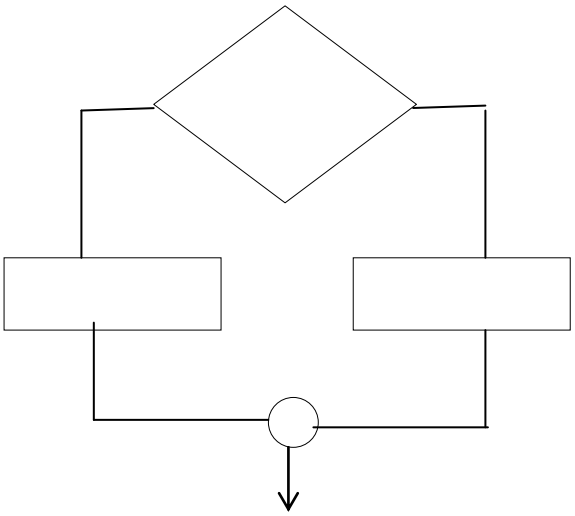
2. IF-ELSE STATEMENT: It is similar to IF statement, but additionally contains ELSE statement.

Syntax:

```
if(condition)
{
Statements;
}
else
{
Statements;
}
```

During the process, first the **condition** is evaluated. If the condition is true, then the **statements** inside the **if block** is executed. Otherwise, the **statements** inside the **else block** is executed.

Flow Chart:



Example: //biggest of 2 numbers

```
#include<stdio.h>
#include<conio.h>
void  main()
{
int a,b;
printf(" Enter a,b values:");
scanf("%d%d",&a,&b);
if(a>b)
{
printf("a is big");
}
else
{
printf("b is big");
}
getch();
}
```

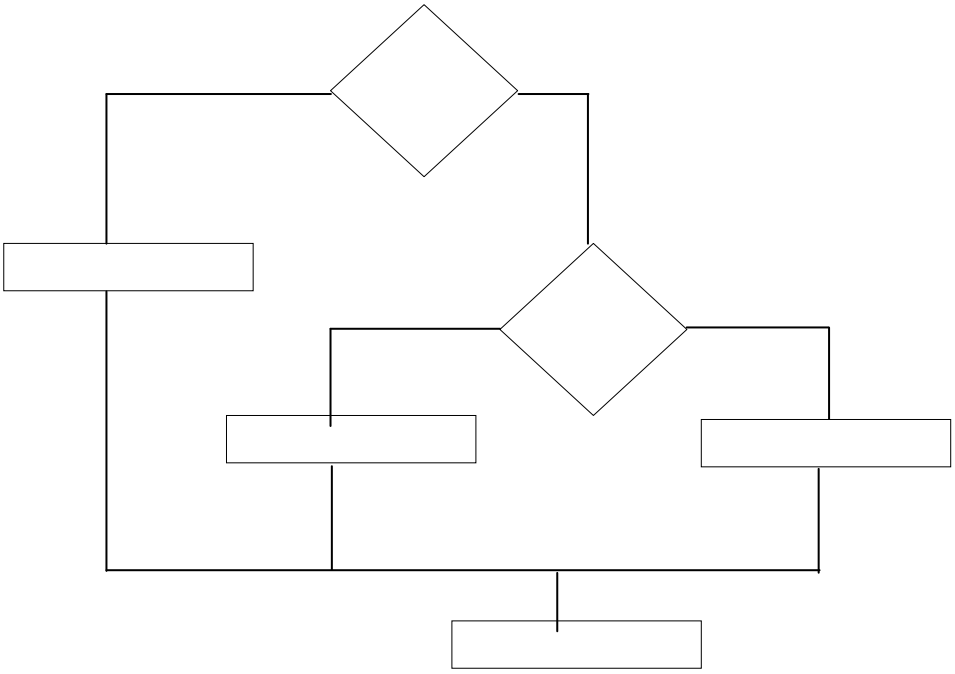
3. NESTED IF STATEMENT: When an IF statement contains another IF statement then it is called as "Nested if" statement.

Syntax:

```
if (condition-1)
if(condition-2)
statemet-1;
else
statement-2;
else
statement-3;
```

During the process, If the **condition-1** is false **statement -3** will be executed, otherwise **condition -2** is evaluated. If the **condition – 2** is true, the **statement -1**is executed, otherwise the **statement -2**is executed.

Flow chart:



Example: //biggest among 3 numbers

```
#include<stdio.h>
#include<conio.h>
void  main()
{
int a,b,c;
printf(" \n Enter a,b,c values:");
scanf("%d%d%d",&a,&b,&c);
if(a>b)
{
if(a>c)          printf(" %d is big",a);
else             printf("%d is big",c);
}
else
{
if(b>c)          printf("%d is big",b);
else             printf("%d is big",c);
}
getch();
}
```

4. ELSE-IF LADDER

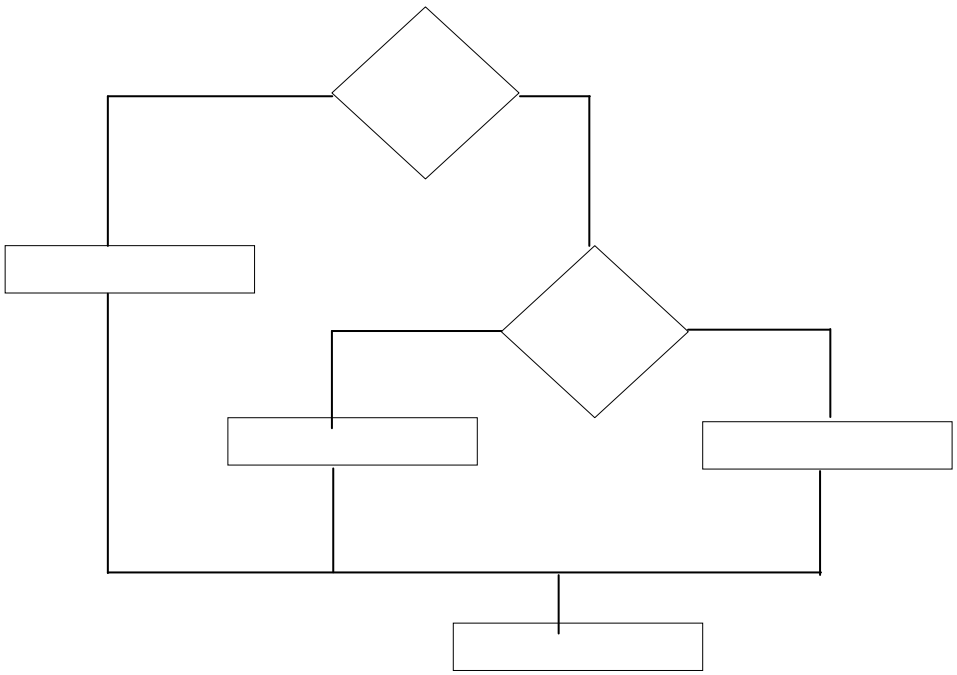
It is used when we have multiple conditions and we have to execute the related statements based on which condition is true.

Syntax:

```
if(condition1)
{
        Statement1;
}
else if (condition2)
{
        Statement2;
}
else
{
        Statement3;
}
```

During the process, If the **condition-1** is true **statement -1**is executed, otherwise **condition – 2** is evaluated. If **condition -2** is true, the **statement -2**is executed, otherwise the **statement -3**is executed. When all the conditions are False then **statement – 3** is executed.

Flow chart:



Example:

```
#include<stdio.h>
#include<conio.h>
void  main()
{
int a =10;
if( a>10)
   printf("a>10");
else if( a>20)
   printf("a>20");
else if(a<10)
   printf("a<10");
else
   printf("Not matched");
getch();
}
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int a, b, c;
printf("enter 3 numbers:");
scanf("%d%d%d",&a,&b,&c);
if(a = = b && a = = c)
printf("three are equal");
else if(a>b && a>c)
printf(" big= %d",a);
else if(b>c)
printf(" big= %d",b);
else
printf("big= %d",c);
getch( );
}
```

5. SWITCH STATEMENT:

The switch statement is a multiple-way condition statement. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. Otherwise, the default statement is executed

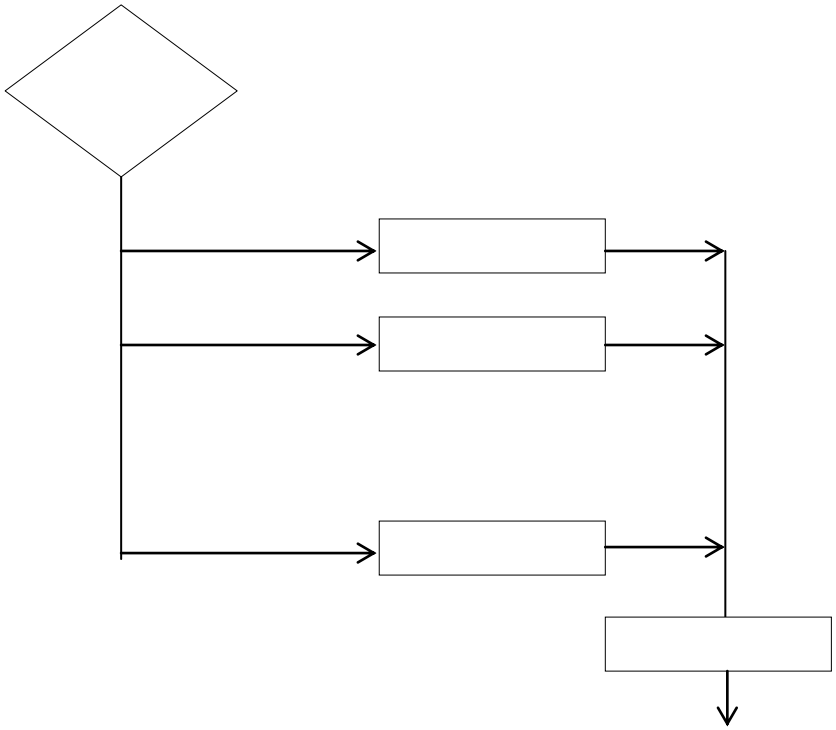
Syntax:

```
switch(expression)
{
case value1 :    statements;
                 break;

case value2 :    statements;
                 break;

case value3 :    statements;
                 break;
default :        statements;
}
```

Flow chart:



Example 1:

//program to perform arithmetic operators using switch case.

```
void  main( )
{
char op;
int a=10,b=5;
clrscr( );
printf("enter your operator +,-,*,/,%");
scanf("%c",&op);
switch(op)
{
case '+': printf("sum=%d",a+b);
          break;
case '-': printf("difference=%d",a-b);
          break;
case '*': printf("mul=%d",a*b);
          break;
case '/': printf("div=%d",a/b);
          break;
case '%': printf("rim=%d",a%b);
          break;
default: printf("invalid");
}
getch();
}
```

Example-2:

```
void  main()
int n;
clrscr();
printf("Enter a digit");
scanf("%d",&n);
switch(n)
{
case 1: printf("Sunday");
          break;
case 2: printf("Monday");
          break;
case3: printf("Tuesday");
          break;
case4: printf("Wednesday");
          break;
case5: printf("Thursday");
          break;
case6: printf("Friday");
          break;
case7: printf("Saturday");
          break;
default: printf("no match found");
}
getch();
}
```

DECISION MAKING AND LOOPING:

The process of executing group of statements for a number of times repeatedly is known as Looping. The C language supports following types of loop statements

1. WHILE statement
2. DO-WHILE Statement
3. FOR statement

1. WHILE Statement:

It is used to execute one or more statements repeatedly until the given condition is satisfied.It is an entry-controlled loop statement (i.e., condition is tested before executing the body of the loop)

Syntax:

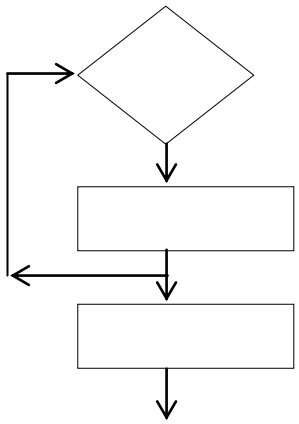
```
Initialization;
while (condition)
{
    //body of the loop
    Increment/decrement;
}
Statements;
```

In while loop, first the **condition** is evaluated. If the condition is true then the body of the WHILE loop is executed. After that, again **condition** is evaluated and if it is true, the body of the WHILE loop is executed. This process is repeated until the **condition** becomes false and the control is transferred out of the loop.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i=1,n;
printf("\n Enter limit:");
scanf("%d",&n);
while(i<=n)
{
printf(" %d ",i);
i++;
}
getch();
}
```

Flow chart:



2. DO-WHILE Statement:

It is used to execute one or more statements repeatedly until the given condition is satisfied. It is exit control loop statement. It executes atleast one statement even the condition is false.

Syntax:

```
Initialization;
do
{
    //body of the loop
    Increment/decrement;
}while (condition);
```

In this, first the statements are executed. At the end, the **condition** is evaluated. If the condition is true, then the body of the loop is executed. This process continues until the condition is false. When the condition becomes false, the loop will be terminated.

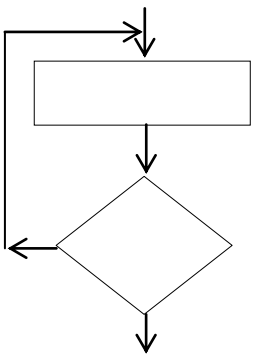
Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i=1,n;
printf(" Enter limit:");
scanf("%d",&n);
do
{
printf(" %d ",i);
i++;
}while(i<=n);
getch();
}
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x=1, sum=0;
do
{
sum=sum+x;
x++;
}
while(x<=10);
printf(" The sum is: %d
",sum);
getch();
}
```

Flow chart:



3. FOR statement: The 'for' loop is used to execute a set of statements until given condition is satisfied. The 'for' loop statement is contains three sections. The sections are separated by semi-colon. It is an entry-controlled loop.

Syntax:

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

During the process, initialization will be done after that condition is evaluated. When the condition is true then the body of for loop is executed and that updation is performed. Again condition is evaluated. This process continues until condition become false.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
clrscr();
printf(" Enter limit:");
scanf("%d",&n);
for(i = 1; i <= n; i++)
printf("%d ",i);
getch();
}
```

JUMP STATEMENTS:

1. GOTO Statement:

The goto statement is an unconditional statement used to transfer the control from one place to another place within a program.

Syntax: goto label;
Label: statement;

Ex: Forward Jump

```
void main()
{
int num;
printf("Enter a value");
scanf("%d",&num);
if(num<0)
goto end;
printf("value=%d",num);
end:
printf("end of program");
}
```

Ex: Backward Jump

```
void main()
{
int num;
input:
printf("Enter a value");
scanf("%d",&num);
if(num<0)
goto input;
printf("value=%d",num);
}
```

2. BREAK statement:

The break statement is used to terminate the execution of the loop. It is used in switch statement. It is used in order to exit from a loop (or) from a switch statement.

Syntax: Break;

Example:

```
for(i=1;i<10;i++)
{
printf("%d",i);        //prints from 1 to 5
if(i==5)
  break;
}
```

3. CONTINUE statement:

It is just like break statement, C supports another similar statement called the continue statement. It causes the loop to be continued with the next iteration after skipping any statements in between.

Syntax: Continue ;

Example:

```
for(i=1;i<10;i++)
{
printf("%d",i);        //prints all except 5
if(i==5)
  continue;
}
```

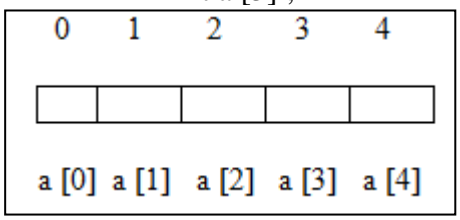
UNIT - III
ARRAYS

Q. Explain about Arrays in C

Array:

An array is a collection of same type of elements that share a common name. The elements of the array are stored in consecutive memory locations and are referred by an index (also known as subscript). The size of the array is started from zero and ends with size-1.

a) **Declaration of an Array:** An array must be declared and defined before it can be used.
Syntax for creating an array: Data_type array-name [size];
Example: int a [5] ;



The above statement declares an array containing 5 elements. In C, the array index (size) starts form "zero" , a[0], the second element in a[1], and so no. Therefore the last element, i.e the 5th element will be stored in a[4].

b) **Initialization of an array:** After an array is declared, its elements must be initialized
Syntax:
data type array name [size] = {no.of elements};
Example:
int a [5] = {10,20,30,40,50};

c) **Accessing array elements:**
All the elements in an array are individually accessed by using index numbers. The index number of array always starts with '0' and ends with 'size-1'.
Syntax: array_name[index];
Example: arr[4];

Advantages of Arrays:

1) Array can store multiple values of same data type.
2) All the elements in array are accessible by using single name (array name). It overcomes the name conflict problem.
3) It is capable of storing many elements at a time.
4) The memory locations of elements in the array are sequential.
5) Sorting and Searching becomes easy.

Disadvantages of arrays:

1) Memory wastage will be there.
2) To delete an element in an array we need to traverse throughout the array.
3) The size of the array is fixed while declaration itself.
4) The size can't be increased/decreased dynamically.

Q. Explain about different types of Arrays in C

Array: An array is a collection of same type of elements that share a common name.

Types of arrays:

1. One Dimensional Arrays
2. Two Dimensional Arrays
3. Multi-Dimensional Arrays

1. ONE DIMENSIONAL ARRAY:

When array is declared with only one dimension (or subscript) then it is called one dimensional array or single dimensional array.

a) **Declaration of one-dimensional array:** An array must be declared and defined before it can be used.

Syntax: Datatype array_name [size];

Ex: int a[5];

The above statement declares an array containing 5 elements. In C, the array index (size) starts form "zero" , a[0], the second element in a[1], and so no. Therefore the last element, i.e the 5^{th} element will be stored in a[4].

b) **Initialization of arrays:**

After an array is declared, its elements must be initialized. Otherwise, they will contain garbage value.

Syntax: Data type array_name [size]={no. of elements};

Ex: int a[5]={5,7,3,9,1};

c) **Accessing array elements:**

Once an array is defined, its elements can be accessed by using an index or subscript.

Syntax: array_name[index];

Example: a[4];

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],n,i;
clrscr();
printf("\n Enter the size of the array:");
scanf("%d",&n);
printf("\n Enter the elements into array\n");
for(i=0;i<=n-1;i++)
{
scanf("%d",&a[i]);
}
printf("\n The array elements are\n");
for(i=0;i<=n-1;i++)
{
printf("%3d",a[i]);
}
getch();
}
```

2. TWO-DIMENSIONAL ARRAY

When an array uses only two subscripts then it is called "Two dimensional arrays". It can be viewed as table of elements, which contains rows and columns.

a) **Declaration of two-dimensional arrays:** Similar to one-dimensional array, two-dimensional array declared and defined before it can be used

Syntax: data type array_name[row_size][col_size]

Example: int marks[2][3];

Example: int marks[3][5];

Rows / Columns	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

b) **Initialization of 2-D arrays:** Similar to one-dimensional array, two-dimensional array initialized by list of values enclosed in braces. The initialization is done row by row.

Syntax: data_type array_name[row][col]={ list of values};

Example: int marks[2][3]={90,85,74,88,66,86};

or

int marks[2][3]={{86,89,66},{56,75,78}};

c) **Accessing the Array elements:**

Two dimensional arrays contains two subscripts, we will use two for loop to access the elements

Syntax: array_name[row][col];

Example: marks[1][1];

Example:

```
#include<stdio.h>
#include<conio.h>
voidmain()
{
int a[3][3],i,j;
clrscr();
printf(" \n Enter the elements for Matrix A ");
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("\n The elements of Matrix A are :\n");
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
printf("%3d",a[i][j]);
}
printf("\n");
}
getch();
}
```

3. MULTI DIMENSIONAL ARRAYS:

Array of arrays is known as multi-dimensional array. Like one index in 1D array, two indices in 2D array, in the same way a multi-dimensional (or) n-dimensional array contains *n* indices.

A multi-dimensional array is declared and initialized similar to one-dimensional and two-dimensional arrays.

Syntax: data_type Array_name [table][row][col];

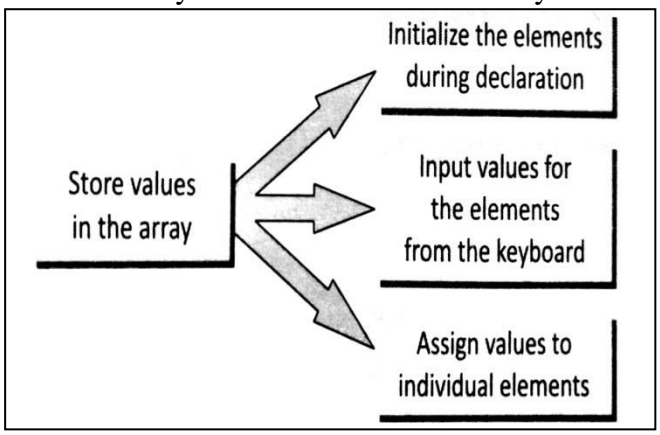
Example: int a[2][3][2] = {{{1,2},{3,4},{5,6}},{{7,8},{9,10},{11,12}}}

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[3][3][3],i,j,k;
clrscr();
printf(" \n Enter the elements for Matrix A ");
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
for(k=0;k<2;k++)
{
scanf("%d",&a[i][j][k]);
}
}
}
printf("\n The elements of Matrix A are :\n");
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
for(k=0;k<2;k++)
{
printf("%3d",a[i][j][k]);
}
printf("\n");
}
}
getch();
}
```

Q. STORING VALUES IN ARRAY:

When we declare and define an array, we are just allocating space for the elements. No elements are stored in the array. There are three ways to store values in the array



1) First, to initialize the array elements:

Elements of the array initialized at the time of declarations as other variables. When an array is initialized, we need to provide a value for every element in the array

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

Ex: int marks[5]={1,2,4,5,8};

2) Second, to input values for every individual element

An array is filled by inputting values from the keyboard. In this method, a while / do-while / for loop is executed to input the values for each elements of the array

Ex: int i, marks[10];
for(i=0; i<10;i++)

scanf("%d", &marks[i]);

3) Third, to assign values to the elements

The third way is to assign values to individual elements of the array by using the assignment operator.

int marks[3] = 77;

here, 77 is assigned to the fourth element of the array which is specified as marks[3].

Q. CALCULATING THE LENGTH OF THE ARRAY

Length of the array is given by the number of elements stored in it. The general formula to calculate the length of the array is,

Length=upper_bound - lower_bound + 1

Where upper_bound is the index of the last element and lower_bound is the index of the first element in the array.

Q. OPERATIONS ON ARRAYS

There are number of operations that can be performed on arrays. These operations include.

1. Traversing an array
2. Inserting an element in an array
3. Deleting an element from an array
4. Merging two arrays
5. Searching an element in an array
6. Sorting an array in ascending or descending order

1. Traversing an array:

Traversing an array means accessing each and every element of the array for a specific purpose.

2. Inserting an element in an array:

Inserting an element in array means adding a new data element to an already existing array. If the element has to be inserted at the end of the existing array, then the task of inserting is quite simple.

3. Deleting an element from an array:

Deleting an element from an array means removing a data element from an already existing array.

4. Merging two arrays:

Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array.

5. Searching an element in an array:

Searching means to find whether a particular value is present in the array or not. If the value is present in the array then searching is said to be successful and the searching process gives the location of that value in the array. Otherwise, if the value in not present in the array, the searching said to be unsuccessful.

6. Sorting an array:

Sorting means the process of arranging the elements of an array in a particular order, either ascending or descending order.

Q. Write about strings in C (or) Character Array in C

STRING:

A string is a collection of characters placed within the double quotes. A string contains letters, digits and various special characters.

Example: "Hello" , "123" , "10+20" , "@"

In C language a string is nothing but a null-terminated character array. This means that after the last character, a null character ('\0') is stored to signify the end of the character array.

Example: char str[]="hello";

a) **Declaring string:** In C, a string variable is declared as an array of characters.

Syntax: char string_name [size];

here, ***size*** specify the maximum number of characters in the string_name

Example: char name[10];
char city[10];

When the compiler assigns a character string to character array, it automatically supplies a null character (\0) at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one

b) **Initializing string:** C allows us to initialize a string without specifying no. of elements. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

Syntax: datatype string_name[size] = {values};

Ex: char str[ ] = { 'h', 'e', 'l', 'l', 'o' };

Or

char str [10] = "hello";

H	E	L	L	O	\0				

The above declarations initialize a variable to the string "Hello", it creates 6 elements array **str** containing the character 'H', 'E', 'L', 'L','O', and '\0'.

Q. Explain about string input/output operations

READING STRINGS:

The string can be read from the user by using three ways

1. Using scanf( )
2. Using gets( )
3. Using getchar(),getch() and getche( )

1. scanf (): The scanf () function is used to read the string.

We use the % s format specification to read string of characters. No ampersand is required before the variable. While reading a value using this function there should not be any white space. The string is terminated once the white space is encountered.

Example: char college[10];
scanf("%s",college);

2. gets(): This function is used to read characters until enter key is pressed.

Syntax: gets(variable_name);

Example: char city[10];
gets(city)

3. getchar(): This function is used to read a single character at a time. This reading is terminated when new line character ('\n') is encountered.

Example: getchar(ck);

WRITING STRINGS:

The string can be displayed on the screen using three ways.

1. Using printf() function.
2. Using puts() function.
3. Using putchar() function.

1. printf(): This function is used to print the string on the screen. We use %s format specification.

Ex: printf("%s",city);

2. puts(): The puts() functions is used to display the string on the screen.

Syntax: puts(variable_name);

Ex: puts(city)

3.putchar(): This function is used to print or display one character at a time on the screen.

Ex: putchar(city);

Q. Explain about String Handling Functions (or) String Library Functions

The C-language provides the several functions used to manipulate the strings. These functions are available in the header file <string.h>.

1. strlen()

It is used to find the length of the string. That means, it counts and returns the no. of characters present in the string.

Syntax: strlen(string_name)
Example: strlen(ch);
Example2:
```
int n;
char st[20] = "Bangalore";
n = strlen(st);
```

2. strcmp()

It is used to compare two strings. It returns 0(zero) if two strings are equal. If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value

Syntax: strcmp((string1,string2);
Example :
```
char city[20] = "Madras";
char town[20] = "Mangalore";
strcmp(city, town);
```

3. strlwr()

This function is used to convert all the uppercase letters in the given string into the lower case.

Syntax: strlwr(string);
Example: strlwr("Hello World");

4. strupr()

This function is used to convert all lower case letters in the given string into the upper case.

Syntax: strupr(string);
Example: strupr("Hello World")

5. strcat ()

This function is used to concatenate (i.e., appending the content of one string to another string) two given strings.

Syntax: strcat(string1,string2);
Example: strcat("Hello","World");

6. strrev ()

This function is used to reverse all the characters in the given string except null character.

Syntax: strrev(string);
Example: strrev("Hello");

7. strcpy( ):

It is used to copy the content of one string into another string. The content of the string are unchanged.

Syntax: strcpy(dest_string,source_string);
Example: strcpy(b,a);

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char str1[30] = " Sahithya";
char str2[30] = " Degree College";
int n,len;
clrscr( );
printf("string1 =%s\n",str1);
printf("string2 =%s\n",str2);
strcat( str1, str2);
n = strcmp(str1, str2);
if(n= =0)
  printf(" both strings are identical \n");
else
  printf(" strings are different");
len= strlen(str1);
  printf(" the length of a string =%d \n", len);
printf( " string before strrev : %s\n", str1);
printf(" string after strrev :%s\n", strrev(str1));
printf( " %s\n", strupr(str1));
printf( " %s\n", strlwr(str1));
getch( );
}
```

UNIT - IV
Chapter – I - FUNCTIONS

FUNCTION:

A function is a block of code (or sub program or self-contained program) that performs a particular task. Every C program contains one or more functions.

Syntax:

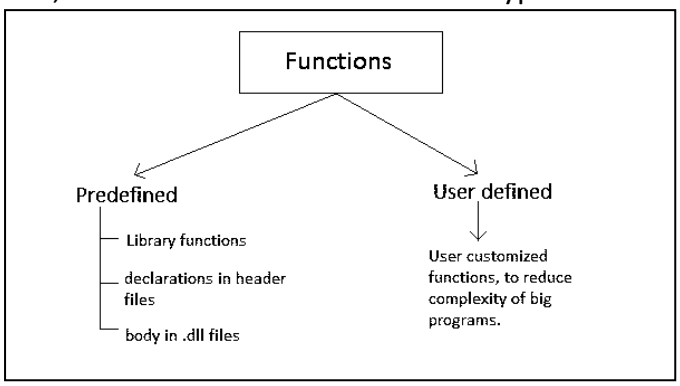
```
Return-type function-name (argument-list)
argument declaration;
{
 local variable declaration;
 statement;
 ….
 ….
 return statement;
}
```

EXAMPLE :

```
void sum( int,int);          //function declaration(prototype)
main ( )
{
sum(10,20);                  //call of function
}
void sum(int x, int y)       //function definition
{
printf (" sum=%d",x+y);
}
```

Types of Functions:

In C, functions are divided in to two types



1. Built-in (or) Library functions:

Library functions are functions defined and declared by C library in the header files.

Example: **printf()**, **scanf()**, **strcat()** etc.

2. User-defined functions: These functions are defined by the user at the time of writing a program.

USER-DEFINED FUNCTIONS (or) USING FUNCTIONS

The functions which are defined by user are called as user-defined functions. The user-defined function contains following elements:

1. Function Declaration / prototype
2. Function definition (or) called function
3. Function call (or) calling function
4. Arguments list
5. Return statement

1. Function Declaration / Function prototype:

Like variables, functions are declared before they are used in a C program. It can be declared either inside the main function or outside the main function with a statement terminator (;).

Syntax:

return-type function-name (parameter-list);

A function declaration contains following parts: **Function type (return type), Function name** and **Parameter list**

Example: *void display();*
int sum(int a, int b);

2. Function Definition (or) called function:

It contains coding of the function. It has two parts – **function header** and **function body**

Syntax:

```
return-type function-name(arguement-list)
{
  Local variable declarations;
  statement1;
  statement2;
  ……..
  Return statement;
}
```

function body

Example:

```
int add( int x, int y)
{
        int z;
        z = x+y;
        return (z);
}
```

3. Function Call (or) calling function:

A function is called by function name followed by arguements placed in parentheses and separated by commas.

Syntax:

function-name (data-type arg1, data-type arg2, ……., data-type argn);

Example: sum(a,b);
square(10);

4. Return Statement:

The return statement is used to return a value from a user-defined function to its calling point.

Syntax-1: return;

Syntax-2: return (value);

Examples: return;
return (25);
return (x+y);
return x+y;

5. Arguements:

There are two types of arguements. They are

a) **Actual Parameters:** The parameters that are included in function calling point are called "actual parameters". These are used to send vales to the called function.

b) **Formal Parameters:** The parameters that are included in function definition are called "formal parameters". These are used to receive values from the calling point.

Example:

```
#include<stdio.h>
#include<conio.h>
int sum(int a,int b);                // function prototype
void main()
{
int x,y,z;
clrscr();
printf("\n Enter x,y values:");
scanf("%d%d",&x,&y);
z=sum(x,y);                  //actual arguments
```

Function call or calling function

```
printf("sum:%d",z);
getch();
}
int sum(int a,int b)     // formal arguments
{
return(a+b);
}
```

Function definition or called function

PASSING PARAMETERS TO THE FUNCTION:

When a function is called, the calling function has to pass some values to the called function. In 'C' language there are two types of methods used to send parameters to a function. They are

1**. Call by value**: In the call by value, the values of variables are passed by the calling function to the called function. When the function is called by 'call by value' method, it doesn't affect content of the actual argument.

Example:

```
#include<stdio.h>
#include<conio.h>
void add( int n);
int main()
{
int num = 2;
printf("\n before calling the function = %d", num);
add(num);
printf("\n after calling the function = %d", num);
return 0;
}
void add(int n)
{
n = n + 10;
printf("\n The value of num = %d", n);
}
```

2. Call by reference: In the call by reference, instead of passing the value of variable, address or reference is passed by the calling function to the called function.

To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list.

Example:

```
#include<stdio.h>
#include<conio.h>
void add( int &n);
int main()
{
int num = 2;
printf("\n before calling the function = %d", num);
add(num);
printf("\n after calling the function = %d", num);
return 0;
}
void add( int &n)
{
n = n + 10;
printf("\n The value of num = %d", n);
}
```

SCOPE OF VARIABLES:

The scope of variable means accessibility and visibility of the variables at different points in a program (that is, in which part of the program a variable is actually available for use)

1. Local variable:- The variables which are defined with in a function is called local variables. These variables are lost once they are used in a function.

Example:

```
function()
{
int a,b;
function1( );
}
function2( )
{
int a=0; b=20;
}
```

2. Global variable:- The variables which are defined outside of the function is called global variables. Global variables are automatically initialized at the time of initialization.

Example:

```
#include<stdio.h>
void function(void);
void function1(void);
void function2(void);
int a, b=20;
void main()
{
printf("inside main a=%d,b=%d \n",a,b);
function( );
function1( );
}
function( )
{
prinf("inside function a=%d, b=%d",a,b);
}
function1( )
{
prinf("inside function a=%d, b=%d",a,b);
}
```

3. Static variables: static variables are declared by writing the key word **static**.

Syntax:- static data-type variable-name;

Ex: static int a;

Q. STORAGE CLASSES

A storage class represents the visibility, life time, initial value and location of a variable within a C Program.

C supports following types of storage classes–

1. Automatic variables (auto)
2. Extern variables (Extern)
3. Static variables (static)
4. Register variables (register)

Storage Classes	Storage Place	Default Value	Scope	Purpose
auto	RAM	Garbage Value	Local	It is a default storage class
extern	RAM	Zero	Global	It is a global variable
static	RAM	Zero	Local	It is a local variable which is capable of returning a value even when control is transferred to the function call
register	Register	Garbage Value	Local	It is a variable which is stored inside a Register

1**. Automatic Variables:**

Automatic variables are always declared within a function with the keyword '**auto**'. They are local to the function in which they are declared. A variable declared inside a function without storage class specification is, by default an automatic variable.

Ex:
```
void main()
{
int a; (or) auto int a;
printf("%d",a);
}
```

Ex2:
```
#include <stdio.h>
void main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a);
}
}
printf("%d ",a);
}
```

Output:
11 20 20 20 11

2. External Variables:

Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables. Unlike local variables, global variables can be accessed by any function in a program. External variables are declared outside of any function. These variables are stored in memory. Default initial value is zero.

Example:
```
#include<stdio.h>
#include<conio.h>
int fun1( );
int x;
main( )
{
x=10;
extern int z;
clrscr( );
printf("\n x=%d",x);
printf("\n x=%d",fun1( ));
getch();
}
z=25;
int fun1( )
{
x=x+10;
return x;
}
```

3. Static Variables:

A variable can be declared as static by using the keyword "static". These variables are stored in memory. Default initial value is zero. Local to the block in which it is declared.

Example:
```
#include<stdio.h>
#include<conio.h>
void stat();
void main()
{
int i;
clrscr();
for(i=1;i<=3;i++)
{
stat();
}
getch();
}
void stat()
{
static int x=0;
x=x+1;
printf("\n x=%d",x);
}
```

4.Register Variables:

These variables are stored in CPU registers. Default initial value is unpredictable. These variables are local to the block in which they are declared. They are local to the block in which they are declared.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
register int i;
clrscr();
for(i=1;i<=3;i++)
{
printf("\n %d",i);
}
getch();
}
```

RECURSIVE FUNCTIONS:

When function calls itself (inside function body) again and again then it is called as recursive function and this process is known as recursion. In recursion, calling function and called function are same.

Example:

```
int fcat(int);
main( )
{
int n;
clrscr( );
printf(" enter a no");
scanf(" %d",&n);
printf("fact=%d",fact(n));
getch( );
}
int fact( int x)
{
if((x==1))||(x==0))
return 1;
else
return x*fact(x-1)
}
```

Ex: find the factorial of a given number using recursive function

```
#include<stdio.h>
#include<conio.h>
int fact(int n);
void main()
{
int n,res;
clrscr();
printf("\n Enter a value:");
scanf("%d",&n);
res=fact(n);
printf("\n factorial of %d is %d",n,res);
getch();
}
int fact(int n)
{
if(n==0)
return 1;
else
return (n*fact(n-1));
}
```

UNIT - IV
Chapter – II – Structure, Union & Enumerated Data types

STRUCTURES:

A structure is a collection of data items of different data types referenced under a same name. A structure can be defined by using the keyword **'struct'**. Each individual item of a structure is called a member (or member variable).

Syntax:

```
struct structure-name
{
datatype member1;
datatype member2;
----------
----------
};
```

Example:

```
Struct student
{
 int sno;
 char sname[10];
 int s1,s2;
};
```

Defining structure variable:

A structure variable can be defined in the main() by using structure name.

Syntax: struct structure-name variable-list;

Ex: struct student s; (or)
struct student s1,s2,s3;

Initializing of structure:

Like variables, structure type variables also initialized while they are declaring in a program.

Syntax:

struct structure-name var1 = { values }, var2 = {values},……., var-n = {values};

Ex: 1) struct student s={a,"raju",77,88};
2) struct stud s1 = {1,"ABC", 45,55 }, s2 = {2,"XYZ",90,89};

Accessing Structure Elements:

The members of a structure are accessed through the use of dot (.) operator along with structure variable name and member name.

Syntax: structure_variable.member

Ex: printf("\n sno=%d",s.sno);
printf("\n sname=%s",s.sname);

Example:

```
#include<stdio.h>
#include<conio.h>
struct student
{
int sno,s1,s2,tot;
char sname[10];     float avg;
};
void main()
{
struct student s;
clrscr();
printf("\n Enter sno,sname,s1,s2:  \n");
scanf("%d%s%d%d",&s.sno,s.sname,&s.s1,
&s.s2);
s.tot=s.s1+s.s2;
s.avg=s.tot/2.0;
printf("\n total=%d",s.tot);
printf("\n average=%f",s.avg);
getch();
}
```

NESTED STRUCTURES:

When a structure is an element of another structure, it is called a nested structure.

Example:

```
#include<stdio.h>
#include<conio.h>
struct marks
{
int s1,s2;
};
struct student
{
int sno;
char sname[10];
struct marks m;
};
void main()
{
struct student s;
int tot;
printf("\n Enter sno & sname:\n");
scanf("%d%s",&s.sno,s.sname);
printf("\n Enter s1 marks:");
scanf("%d",&s.m.s1);
printf("\n Enter s2 marks:");
scanf("%d",&s.m.s2);
tot=s.m.s1+s.m.s2;
printf("\n total=%d",tot);
getch();
}
```

ARRAY OF STRUCTURES:

Like any other type, we can also declare an array of structures. To declare an array of structure, we must first define a structure and then declare an array of that type.

Syntax: ***structure structure_name array_name [size];***

Example:

```
struct student
{
int sno;
char sname[20];
float avg;
};
struct student s[10];
```

To access the element, we use a subscript or index. Like all array variables, arrays of structures also begin indexing at 0.

s[0].sno, s[0].sname, s[0].avg,s[1].sno,s[1].sname,s[1].avg……..s[9].avg

Example:

```
#include<stdio.h>
#include<conio.h>
struct emp
{
int eno;
char ename[10];
int sal;
};
main()
{
int i;
struct emp
e[3]={{1,"ramu",1000},{2,"sita",2000}};
for(i=0;i<2;i++)
{
printf("\n eno=%d",e[i].eno);
printf("\n ename=%s",e[i].ename);
printf("\n salary=%d",e[i].sal);
}
getch();
}
```

STRUCTURES AND FUNCTIONS:

Like any other variables, we can pass the members of a structure to a function as arguments. When we pass a member of a structure to the function, it is a call by value method.

Example:

```
#include<stdio.h>
#include<conio.h>
struct temp
{
int a;
}
;
int square(int x);
void main()
{
struct temp m;
printf("enter a value:");
scanf("%d",&m.a);
printf("square of a=%d",square(m.a));
}
int square(int x)
{
return(x*x);
}
```

UNIONS:

Union is also a collection of data items may be of different types or same data types referenced under a same name. The keyword **'union'** is used to define a union.

The members within the union all share the same storage area within the computer's memory. When the union is declared, the compiler automatically allocates memory location to the largest data type of members in the union.

Syntax:

```
union union_name
{
datatype member1;
datatype member 2;
--------------------
datatype member n;
};
union union_name variable-list;
```

Example:

```
union member
{
int a;
flaot b;
char c;
};
Union member m;
```

Accessing Union Members:

We can access the individual members of a union by using a dot operator.

Syntax: uninon_name.variable;

Example: m.a, m.b

Example:

```
#include<stdio.h>
union point
{
int x,y;
};
void main()
{
union point p;
p.x=3;
p.y=5;
clrscr();
printf("\n P values = %d and %d",p.x, p.y);
getch();
}
```

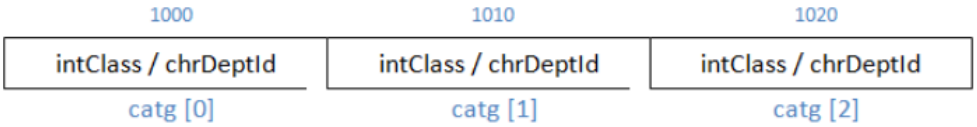
ARRAY OF UNION:

Like array of structures we can also create array of unions and access them in similar way. When array of unions are created it will create each element of the array as individual unions with all the features of union.

That means, each array element will be allocated a memory which is equivalent to the maximum size of the union member and any one of the union member will be accessed by array element.

Example:

```
union category
{
int intClass;
char chrDeptId[10];
};
union category catg[10]; // creates an array of unions with 10 elements of union type
```



Members of array of unions are accessed using dot (.) operator on union variable name along with the index to specify which array element that we are accessing.

```
catg[0].intClass = 10;
catg[5].chrDeptId = “DEPT_001”;
```

Here note that, each element of the union array need not access the same member of the union. It can have any member of the union as array element at any point in time.

In above example, first element of the union array accesses intClass while 6th element of union array has chrDeptId as its member.

ENUMERATED DATA TYPES:

The enum in C is also known as the enumerated type. It is a user-defined data type used to assign names to the integer constant values, and makes a program easy to read and maintain. The keyword “enum” is used to declare enumeration type.

Syntax: enum enum_name{const1,const2, ……., const n};

By default, ‘c’ compiler assigns integer values starting from zero to each identifier automatically. But we can change these values.

Example: enum grade {distinction, first, second, third};

enum fruits{mango, apple, strawberry, papaya};

Declaring Enumeration variables:

Once enumeration type has been defined, one or more variables of that type can be declared. Every variable of enumeration type occupies only two bytes of memory.

Syntax: enum enum_name var1, var2, …, var-n;

Example: enum grade g;

Example:

```
#include<stdio.h>
#include<conio.h>
enum grade{distinction,first,second,third};
main()
{
enum grade g;
printf("%d\n",distinction);
printf("%d\n",third);
getch();
}
```

UNIT - V
Chapter – I – POINTERS

POINTERS

A pointer is a variable which stores the memory address of another variable. Pointers are used to store the address of variables.

Declaring a pointer:

Like other variables, the pointer variable must be declared before using in the program

Syntax: datatype *ptr_var;

Ex:
```
int *ptr;
char *c;
```

Here 'ptr' is a pointer variable that contains address of a variable which is of integer type.

Initializing a pointer:

We can assign the address of a variable to a pointer variable by using address (&) operator.

Syntax: pointer_variable= &variable;

Ex:
```
int a,*p;
a=10;
p=&a;
```

Accessing Pointer variable:

We can access the value of a pointer variable by using the indirection (*) operator.

Syntax: *pointer_name;

Advantages and Dis-Advantages:

Advantages:

- Pointers provide direct access to memory
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers are used to pass information back and forth between the calling function and called function.
- A pointer allows us to perform dynamic memory allocation and de-allocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- A pointer allows us to resize the dynamically allocated memory block.

Drawbacks of pointers in c:

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

NULL POINTER:

A Null Pointer is a pointer that does not point to any memory location. **Following are the applications of a Null pointer:**

- It is used to initialize o pointer variable when the pointer does not point to a valid memory address.
- It is used to perform error handling with pointers before dereferencing the pointers.
- It is passed as a function argument and to return from a function when we do not want to pass the actual memory address

Examples of Null Pointer:

```
int *ptr=(int *)0;
float *ptr=(float *)0;
char *ptr=(char *)0;
double *ptr=(double *)0;
char *ptr='\0';
int *ptr=NULL;
```

POINTER EXPRESSIONS AND POINTER ARITHMETIC

Like other variables, pointer variable can be used in expression. An expression is a collection of operands joined together by certain operator that represent a value.

Ex:

//arithmetic operations using pointers

```
#include <stdio.h>
#include <conio.h>
main()
{
int n1,n2,sum=0,mul=0,*p1,*p2;
float div;
p1=&n1;
p2=&n2;
clrscr();
printf("\n Enter n1,n2 values:");
scanf("%d%d",&n1,&n2);
sum=*p1+*p2;
mul=*p1**p2;
div=(*p1)/(*p2);
printf("\n sum= %d",sum);
printf("\n mul= %d",mul);
printf("\n div= %f",div);
getch();
}
```

Output:

```
Enter n1, n2 values: 25
4
sum= 29
mul= 100
div=6.000000
```

UNIT - V
Chapter – II – FILES

FILES:

A file is a collection of data or information stored in secondary storage devices such as hard disk, floppy disk devices.

Normally, a program input and output values through the variables are created in primary memory. The values given by a user are stored temporarily. Once the program execution completes, the values are automatically removed.

A file is created in the secondary memory device such as disk or tape. After creating a file, we can modify or delete data present in the file as and when needed.

To manipulate files in 'C' language, we must use FILE data type. It is structure data type that is used to create file buffer area.

Syntax: FILE *fp;

Here, FILE is the name of data type and 'fp' is file pointer which will contain all the information about the file.

USING FILES IN C:

To use files in C, we must follow the below steps:

1. Declare a file pointer variable
2. Open the file
3. Process the file
4. Close the file

1. Declare a file pointer variable: We can declare a file pointer variable as follows

Syntax: FILE *file_pointer_name;

Example: FILE *fp

2. Opening the file:

A file should be opened before it is used in the program. The fopen() function is used to open a file. If the file open operation is success, this function returns a FILE pointer otherwise it returns NULL.

Syntax: fp=fopen("file_name","mode")

Ex: fp=fopen("first","r");

Mode	Meaning
"r"	Opens text file for reading
"w"	Opens text file for writing
"a"	Opens text file in append mode

3. Closing a File:

A file must be closed when no input/output is to be performed on it. The function fclose() is used to close the file.

Syntax: fclose (file_pointer)

Example: fclose (fp)

Q. Explain I/O operations on FILES

(or)

Explain read data from files & writing data to files?

The following functions are used to manipulate data in the files

Input functions (or) Read data from FILES

1. **fgetc():** It is used to read a single character from the file and stores it in a variable.

Syntax: variable = getc(fp);

2. fgets(): It is used to read the specified number of characters from the file and stores it in a variable.

Syntax: fgets(str, n, fp);

Ex: fgets(nm, 10, fp);

3. fscanf(): It is used to read data items from the file and stores them into variables. It read values based on formatting characters specified. This function returns an integer if it is success otherwise it returns zero.

Syntax: fscanf(fp, format-string, variable-list);

Ex: fscanf(fp, "%d %s %f" , &rno, snm, &avg);

Output functions (or) Write data to FILES

1. **fputc() functions:** It is used to store a single character into file.

Syntax: fputc(ch, fp);

2. **fputs() function:** It is used to store a string into the file.

Syntax: fputs(string, fp);

Ex: fputs(nm, fp);

3. **fprintf():** It is used to store data items into the file. It writes values based on formatting characters specified. The general format is as follows:

Syntax: fprintf(fp, format-string,variable-list);

COMMAND-LINE ARGUMENTS:

C allows the programmers to pass command line arguments. Command line arguments are given after the name of a program in command line operating system like DOS or LINUX.

The main () function can accept two arguments.

1. The first argument is an integer value that specifies number of command line arguments. (argc)
2. The second argument is a full list of all of the command line argument(argv)

Syntax: main(int argc, char *argv[])

Example:

```
#include<stdio.h>
main(int argc, char *argv[])
{
int i;
printf("number of arguments %d\n", argc);
for(i=0;i<argc; i++)
printf("argv[%d]=%s\n" ,i,argv[i]);
}
```

Error handling during FILE operations:

An error may occur while reading data from or writing data to a file. An error may rise

- When trying to read a file beyond EOF indicator.
- When trying to read a file that does not exist
- When trying to use a file that has not been opened.
- When trying to use a file in an inappropriate mode, i.e., writing data to a file that has been opened for reading.
- When writing to a file that is write – protected.

1. **ferror():**
   It is used to check for errors in the stream. It returns value 0 if no errors have occurred and a non-zero value if there is an error
   Syntax: int ferror(FILE *stream)
2. **clearerr ():**
   It is used to clear the end of file and error indicators for the stream. The function is used because error indicators are not automatically cleared
   Syntax: void clearer(FILE *stream);
3. **perror():**
   The function perror() stands for print error.
   Syntax: void perror(char *msg);

INDEX

1. Write a C program to check whether a number is Armstrong or not.

Aim: To write a C-program to check whether a number is Armstrong or not.

Definition:

An Armstrong number of three digits is an integer such that the sum of the cubes of its digits is equal to the number itself.

For example, 371 is an Armstrong number since

$3^3 + 7^3 + 1^3 = 371$.

Algorithm:

Step 1: Start

Step 2: intialize n,m,rem,sum←0

Step 3: write "Enter a positive integer:"

Step 4: Read n

Step 5: Assign m←n

Step 6: Repeat while n>0

a. rem← n%10

b. sum←sum+ rem*rem*rem

c. n←n/10

Step 7: If sum==temp then

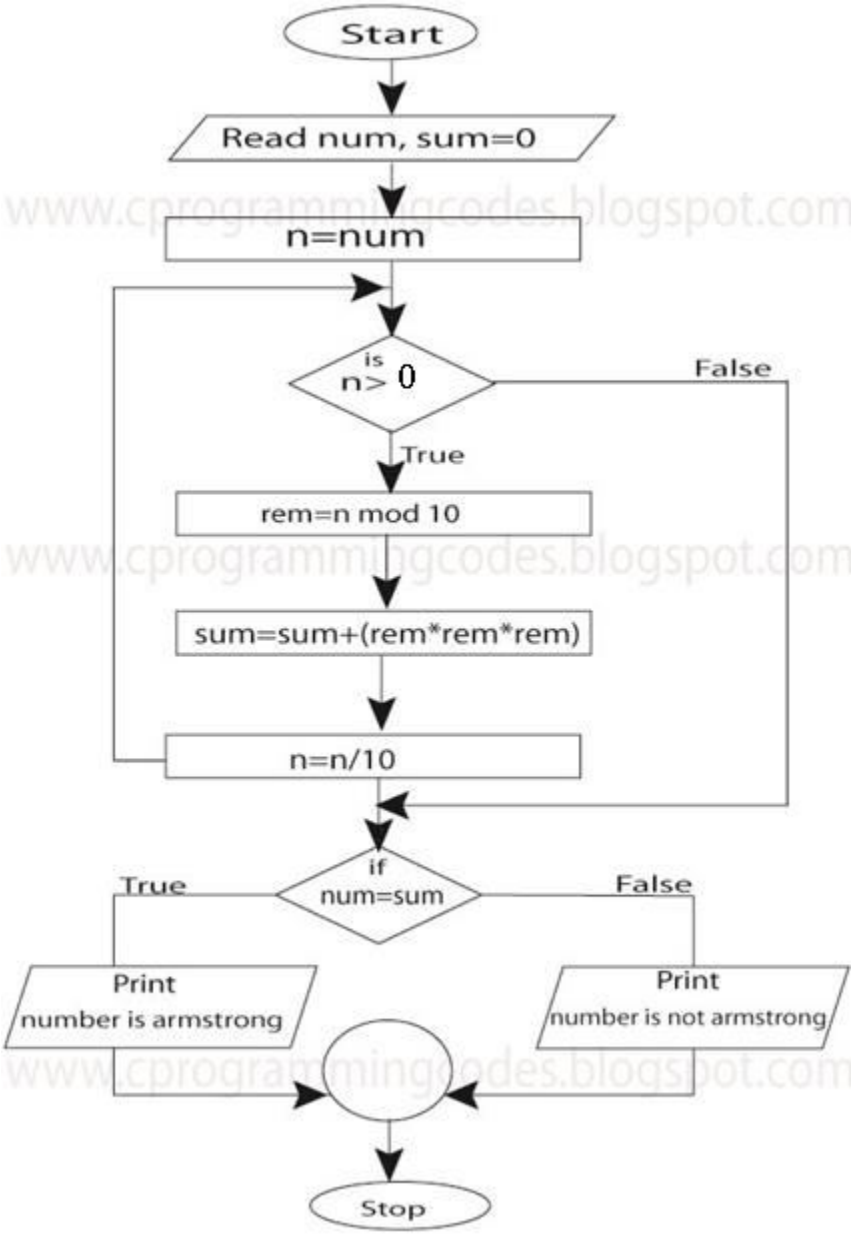
a. Write “given number is an Armstrong number”

Step 8: else

a. Write “given number is not an Armstrong number”

Step 9: Stop

Flow chart



www.cprogrammingcodes.blogspot.com

Source Code:

```
/* Armstrong.c */
#include <stdio.h>
#include<conio.h>
int main()
{
  int n,m,rem,sum=0;
  clrscr( );
  printf("Enter a positive  integer: ");
  scanf("%d", &n);
  m=n;
  while(n>0)
  {
     rem=n%10;
     sum=sum+rem*rem*rem;
     n=n/10;
  }
  if(sum==m)
    printf("%d is an Armstrong number.",m);
  else
    printf("%d is not an Armstrong number.",m);
  getch( );
}
```

Output1:

Enter a positive integer: 153
153 is an Armstrong number.

Output2:

Enter a positive integer: 171
171 is not an Armstrong number.

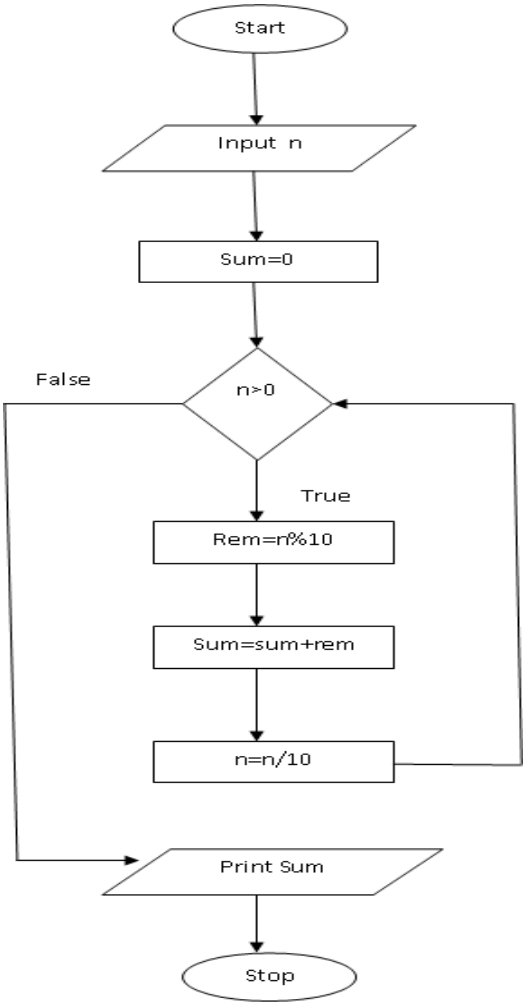
2. Write a C program to find the sum of individual digits of a positive integer.

Program Statement:
Write a C program to find the sum of individual digits of a positive integer.

Aim: To write a C-program to find the sum of individual digits of a positive integer.

Algorithm:
Step 1: Start
Step 2: initialize n,res,sum←0
Step 3: write "\n Enter a Positive Integer:"
Step 4: Read n
Step 5: Repeat while n>0
a. rem← n%10
b. sum← sum+ rem
c. n← n/10
Step 6: write "\nThe sum of digits is:",sum
Step 6: Stop

Flow chart:


Start
Input n
Sum=0
n>0
False
True
Rem=n%10
Sum=sum+rem
n=n/10
Print Sum
Stop

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n,rem,sum=0;
        clrscr( );
        printf("\nEnter a +ve integer:");
        scanf("%d",&n);
        while(n>0)
        {
                rem=n%10;
                sum=sum+rem;
                n=n/10;
        }
        printf("\nThe sum of digits is:%d",sum);
        getch( );
}
```

Output:

Enter a positive integer: 1234
The sum of digits is: 10

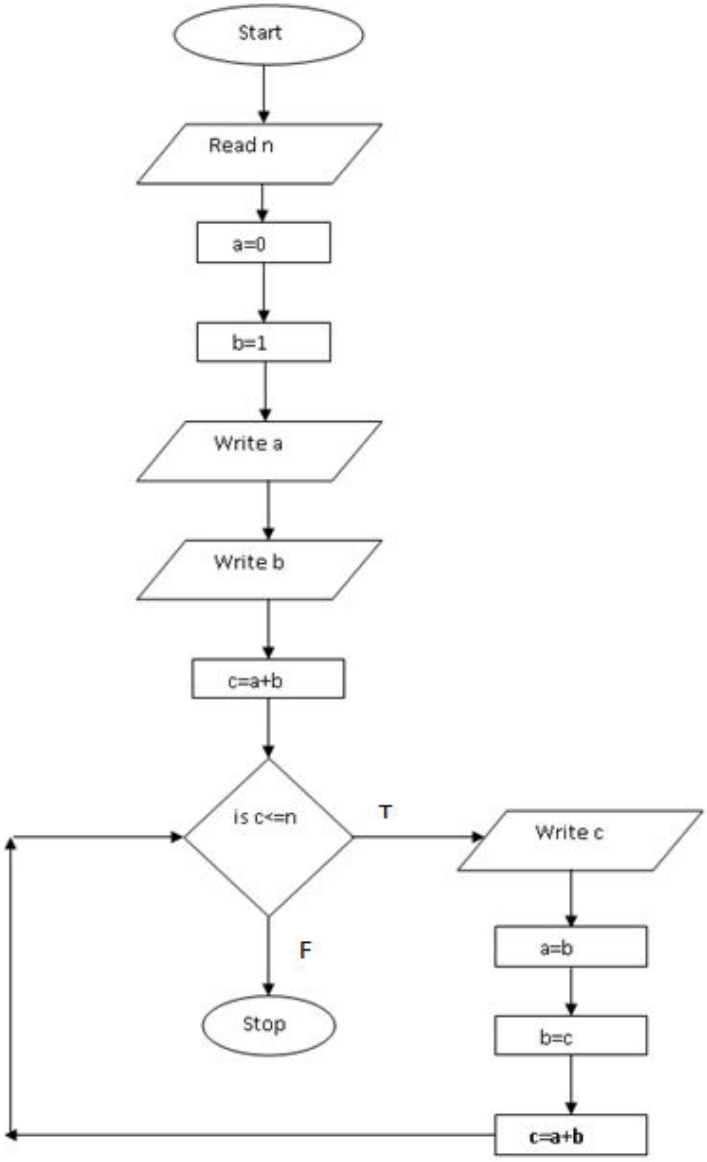
3. Write a program to generate the first n terms of the Fibonacci sequence
Program Statement:
Write a program to generate the first n terms of the Fibonacci sequence

Aim: To Write a program to generate the first n terms of the Fibonacci sequence

Algorithm:
Step 1: Start.
Step 2: initialize a,b,c,n
Step 3: Write "\n Enter n value:"
Step 4: Read n
Step 3: Assign a←0
Step 4: Assign b←1
Step 5: Write a.
Step 6: Write b.
Step 7: Write "\nThe Fibonacci series is:\n"
Step 8: write a
Step 9: write b
Step 10: c← a+b
Step 11: Repeat steps (i) to (iv) until c <= n.
- **a.** Write c
- **b.** Assign a←b
- **c.** Assign b←c.
- **d.** c←a +b

Step 9: End.

Flow chart:


Start
Read n
a=0
b=1
Write a
Write b
c=a+b
is c<=n
T
Write c
a=b
b=c
c=a+b
F
Stop

Source Code:

```
/* Fibonacci.c */
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,c,n;
        clrscr( );
        printf("Enter n value \n");
        scanf("%d",&n);
        a=0;
        b=1;
        printf("\nThe Fibonacci series is….\n");
        printf("%d",a);
        printf("%3d",b);
        c=a+b;
        while(c<=n)
        {
                printf("%3d",c);
                a=b;
                b=c;
                c=a+b;
        }
        getch( );
}
```

Output:

```
Enter n value:      10
The Fibonacci series is:
 0      1        1        2        3        5        8
```

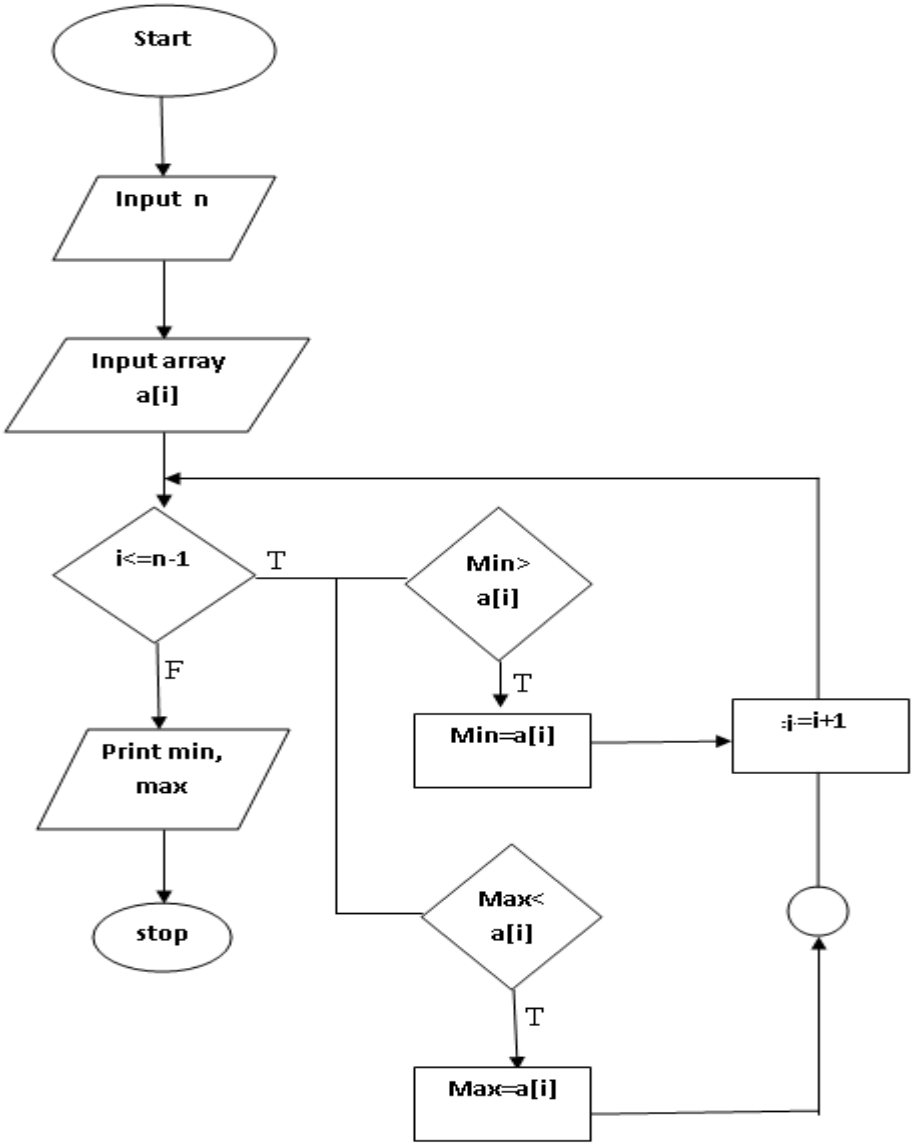
4. Write a C program to find both the largest and smallest number in a list of integer values

Program Statement:
Write a C program to find both the largest and smallest number in a list of integer values

Aim:
To write a C-program to find both the largest and smallest number in a list of integers.

Algorithm:
Step 1: Start
Step 2: initialize a[10],n,i,min,max
Step 3: Write “Enter array size:”
Step 4: Read n
Step 5: Write "\n Enter elements into the array:"
Step 6: Repeat the following steps until i<=n-1 starts from i=1
    a. Read a[i]
Step 7: assign max = min = a[0]
Step 8: Repeat the following steps until i<=n-1 starts from i←1
    a. If min>a[i] then
            assign min←a[i].
    b. If max<a[i]
            assign max←a[i].
Step 9: write “\n Max Value:”,max
Step 10: write “\n Min Value:”. Min
Step 11: Stop

Flow chart:


Start
Input n
Input array
a[i]
i<=n-1
T
F
Min>
a[i]
T
Min=a[i]
i=i+1
Print min,
max
stop
Max<
a[i]
T
Max=a[i]

Source Code:

```
/* largesmall.c */
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10],n,i,min,max;
        clrscr( );
        printf("\nEnter array size:");
        scanf("%d",&n);
        printf("\nEnter elements into the array:");
        for(i=0;i<=n-1;i++)
        {
                scanf("%d",&a[i]);
        }
        min=max=a[0];
        for(i=1;i<=n-1;i++)
        {
                if(min>a[i])
                {
                        min=a[i];
                }
                if(max<a[i])
                {
                        max=a[i];
                }
        }
        printf("\nMax Value:%d",max);
        printf("\nMin Value:%d",min);
        getch( );
}
```

Output:

```
Enter array size: 5
Enter elements into the array: 10    20   30   40   50
Max value: 50
Min value: 10
```

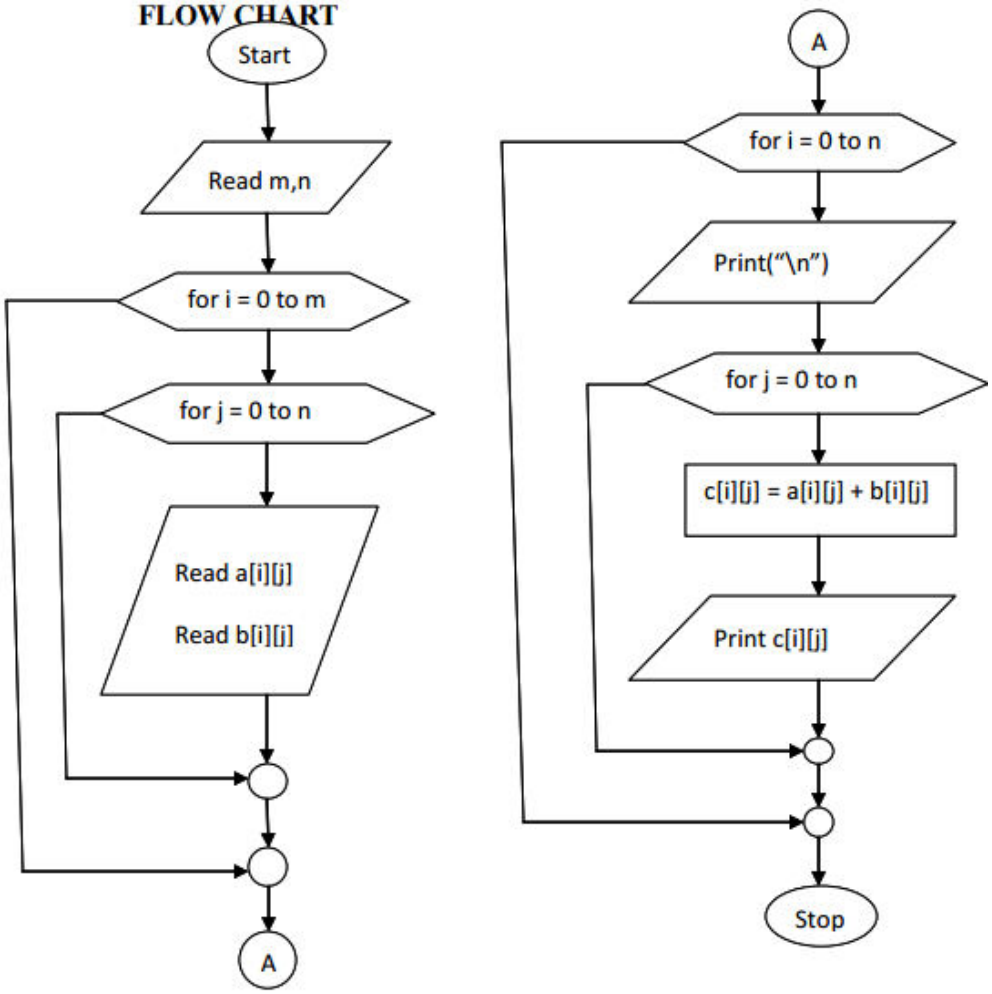
5. Write a program that uses functions to add two matrices
Program Statement:
Write a program that uses functions to add two matrices

Aim:
To Write a program that uses functions to add two matrices

Algorithm:
Step 1: Start
Step 2: Initialize a[10][10],b[10][10],c[10][10],r1,c1,r2,c2,i,j;
Step 3: write "\nEnter order of first matrix:"
Step 4: read r1,c1
Step 5: write "\nEnter order of second matrix:"
Step 6: read r2,c2
Step 7: if(c1==c2&&r1==r2) then
Write "\n Enter elements into first matrix\n");
Repeat for(i=0;i<=r1-1;i++)
Repeat for(j=0;j<=c1-1;j++)
Read a[i][j]

Write "\nEnter elements into second matrix\n"
Repeat for(i=0;i<=r2-1;i++)
Repeat for(j=0;j<=c2-1;j++)
Read b[i][j]
Step 8: Repeat for(i=0;i<=r1-1;i++)
Repeat for(j=0;j<=c1-1;j++)
Read c[i][j] ← a[i][j]+b[i][j]
Step 9: write "\nAddition of matrices is....\n\n");
Step 10: Repeat for(i=0;i<=r1-1;i++)
Repeat for(j=0;j<=c1-1;j++)
Write c[i][j]
Step 11: else
Write "\n Matrix addition not possible"
Step 12: Stop

FLOW CHART


Start
Read m,n
for i = 0 to m
for j = 0 to n
Read a[i][j]
Read b[i][j]
A
A
for i = 0 to n
Print("\n")
for j = 0 to n
c[i][j] = a[i][j] + b[i][j]
Print c[i][j]
Stop

Source Code:

```
/*MatAdd.c*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10][10],b[10][10],c[10][10];
        int r1,c1,r2,c2,i,j;
        clrscr( );
        printf("\n Enter order of first matrix:");
        scanf("%d%d",&r1,&c1);
        printf("\n Enter order of second matrix:");
        scanf("%d%d",&r2,&c2);

        if(c1==c2&&r1==r2)
        {
                printf("\n Enter elements into first matrix: \n");
                for(i=0;i<=r1-1;i++)
                {
                        for(j=0;j<=c1-1;j++)
                        {
                                scanf("%d",&a[i][j]);
                        }
                }
                printf("\n Enter elements into second matrix: \n");
                for(i=0;i<=r2-1;i++)
                {
                        for(j=0;j<=c2-1;j++)
                        {
                                scanf("%d",&b[i][j]);
                        }
                }

/*Logic for matrix addition*/
                for(i=0;i<=r1-1;i++)
                {
                        for(j=0;j<=c1-1;j++)
                        {
                                c[i][j]=a[i][j]+b[i][j];
                        }
                }
                printf("\n Addition of two matrices: \n\n");
                for(i=0;i<=r1-1;i++)
                {
                        for(j=0;j<=c1-1;j++)
                        {
                                printf("%d \t",c[i][j]);
                        }
                        printf("\n\n");
                }
```

```
        }
        else
        {
                printf("\n Matrix addition is not possible");
        }
        getch( );
}
```

Output 1:

Enter order of first matrix: 2 2

Enter order of first matrix: 2 2

Enter elements into first matrix:
1 2
3 4

Enter elements into second matrix:
5 6
7 8

Addition of two matrices:
6 8
10 12

Output 2:

Enter order of first matrix: 2 3

Enter order of first matrix: 3 2

Matrix addition is not possible

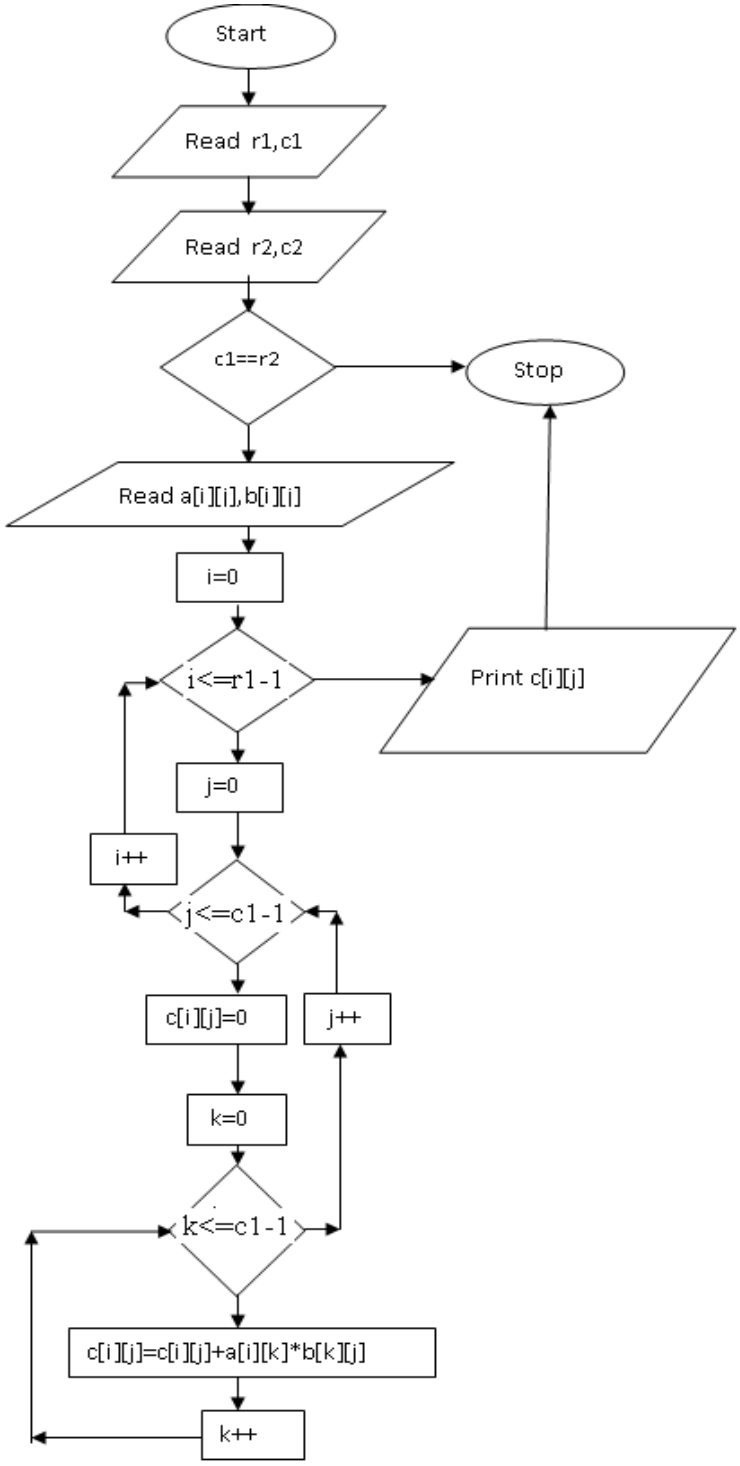
6. Write a program for multiplication of two N X N matrices
Program Statement:
Write a program for multiplication of two N X N matrices

Aim:
To Write a program for multiplication of two N X N matrices

Algorithm:
Step 1: Start.
Step 2: initialize a[10][10],b[10][10],c[10][10],r1,c1,r2,c2,i,j,k
Step 3: write "\n Enter order of first matrix:"
Step 4: read r1,c1
Step 5: write "\n Enter order of second matrix:"
Step 6: read r2,c2
Step 4: If c1 ==r2 then proceed else go to step 11
Step 5: write "\n Enter elements into first matrix: \n"
Step 6: Repeat for(i=0;i<=r1-1;i++)
Repeat for(j=0;j<=c1-1;j++)
Read a[i][j]
Write "\n Enter elements into second matrix: \n"
Repeat for(i=0;i<=r2-1;i++)
Repeat for(j=0;j<=c2-1;j++)
Read b[i][j]

Step 7: Repeat for(i=0;i<=r1-1;i++)
Repeat for(j=0;j<=c2-1;j++)
Read c[i][j]=0
Step 8: Repeat for(i=0;i<=r1-1;i++)
Repeat for(j=0;j<=c2-1;j++)
Repeat for(k=0;k<=c1-1;k++)
c[i][j] ← c[i][j] + a[i][k] * b[k][j]
Step 9: write "\n Multiplication of two matrices: \n\n"
Step 10: Repeat for(i=0;i<=r1-1;i++)
Repeat for(j=0;j<=c2-1;j++)
Write c[i][j]
Step 11: else
Write "\n Matrix multiplication is not posible"
Step 12: Stop

Flow chart


Start
Read r1,c1
Read r2,c2
c1==r2
Stop
Read a[i][j],b[i][i]
i=0
i<=r1-1
Print c[i][j]
j=0
i++
j<=c1-1
c[i][j]=0
j++
k=0
k<=c1-1
c[i][j]=c[i][j]+a[i][k]*b[k][j]
k++

Source Code:

```
/*MatMul.c*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[10][10],b[10][10],c[10][10];
        int r1,c1,r2,c2,i,j,k;
        clrscr( );
        printf("\n Enter order of first matrix:");
        scanf("%d%d",&r1,&c1);
        printf("\n Enter order of second matrix:");
        scanf("%d%d",&r2,&c2);

        if(c1==r2)
        {
                printf("\n Enter elements into first matrix: \n");
                for(i=0;i<=r1-1;i++)
                {
                        for(j=0;j<=c1-1;j++)
                        {
                                scanf("%d",&a[i][j]);
                        }
                }
                printf("\n Enter elements into second matrix: \n");
                for(i=0;i<=r2-1;i++)
                {
                        for(j=0;j<=c2-1;j++)
                        {
                                scanf("%d",&b[i][j]);
                        }
                }

/*Logic for matrix multiplication*/
                for(i=0;i<=r1-1;i++)
                {
                        for(j=0;j<=c2-1;j++)
                        {
                                c[i][j]=0;
                        }
                }
                for(i=0;i<=r1-1;i++)
                {
                        for(j=0;j<=c2-1;j++)
                        {
                                for(k=0;k<=c1-1;k++)
                                {
                                        c[i][j]=c[i][j]+a[i][k]*b[k][j];
                                }
                        }
```

```
                }
                printf("\n Multiplication of two matrices: \n\n");
                for(i=0;i<=r1-1;i++)
                {
                        for(j=0;j<=c2-1;j++)
                        {
                                printf("%d \t",c[i][j]);
                        }
                        printf("\n\n");
                }
        }
        else
        {
                printf("\n Matrix multiplication is not posible");
        }
        getch( );
}
```

Output 1:

Enter order of first matrix: 2 2

Enter order of first matrix: 2 2

Enter elements into first matrix:
1 2
3 4

Enter elements into second matrix:
1 2
3 4

Addition of two matrices:
7 10
15 22

Output 2:

Enter order of first matrix: 2 3

Enter order of first matrix: 3 2

Matrix multiplication is not possible

7. Write a program to search an element in a given list of values

Program Statement:
Write a program to search an element in a given list of values

Aim:
To write a program to search an element in a given list of values

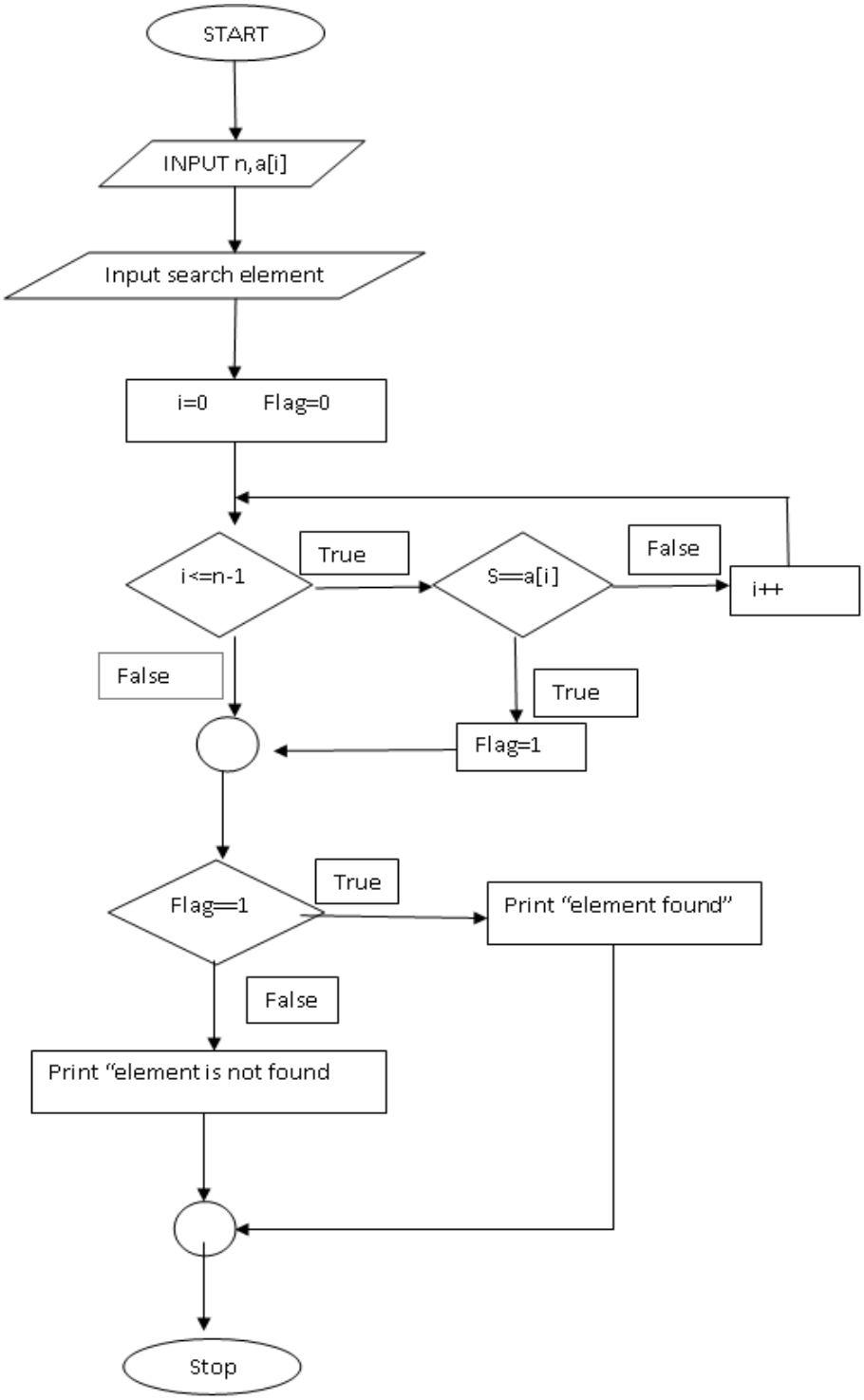
Description:
The linear search is most simple searching method. It does not expect the list to be sorted. The key which is to be searched is compared with each element of the list one by one. If a match exists, the search is terminated. If the end of list is reached it means that the search has failed and key has no matching in the list.

Algorithm:
Step 1: Start the program.
Step 2: Input n as size of the array and initialize flag=0
Step 3: Input array a[i] and search element s
Step 4: Repeat the following steps until i<=n-1 starts from i=0

a. Check whether a[i]==s or not, if it is true do the following otherwise increase i value by 1.
    i. flag=1
    ii. goto step 5

Step 5: Check whether flag==1 or not, if it is true print "Search element is found" otherwise print "Search element is not found"
Step 6: Stop

Flow chart


START
INPUT n, a[i]
Input search element
i=0 Flag=0
i<=n-1
True
S==a[i]
False
i++
False
True
Flag=1
Flag==1
True
Print "element found"
False
Print "element is not found
Stop

Source Code:

```
/* linearsearch.c*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10],n,s,i,flag=0;
        clrscr( );
        printf("\n Enter array size:");
        scanf("%d",&n);
        printf("\n Enter elements into the array: \n");
        for(i=0;i<=n-1;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("\n Enter search element:");
        scanf("%d",&s);

/* Logic for Searching */
        for(i=0;i<=n-1;i++)
        {
                if(s==a[i])
                {
                        flag=1;
                        break;
                }
        }
        if(flag==1)
        {
                printf("\nThe search element %d is found at %d position",s,(i+1));
        }
        else
        {
                printf("\nThe search element %d is not found",s);
        }
        getch( );
}
```

Output 1:

```
Enter array size: 5
Enter elements into the array:
10      20      30      40      50
Enter search element: 40
The search element 40 is found at 4 position
```

Output 2:

```
Enter array size: 5
Enter elements into the array:
10      20      30      40      50
Enter search element: 60
The search element 60 is not found
```

8. Write a program to sort a given list of integers in ascending order.

Program Statement:

Write a program to sort a given list of integers in ascending order.

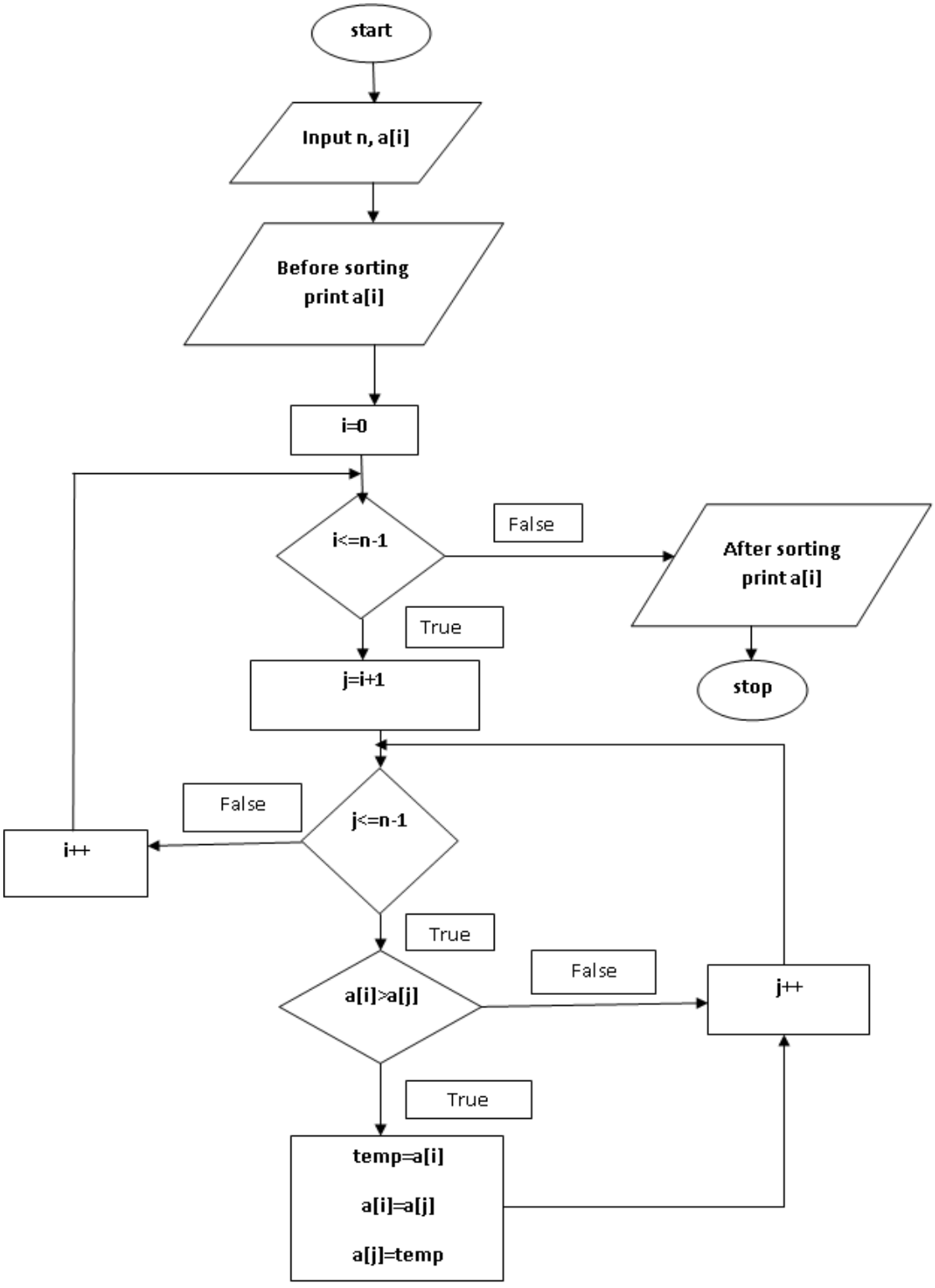
Aim:

To write a program to sort a given list of integers in ascending order

Algorithm:

Step 1: Start.
Step 2: Read the value of n.
Step 3: Input array a[i] from the keyboard
Step 4: Before sorting print a[i]
Step 5: Initializing i to 0.
Step 6: Repeat steps a to c until i < =n-1 start from i=0.
  Step a: Initialize j to i+1.
  Step b: Repeat steps i & ii until j <= n-1.
    Step i: If a[i] > a[j] then,
      temp=a[i]
      a[i]=a[j].
      a[j]=temp.
    Step ii: Increment j by 1.
  Step c: Increment i by 1.
Step 7: After sorting print the array a[i]
Step 8: Stop

Flowchart:


start
Input n, a[i]
Before sorting
print a[i]
i=0
i<=n-1
False
After sorting
print a[i]
True
stop
j=i+1
False
j<=n-1
i++
True
a[i]>a[j]
False
j++
True
temp=a[i]
a[i]=a[j]
a[j]=temp

Source Code:

```
/* arraysort.c*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10],n,i,j,temp;
        clrscr( );
        printf("\n Enter the array size:");
        scanf("%d",&n);
        printf("\n Enter the elements into the array:");
        for(i=0;i<=n-1;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("\n Before sorting the array elements are: \n\n");
        for(i=0;i<=n-1;i++)
        {
                printf("%3d",a[i]);
        }

/* Logic for sorting */
        for(i=0;i<=n-1;i++)
        {
                for(j=i+1;j<=n-1;j++)
                {
                        if(a[i]>a[j])
                        {
                                temp=a[i];
                                a[i]=a[j];
                                a[j]=temp;
                        }
                }
        }
        printf("\n After sorting the array elements are: \n\n");
        for(i=0;i<=n-1;i++)
        {
                printf("%3d",a[i]);
        }
        getch( );
}
```

Output:

```
Enter the array size: 5
Enter the elements into the array:
56      36      12      53      45
Before sorting the array elements are:
56      36      12      53      45
After sorting the array elements are:
12      36      45      53      56
```