

B.E Ist Year PPS Syllabus

Course Objectives

- ☐ To introduce the basic concepts of Computing environment, number systems and flowcharts
- ☐ To familiarize the basic constructs of C language – data types, operators and expressions
- ☐ To understand modular and structured programming constructs in C
- ☐ To learn the usage of structured data types and memory management using pointers
- ☐ To learn the concepts of data handling using pointers

Course Outcomes

The students will be able to

1. Formulate simple algorithms for arithmetic and logical problems.
2. Translate the algorithms to programs (in C language).
3. Test and execute the programs and correct syntax and logical errors.
4. Implement conditional branching, iteration and recursion.
5. Decompose a problem into functions and synthesize a complete program using divide and conquer approach.
6. Use arrays, pointers and structures to formulate algorithms and programs.
7. Apply programming to solve matrix addition and multiplication problems and searching and sorting problems.
8. Apply programming to solve simple numerical method problems, namely root finding of function, differentiation of function and simple integration.

Unit - I

Introduction to Programming: Introduction to components of a computer system (disks, memory, processor, where a program is stored and executed, operating system, compilers etc.). Idea of Algorithm: steps to solve logical and numerical problems.

Representation of Algorithm: Flowchart / Pseudocode with examples. From algorithms to programs; source code, variables (with data types) variables and memory locations, Syntax and Logical Errors in compilation, object and executable code.

Unit - II

Control Structures: Arithmetic expressions and precedence, Conditional Branching and Loops, Writing and evaluation of conditionals and consequent branching. Arrays: Arrays (1-D, 2-D), Character arrays and Strings

Unit - III

Basic Algorithms: Searching, Basic Sorting Algorithms (Bubble and Selection), Finding roots of Equations.

Functions: Functions (including using built-in libraries), Parameter passing in functions, call by value. Passing arrays to functions: idea of call by reference.

Unit - IV

Recursion: Recursion, as a different way of solving problems. Example programs, such as Finding Factorial, Fibonacci series. Structure: Structures, Defining structures and Array of Structures

Unit - V

Pointers - Idea of pointers, Defining pointers, Use of Pointers in self-referential structures, notion of linked list (no implementation), Introduction to File Handling.

Suggested Readings Books:

1. Byron Gottfried, Schism's Outline of Programming with C, McGraw-Hill
2. A.K. Sharma, Computer Fundamentals and Programming in C, Universities Press, 2nd Edition, 2018.
3. E. Balaguruswamy, Programming in ANSI C, Tata McGraw-Hill
4. Brian W. Kernighan and Dennis M. Ritchie, the C Programming Language, Prentice Hall of India.

UNIT I

Introduction to Programming: Introduction to components of a computer system (disks, memory, processor, where a program is stored and executed, operating system, compilers etc.). **Idea of Algorithm:** steps to solve logical and numerical problems. **Representation of Algorithm:** Flowchart / Pseudocode with examples. From algorithms to programs; source code, variables (with data types) variables and memory locations, Syntax and Logical Errors in compilation, object and executable code.

I. Introduction to Programming:

A programming language is a set of commands, instructions, and other [syntax](#) use to create a software [program](#). Languages that programmers use to write code are called "high-level languages." This code can be compiled into a "low-level language," which is recognized directly by the computer hardware.

High-level languages are designed to be easy to read and understand. This allows programmers to write [source code](#) in a natural fashion, using logical words and symbols. For example, reserved words like function, while, if, and else are used in most major programming languages. Symbols like <, >, ==, and != are common operators. Many high-level languages are similar enough that programmers can easily understand source code written in multiple languages.

1. INTRODUCTION TO COMPONENTS OF A COMPUTER SYSTEM:

Computer components are all the parts that make up a computer. Components include software and hardware: The software tells the hardware what to do and the hardware executes the commands. Working together, these components make up a system that can relay commands to a central processing unit, identify the actions needed to carry out the commands, and send the instructions to the component that carries out the command. Computer components work in close conjunction with each other and when one component fails, many others are affected and sometime the entire system can crash.

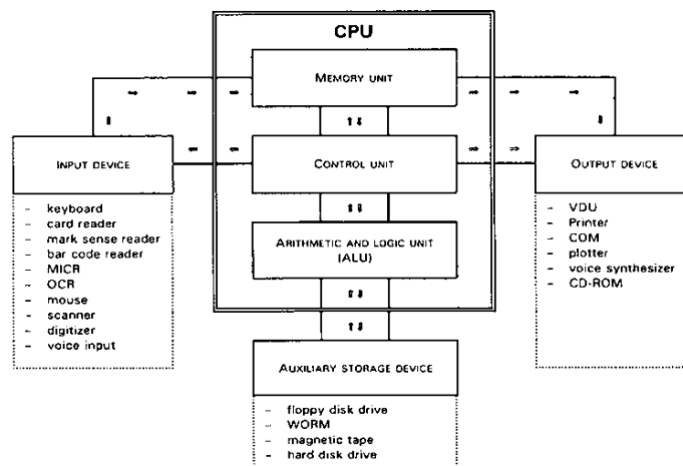


Fig 1.1: Components of Computer System

There are two basic types of computer components: software and hardware.

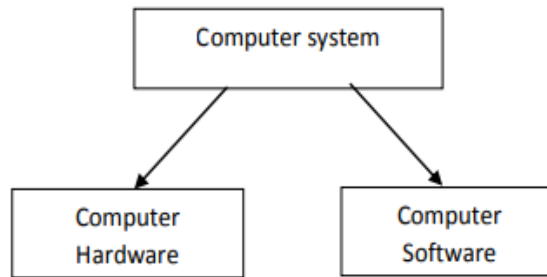


Fig: Computer System

1.1 COMPUTER HARDWARE:-Computer hardware is a physical component of a computer which a user can feel, touch, see and hear. The hardware of a computer system is made up of a number of electronic devices connected together to computer system.

1.1.2 COMPUTER SOFTWARE:- Software is a generic term for organized collections of computer data and instructions, often broken into two major categories They are:

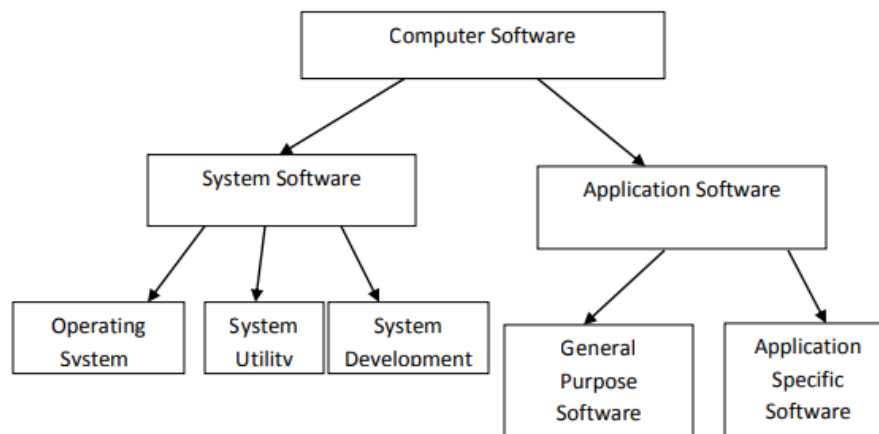


Fig : Computer Software

1. DISKS:

Generally, a disk is a a round plate on which data can be encoded. There are two basic types of disks: *magnetic disks* and *optical disks*.

Magnetic Disks

On magnetic disks, data is encoded as microscopic magnetized *needles* on the disk's surface. You can record and erase data on a magnetic disk any number of times, just as you can with a cassette tape. Magnetic disks come in a number of different forms:

- **Floppy disk :** A typical 5¼-inch floppy disk can hold 360K or 1.2MB (megabytes). 3½-inch floppies normally store 720K, 1.2MB or 1.44MB of data. Floppy disks are obsolete today, and are found on older computer systems.
- **Hard disk :**Hard disks can store anywhere from 20MB to more than 1-TB (terabyte). Hard disks are also from 10 to 100 times faster than floppy disks.
- **Removable cartridge :** Removable cartridges are hard disks encased in a metal or plastic cartridge, so you can remove them just like a floppy disk. Removable cartridges are very fast, though usually not as fast as fixed hard disks.

Optical Disks:

Optical disks record data by burning microscopic holes in the surface of the disk with a laser. To read the disk, another laser beam shines on the disk and detects the holes by changes in the reflection pattern.

Optical disks come in three basic forms:

- **CD-ROM** : Most optical disks are read-only. When you purchase them, they are already filled with data. You can read the data from a CD-ROM, but you cannot modify, delete, or write new data.
- **WORM** : Stands for *write-once, read-many*. WORM disks can be written on once and then read any number of times; however, you need a special WORM disk drive to write data onto a WORM disk.
- **Erasable optical (EO)** : EO disks can be read to, written to, and erased just like magnetic disks.

The machine that spins a disk is called a disk drive. Within each disk drive is one or more *heads* (often called *read/write heads*) that actually read and write data. Accessing data from a disk is not as fast as accessing data from main memory, but disks are much cheaper. And unlike RAM, disks hold on to data even when the computer is turned off. Consequently, disks are the storage medium of choice for most types of data. Another storage medium is magnetic tape. But tapes are used only for backup and archiving because they are *sequential-access* devices (to access data in the middle of a tape, the tape drive must pass through all the preceding data).

A new disk, called a *blank disk*, has no data on it. Before you can store data on a blank disk, however, you must *format* it.

3. PRIMARY MEMORY / VOLATILE MEMORY:

Primary memory is internal memory of the computer. It is also known as main memory and Temporary memory. Primary Memory holds the data and instruction on which computer is currently working. Primary Memory is nature volatile. It means when power is switched off it lost all data.

Types of Primary Memory– Primary memory is generally of two types.

1. **RAM**
2. **ROM**

1. RAM (Random Access Memory) – It stands for Random Access Memory. RAM is known as read /writes memory. It generally refereed as main memory of the computer system. It is a temporary memory. The information stored in this memory is lost as the power supply to the computer is switched off. That's why RAM is also called "**Volatile Memory**"

Types of RAM– RAM is also of two types:

a) Static RAM- Static RAM also known as SRAM ,retain stored information as long as the power supply is ON. SRAM are of higher coast and consume more power .They have higher speed than Dynamic RAM

b)Dynamic RAM– Dynamic RAM also known as DRAM, its stored information in a very short time (a few milliseconds) even though the power supply is ON. The Dynamic RAM are cheaper and moderate speed and also they consume less power.

2. ROM (Read Only Memory) – It stands for Read Only Memory. ROM is a Permanent Type memory. Its content are not lost when power supply is switched off. Content of ROM is decided by the computer manufacturer and permanently stored at the time of manufacturing. ROM cannot be overwritten by the computer. It is also called “**Non-Volatile Memory**”.

Type of ROM: ROM memory is three types names are following-

1. **PROM(Programmable Read Only Memory)**-PROM chip is programmable ROM.it is PROM chips to write data once and read many.once chip has been programmed ,the recorded information cannot be changed. PROM is also nonvolatile memory.
2. **EPROM(Erasable Programmable Read Only Memory)**- EPROM chip can be programmed time and again by erasing the information stored earlier in it. Information stored in EPROM exposing the chip for some time ultraviolet light .
3. **EEPROM (Electrically Erasable Programmable Read Only Memory)**-The EEPROM is programmed and erased by special electrical waves in millisecond. A single byte of a data or the entire contents of device can be erased.

3.2. SECONDARY MEMORY / NON VOLATILE MEMORY– Secondary Memory is external memory of the computer. It is also known as Auxiliary memory and permanent memory. It is used to store the different programs and the information permanently. Secondary Memory is nature non volatile. It means data is stored permanently even if power is switched off.

The secondary storage devices are:

1. Floppy Disks
2. Magnetic (Hard) Disk
3. Magnetic Tapes
4. Pen Drive
5. Winchester Disk (a disk drive in a sealed unit containing a high-capacity hard disk and the read-write heads.)
6. Optical Disk(CD,DVD)

4-PROCESSOR: A processor is an integrated electronic circuit that performs the calculations that run a computer. A processor performs arithmetical, logical, input/output (I/O) and other basic instructions that are passed from an operating system (OS). Most other processes are dependent on the operations of a processor.

The terms processor, CPU and microprocessor are commonly linked. A processor includes an arithmetical logic and control unit (CU), which measures capability in terms of the following:

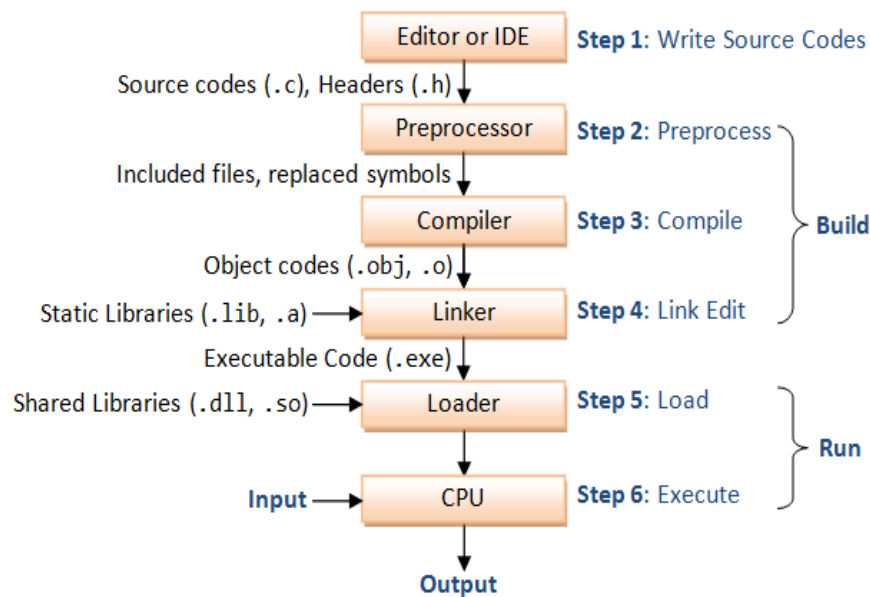
- Ability to process instructions at a given time
- Maximum number of bits/instructions
- Relative clock speed

Most people just say processor instead of "CPU" nowadays.

5. where a program is stored and executed:

- **Step 1:** Write the source codes (.c) and header files (.h).
- **Step 2:** Pre-process the source codes according to the *preprocessor directives*. The preprocessor directives begin with a hash sign (#), such as #include and #define. They indicate that certain manipulations (such as including another file or replacement of symbols) are to be performed BEFORE compilation.

- **Step 3:** Compile the pre-processed source codes into object codes (.obj, .o).
- **Step 4:** Link the compiled object codes with other object codes and the library object codes (.lib, .a) to produce the executable code (.exe).
- **Step 5:** Load the executable code into computer memory.
- **Step 6:** Run the executable code.



6. OPERATING SYSTEM:- An operating system (OS) is software that manages computer hardware and software resources and provides common services for computer. The operating system is an essential component of the system software in a computer system. An operating system is divided into two categories single user and multi user operating system.

i. Single User:- A single-user operating system is a type of operating system (OS) that is developed and intended for use on a computer or similar machine that will only have a single user at any given time. This is the most common type of OS used on a home computer, as well as on computers in offices and other work environments.

Eg: MS-DOS, Mobile Phone O.S

ii. Multi -User:- A multitasking OS can run multiple applications and programs at once. This is often used on computers where someone may wish to navigate the Internet, run a graphics editing program, play music through a media playing program, and type in notes in a simple word processing program all at the same time.

Eg:- Win-95/2000,win-2007,2008,vista,xp,Linux etc

7. COMPILERS: A compiler is a software program that transforms high-level source code that is written by a developer in a high-level programming language into a low level object code (binary code) in machine language, which can be understood by the processor. The process of converting high-level programming into machine language is known as compilation.

2.Idea of Algorithm: The actual term '**algorithm**' is often cited as originated from the 9th Century Persian mathematician Abu Abdullah Muhammad ibn Musa Al-Khwarizmi. Wow, that's quite a name but he's also known as "the father of Algebra". In fact, Al-Khwarizmi built on the work of Brahmagupta. Both of these two gentlemen were immortalized when the term "algorism" was coined to refer to Latin translations of their arithmetic rules by using Hindi-Arabic numerals. Around the 18th century "Allegorism" became the modern "algorithm". The use of the word evolved to include all definite procedures for solving problems or performing tasks.

An **Algorithm** is a step by step instruction of a problem which defines the solution of a particular problem in simple language. An algorithm is expressed in pseudo code which resembles somewhat with C

language, but with some statements in English rather than any kind of programming language. In Algorithm there is no ambiguity about which instruction is to be executed next.

Steps to solve logical and numerical problems.

A computer cannot solve a problem on its own. One has to provide step by step solutions of the problem to the computer. In fact, the task of problem solving is not that of the computer. It is the programmer who has to write down the solution to the problem in terms of simple operations which the computer can understand and execute.

In order to solve a problem by the computer, one has to pass through certain stages or steps. They are

1. Understanding the problem
2. Analyzing the problem
3. Developing the solution
4. Coding and implementation.

1. Understanding the problem: Here we try to understand the problem to be solved in totality. Before with the next stage or step, we should be absolutely sure about the objectives of the given problem.

2. Analyzing the problem: After understanding thoroughly the problem to be solved, we look for different ways of solving the problem and evaluate each of these methods. The idea here is to search for an appropriate solution to the problem under consideration. The end result of this stage is a broad overview of the sequence of operations that are to be carried out to solve the given problem.

3. Developing the solution: Here the overview of the sequence of operations that was the result of analysis stage is expanded to form a detailed step by step solution to the problem under consideration.

4. Coding and implementation: The last stage of the problem solving is the conversion of the detailed sequence of operations into a language that the computer can understand. Here each step is converted to its equivalent instruction or instructions in the computer language that has been chosen for the implementation.

Algorithm Definition

A set of sequential steps usually written in Ordinary Language to solve a given problem is called Algorithm.

It may be possible to solve a problem in more than one way, resulting in more than one algorithm. The choice of various algorithms depends on the factors like reliability, accuracy and easy to modify. The most important factor in the choice of algorithm is the time requirement to execute it, after writing code in High-level language with the help of a computer. The algorithm which will need the least time when executed is considered the best.

Steps involved in algorithm development

An algorithm can be defined as “a complete, unambiguous, finite number of logical steps for solving a specific problem “

Step1. Identification of input: For an algorithm, there are quantities to be supplied called input and these are fed externally. The input is to be identified first for any specified problem.

Step2: Identification of output: From an algorithm, at least one quantity is produced, called for any specified problem.

Step3 : Identification the processing operations : All the calculations to be performed in order to lead to output from the input are to be identified in an orderly manner.

Step4 : Processing Definiteness : The instructions composing the algorithm must be clear and there should not be any ambiguity in them.

Step5 : Processing Finiteness : If we go through the algorithm, then for all cases, the algorithm should terminate after a finite number of steps.

Step6 : Possessing Effectiveness : The instructions in the algorithm must be sufficiently basic and in practice they can be carried out easily.

3.Representation of Algorithm:

An algorithm must possess the following properties

1. Finiteness: An algorithm must terminate in a finite number of steps

2. Definiteness: Each step of the algorithm must be precisely and unambiguously stated

3. Effectiveness: Each step must be effective, in the sense that it should be primitive easily convertible into program statement) can be performed exactly in a finite amount of time.

4. Generality: The algorithm must be complete in itself so that it can be used to solve problems of a specific type for any input data.

5. Input/output: Each algorithm must take zero, one or more quantities as input data produce one or more output values. An algorithm can be written in English like sentences or in any standard representation sometimes, algorithm written in English like languages are called Pseudo Code

Example

1. Suppose we want to find the average of three numbers, the algorithm is as follows

Step 1 Read the numbers a, b, c

Step 2 Compute the sum of a, b and c

Step 3 Divide the sum by 3

Step 4 Store the result in variable d

Step 5 Print the value of d

Step 6 End of the program

2 Algorithms for Simple Problem

Write an algorithm for the following

1. Write an algorithm to calculate the simple interest using the formula.

Simple interest = $P \times N \times R / 100$.

Where P is principle Amount, N is the number of years and R is the rate of interest.

Step 1: Read the three input quantities' P, N and R.

Step 2 : Calculate simple interest as Simple interest = $P \times N \times R / 100$

Step 3: Print simple interest.

Step 4: Stop.

2. Area of Triangle: Write an algorithm to find the area of the triangle.

Let b, c be the sides of the triangle ABC and A the included angle between the given sides.

Step 1: Input the given elements of the triangle namely sides b, c and angle between the sides A.

Step 2: $\text{Area} = (1/2) * b * c * \sin A$

Step 3: Output the Area

Step 4: Stop.

3. Write an algorithm to find the largest of three numbers X, Y, Z.

Step 1: Read the numbers X, Y, Z.

Step 2: if $(X > Y)$

Big = X

else BIG = Y

Step 3 : if $(BIG < Z)$

Step 4: Big = Z

Step 5: Print the largest number i.e. Big

Step 6: Stop.

4. Write down an algorithm to find the largest data value of a set of given data values

Algorithm largest of all data values:

Step 1: LARGE \leftarrow 0

Step 2: read NUM

Step 3: While $\text{NUM} \geq 0$ do

3.1 if $\text{NUM} > \text{LARGE}$

3.1.1 then

3.1.1.1 LARGE \leftarrow NUM

3.2. read NUM

Step 4: Write “largest data value is”, LARGE

Step 5: end.

5. Write an algorithm which will test whether a given integer value is prime or not.

Algorithm prime testing:

Step 1: M \leftarrow 2

Step 2: read N

Step 3: MAX \leftarrow SQRT (N)

Step 4: While $M \leq \text{MAX}$ do

4.1 if $(M * (N/M) = N)$

4.1.1 then

4.1.1.1 go to step 7

4.2. $M \leftarrow M + 1$

Step 5: Write “number is prime”

Step 6: go to step 8

Step 7: Write “number is not a prime”

Step 8: end.

6. Write algorithm to find the factorial of a given number N

Step 1: $PROD \leftarrow 1$

Step 2: $I \leftarrow 0$

Step 3: read N

Step 4: While $I < N$ do

4.1 $I \leftarrow I + 1$

4.2. $PROD \leftarrow PROD * I$

Step 5: Write “Factorial of”, N, “is”, PROD

Step 6: end.

7. Write an algorithm to find sum of given data values until negative value is entered.

Algorithm Find – Sum

Step 1: $SUM \leftarrow 0$

Step 2: $I \leftarrow 0$

Step 3: read NEW VALUE

Step 4: While $NEW\ VALUE \leq 0$ do

4.1 $SUM \leftarrow SUM + NEW\ VALUE$

4.2 $I \leftarrow I + 1$

4.3 read NEW VALUE

Step 5: Write “Sum of”, I, “data value is”, SUM

Step 6: END

8. Write an algorithm to calculate the perimeter and area of rectangle. Given its length and width.

Step 1: Read length of the rectangle.

Step 2: Read width of the rectangle.

Step 3: Calculate perimeter of the rectangle using the formula $perimeter = 2 * (length + width)$

Step 4: Calculate area of the rectangle using the formula $area = length * width$.

Step 5: Print perimeter.

Step 6: Print area.

Step 7: Stop.

Flowchart:

A flow chart is a step by step diagrammatic representation of the logic paths to solve a given problem. Or A flowchart is visual or graphical representation of an algorithm.

The flowcharts are pictorial representation of the methods to be used to solve a given problem and help a great deal to analyze the problem and plan its solution in a systematic and orderly manner. A flowchart when translated into a proper computer language, results in a complete program.

Advantages of Flowcharts

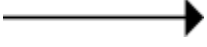

1. The flowchart shows the logic of a problem displayed in pictorial fashion which facilitates easier checking of an algorithm.
2. The Flowchart is good means of communication to other users. It is also a compact means of recording an algorithm solution to a problem.
3. The flowchart allows the problem solver to break the problem into parts. These parts can be connected to make master chart.
4. The flowchart is a permanent record of the solution which can be consulted at a later time.



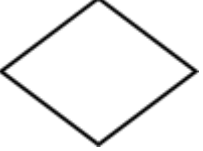



Differences between Algorithm and Flowchart

Algorithm	Flowchart
1. A method of representing the step-by-step logical procedure for solving a problem	1. Flowchart is diagrammatic representation of an algorithm. It is constructed using different types of boxes and symbols.
2. It contains step-by-step English descriptions, each step representing a particular operation leading to solution of problem	2. The flowchart employs a series of blocks and arrows, each of which represents a particular step in an algorithm
3. These are particularly useful for small problems	3. These are useful for detailed representations of complicated programs
4. For complex programs, algorithms prove to be inadequate	4. For complex programs, Flowcharts prove to be adequate

Symbols Used In Flowchart

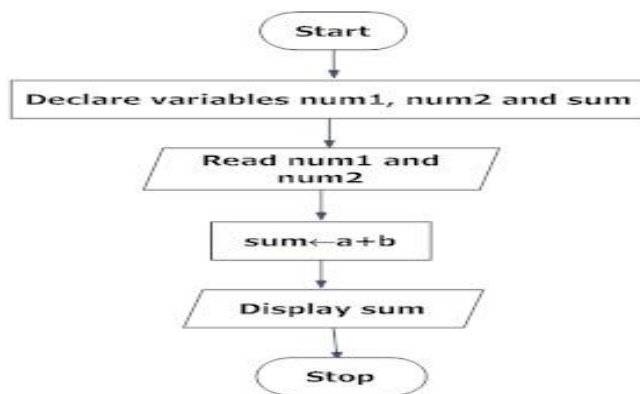
Different symbols are used for different states in flowchart, For example: Input/Output and decision making has different symbols. The table below describes all the symbols that are used in making flowchart

Symbol	Purpose	Description
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.

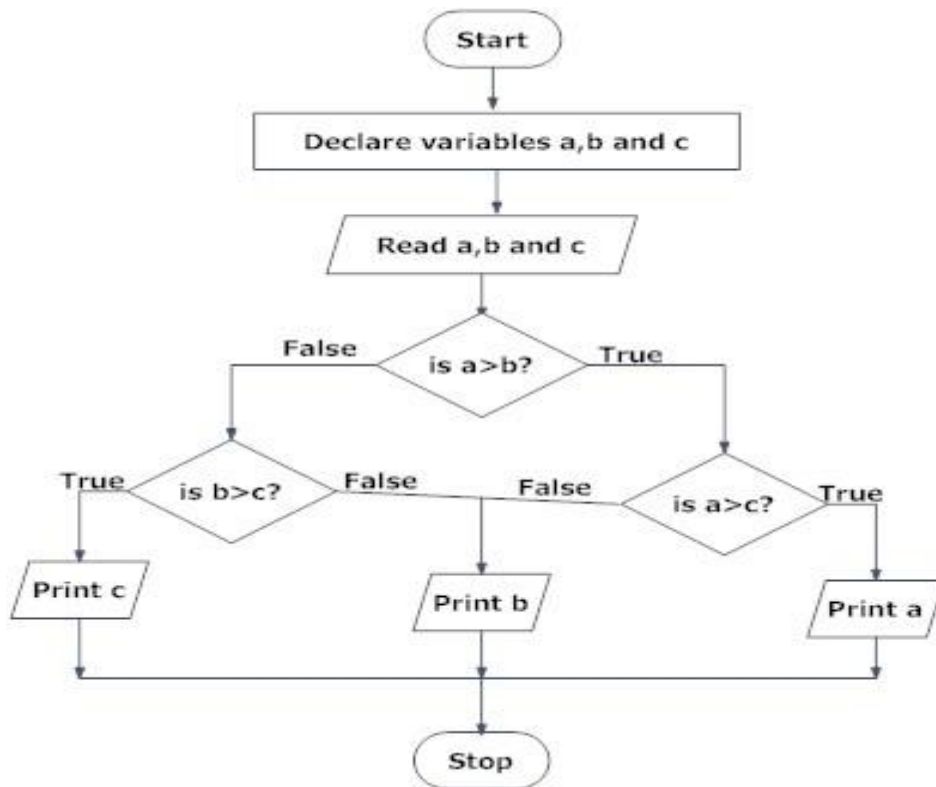
Symbol	Purpose	Description
	Input/Output	Used for input and output operation.
	Processing	Used for airthmetic operations and data-manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect flowchart portion on different page.
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

EXAMPLES OF FLOWCHARTS IN PROGRAMMING

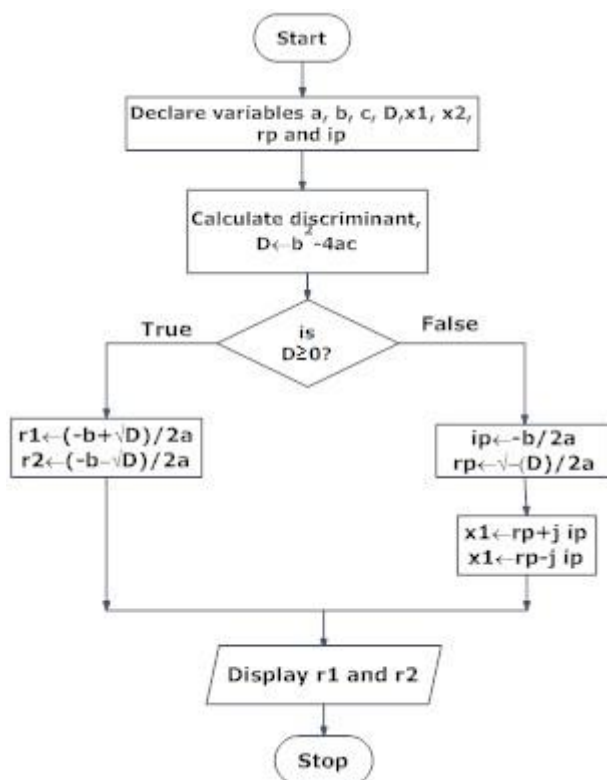
Draw a flowchart to add two numbers entered by user.



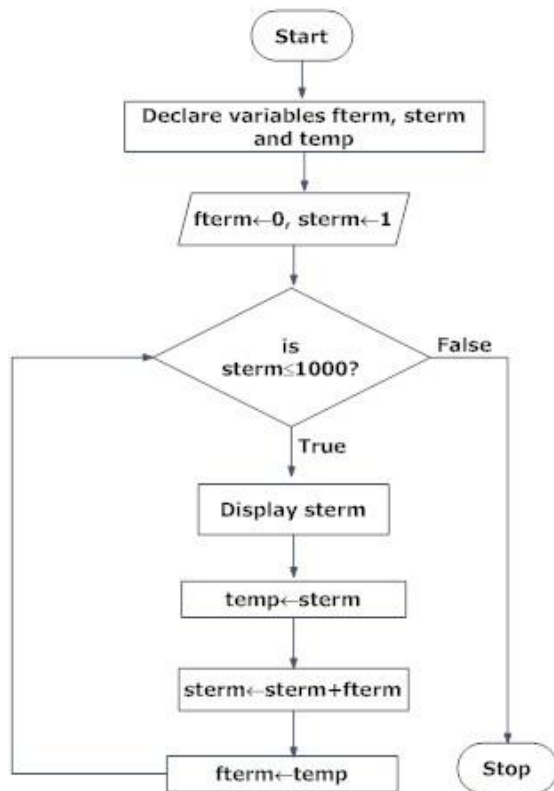
Draw flowchart to find the largest among three different numbers entered by user.



Draw a flowchart to find all the roots of a quadratic equation $ax^2+bx+c=0$



Draw a flowchart to find the Fibonacci series till $\text{term} \leq 1000$.



Though, flowchart are useful in efficient coding, debugging and analysis of a program, drawing flowchart in very complicated in case of complex programs and often ignored.

PSEUDO Code:

The Pseudo code is neither an algorithm nor a program. It is an abstract form of a program. It consists of English like statements which perform the specific operations. It is defined for an algorithm. It does not use any graphical representation. In pseudo code, the program is represented in terms of words and phrases, but the syntax of program is not strictly followed.

Advantages:

- * Easy to read,
- * Easy to understand,
- * Easy to modify.

Example: Write a pseudo code to perform the basic arithmetic operations.

Read n1, n2

Sum = n1 + n2

Diff = n1 - n2

Mult = n1 * n2

Quot = n1/n2

Print sum, diff, mult, quot

End.

PSEUDOCODE

Pseudocode can also be modular.

This means that large tasks can be sub divided into smaller tasks and each one planned individually. The tasks (modules) can then be put together to form the complete program.

PSEUDOCODE

The advantages of pseudocode are that:

- It is independent of a particular language so programmers of different languages can all understand it.
- Programmers do not have to worry about exact syntax.
- Lay people can help to determine the validity of the logic because it is not that difficult to understand

PSEUDOCODE:

The disadvantages of pseudo code are that

- It can become very detailed and so requires a high level of concentration to determine the overall logic
- The loose standards means that one solution may be depicted in several different ways by different persons.

VARIABLES:

- A variable is a name that may be used to store a data value.
- Unlike constant, variables are changeable, we can change value of a variable during execution of a program.
- A programmer can choose a meaningful variable name.

Example : average, height, age, total etc.

RULES TO DEFINE VARIABLE NAME:

- Generally most of the compilers support eight characters excluding extension However the ANSI standard recognizes the maximum length of a variable up to 31 characters.
- Variable name must not start with a digit.
- Variable name can consist of alphabets, digits and special symbols like underscore _.
- Blank or spaces are not allowed in variable name.
- Keywords are not allowed as variable name.

DECLARATION OF VARIABLE

- Declaration of variables must be done before they are used in the program.

Declaration does two things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

EXAMPLE: WRITE A PROGRAM FOR ADDITION OF TWO NUMBERS.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,sum; //variable declaration
    a=30;
    b=20;
    sum=a+b;
    printf("Sum is %d",sum);
    getch();
}
```

Output:

Sum is 50

INITIALIZING VARIABLES

- Variable declared can be assigned or initialized using assignment operator “=” . The declaration and initialization can also be done in the same line.

Syntax :

Variable_name = constant;

Or

Data_type variable_name = constant ;

Example :

X = 10; //where x is an integer variable

Example :

int y = 21;

X=Y=Z=3; is invalid assignment.

DYNAMIC INITIALIZATION

- The initialization of variable at run time is called dynamic initialization.
- Dynamic is the process during execution.

EXAMPLE:

```
#include<stdio.h>
void main()
{
int r =2;
float area;
area = 3.14 * r *r;
clrscr();
printf(“area is = %f”,area);
}
```

Output

area =12.56

Explanation:

- In the above program area is calculated and assigned to variable area.
- The expression is solved at a run time and assigned to area at a run time.
- Hence it is called dynamic initialization.

DATA TYPE: A programming language is proposed to help programmer to process certain kinds of data and to provide useful output.

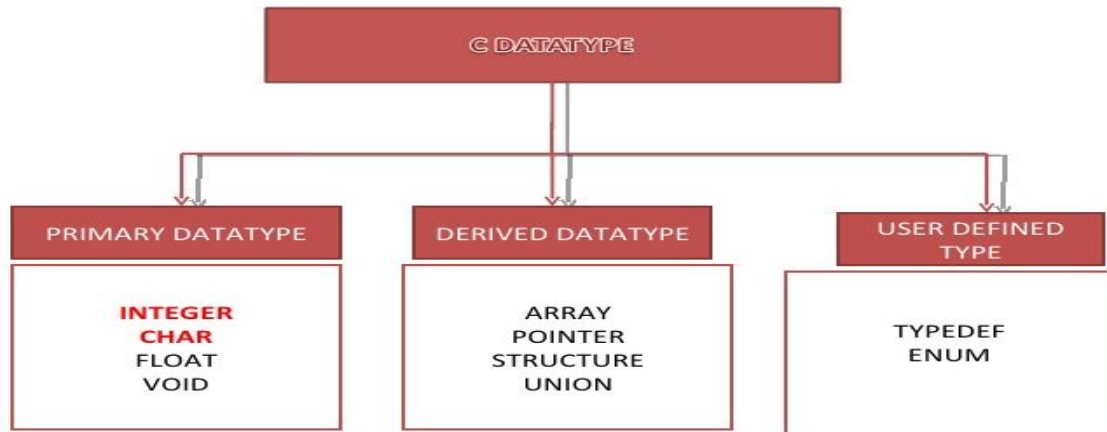
A program usually contains different types of data types (integer, float, character etc.) and need to store the values being used in the program. C language is rich of data types. A C programmer has to employ proper data type as per his requirement.

C has different data types for different types of data and can be broadly classified as :

1. Primary data types

2. User defined data types

3. Derived data types



For 16 or 32 bit compiler

int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
Char	1 byte	-128 to 127 or 0 to 255
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255

1.17 Syntax and logical errors in compilation:

Error is an illegal operation performed by the user which results in abnormal working of the program.

Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing. The most common errors can be broadly classified as follows.

Type of errors

- **Syntactic Errors:** These errors occur due to the usage of wrong syntax for the statements.

Syntax means rules of writing the program.

Example: `x=z*/b;`

There is syntax error in this statement. The rules of binary operators state that there cannot be more than one operator between two operands.

1. Errors that occur when you **violate the rules** of writing C/C++ syntax are known as syntax errors. This compiler error indicates something that must be fixed before the code can be compiled. All these

errors are detected by compiler and thus are known as compile-time errors.

Most frequent syntax errors are:

- Missing Parenthesis (})
- Printing the value of variable without declaring it
- Missing semicolon like this:

```
// C program to illustrate
// syntax error
#include<stdio.h>
void main()
{
    int x = 10;
    int y = 15;

    printf("%d", (x, y)) // semicolon missed
}
```

OUTPUT:

Error:

- error: expected ';' before '}' token
- Syntax of a basic construct is written wrong. For example : while loop

```
// C program to illustrate
// syntax error
#include<stdio.h>
int main(void)
{
    // while() cannot contain "." as an argument.
    while(.)
    {
        printf("hello");
    }
    return 0;
}
```

OUTPUT

Error:

error: expected expression before '.' token

- while(.)

- In the given example, the syntax of while loop is incorrect. This causes a syntax error.

Logical Errors : These Errors occur due to incorrect usage of the instruction in the program. These errors are neither displayed during compilation or execution nor cause any obstruction to the program execution. They only cause incorrect outputs. Logical Errors are determined by analyzing the outputs for different possible inputs that can be applied to the program. By this way the program is validated.

On compilation and execution of a program, desired output is not obtained when certain input values are given. These types of errors which provide incorrect output but appears to be error free are called logical errors. These are one of the most common errors done by beginners of programming.

These errors solely depend on the logical thinking of the programmer and are easy to detect if we follow the line of execution and determine why the program takes that path of execution.

```

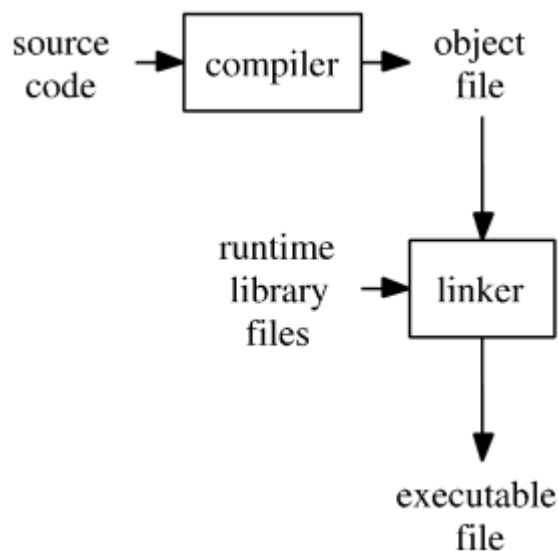
// C program to illustrate
// logical error
int main()
{
    int i = 0;

    // logical error : a semicolon after loop
    for(i = 0; i < 3; i++);
    {
        printf("loop ");
        continue;
    }
    getchar();
    return 0;
}

```

1.18 OBJECT AND EXECUTABLE CODE:

- **OBJECT CODE:** Translation of the source code of a program into machine code, which the computer can read and execute directly. Object code is input to the linker.
- The following illustrates the programming process for a compiled programming language.



- A compiler takes the program code (source code) and converts the source code to a machine language module (called an object file). Another specialized program, called a linker, combines this object file with other previously compiled object files (in particular run-time modules) to create an executable file. This process is diagrammed below. Click **Initial build** to see an animation of how the executable is created. Click **Run executable** to simulate the running of an already created executable file. Click **Rebuild** to simulate rebuilding of the executable file.
- executable file
- So, for a compiled language the conversion from source code to machine executable code takes place before the program is run. This is a very different process from what takes place for an interpreted programming language.
- This is somewhat simplified as many modern programs that are created using compiled languages makes use of dynamic linked libraries or shared libraries. Therefore, the executable file may require these dynamic linked libraries (Windows) or shared libraries (Linux, Unix) to run.

Unit – II

Control Structures: Arithmetic expressions and precedence, Conditional Branching and Loops, Writing and evaluation of conditionals and consequent branching. Arrays: Arrays (1-D, 2-D), Character arrays and Strings.

2. **Control Structures:** In programming, “control structures” are the structures used to control the flow of a program. The two main classes are conditionals and loops. In C, blocks (also called compound statements) group multiple statements together so they can be treated as one.

2.1: Arithmetic expressions and precedence

Expression:

Expression is an expression whose value is assigned to the constant. The expression must be enclosed in brackets if it contains operators.

- Arithmetic Expressions
- Evaluation of Expressions
- Precedence in Arithmetic Operators
- Rules for evaluation of expression
- Type conversions in expressions
- Implicit type conversion
- Explicit Conversion
- Operator precedence

ARITHMETIC EXPRESSIONS

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

```
main ()
{
float a, b, c x, y, z;

a = 9;

b = 12;

c = 3;

x = a - b / 3 + c * 2 - 1;

y = a - b / (3 + c) * (2 - 1);

z = a - ( b / (3 + c) * 2) - 1;

printf (“x = %fn”,x);

printf (“y = %fn”,y);

printf (“z = %fn”,z);
```

}

An expression is a combination of variables constants and operators written according to the syntax of C language.

Algebraic Expression	C Expression
$a \times b - c$	$a * b - c$
$(m + n) (x + y)$	$(m + n) * (x + y)$
(ab / c)	$a * b / c$
$3x^2 + 2x + 1$	$3 * x * x + 2 * x + 1$
$(x / y) + c$	$x / y + c$

EVALUATION OF EXPRESSIONS:-

Expressions are evaluated using an assignment statement of the form

Variable=expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Example of evaluation statements are

```
x = a * b - c ;  
y = b / c * a;  
z = a - b / c + d;
```

PRECEDENCE IN ARITHMETIC OPERATORS:-

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority ==> * / %

Low priority ==> + -

RULES FOR EVALUATION OF EXPRESSION:-

First parenthesized sub expression left to right are evaluated. If parenthesis are nested, the evaluation begins with the innermost sub expression. The precedence rule is applied in determining the order of application of operators in evaluating sub expressions. The associability rule is applied when two or more operators of the same precedence level appear in the sub expression. Arithmetic expressions are evaluated from left to right using the rules of precedence. When Parenthesis are used, the expressions within parenthesis assume highest priority.

A statement is a block of code that does something. An assignment statement assigns a value to a variable. A for statement performs a loop.

In C, Statements can be grouped together as one statement using curly brackets

```
{  
    statement1;  
    statement2;  
}
```

Examples:

A=100;

I++;

Printf(.....);

An expression is a statement that has a value.

2.2: CONTROL STATEMENT:-

Control statements specifies the order in which the various instructions in a program are to be executed by the computer i.e they determine the “flow of control” in a program. Various control statements in c are:

- Sequence control statements
- Selection and Decision control statements
- Case control statements
- Repetition or loop control statements

The sequence control statement ensures that the instruction in the program are executed in the same order in which they appear in the program. Decision and case control statements allow the computer take a decision as to which statements repeatedly till a condition satisfied.

SELECTION STATEMENT:-

A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is **true** or not.

The following keywords are used in selection statements:

- if
- else
- switch
- case

IF STATEMENTS:- the if statement may be implemented in different forms.

- Simple if statement
- If...else statement
- Nested if...else statement
- Else if ladder

SIMPLE IF STATEMENT:-The simplest form of the control statement is the If statement. It is very frequently used in decision making and allowing the flow of program execution.

syntax:

```
if(expression)

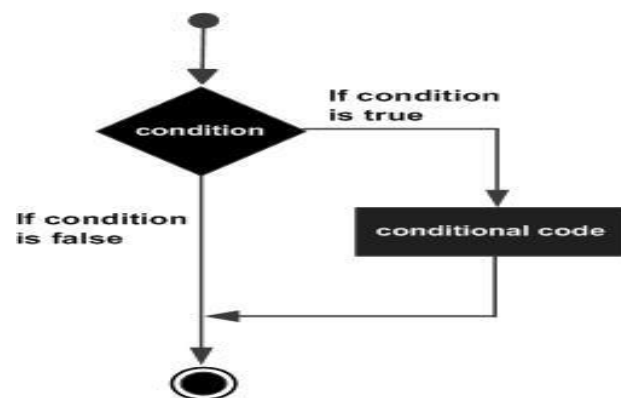
{
    statement;
}
```

The statement is any valid C' language statement and the condition is any valid C' language expression, frequently logical operators are used in the condition statement. The condition part should not end with a semicolon, since the condition and statement should be put together as a single statement.

Example:if(marks>100)

Printf(“invalid marks”);

Flow Diagram:



Program for if statement

```
#include<stdio.h>

int main ()
{
    /* local variable definition */
    int a =10;

    /* check the boolean condition using if statement */
    if( a <20)
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n");
    }
    printf("value of a is : %d\n", a);

    return 0;
}
```


When the above code is compiled and executed, it produces the following result:

```
a is less than 20;  
value of a is : 10
```

IFELSE STATEMENT:- the if.....else statement is an extension of the simple if statement.

If else syntax:

```
If (expression)  
{  
statement 1;  
}  
Else  
{  
statement 2;  
}
```

The if else is actually just an extension of the general format of if statement. If the result of the condition is true, then program statement 1 is executed, otherwise program statement 2 will be executed. If any case either program statement 1 is executed or program statement 2 is executed but not both when writing programs this else statement is so frequently required that almost all programming languages provide a special construct to handle this situation.

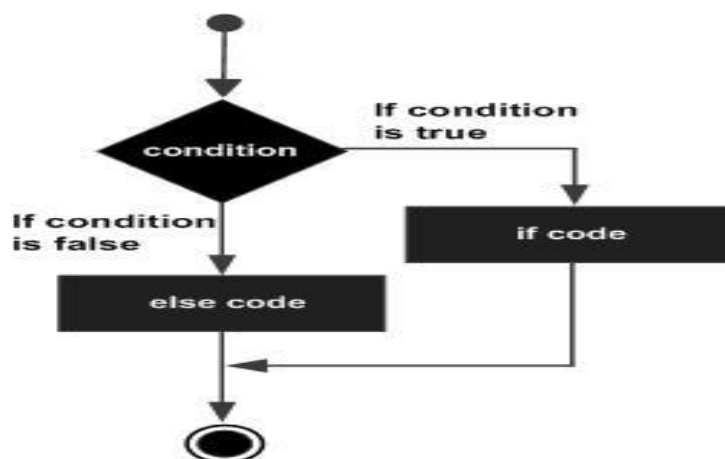
Example:if(marks>100)

```
Printf("invalid marks");
```

```
Else
```

```
Printf("marks+>%d\n",marks);
```

Flow Chart:-



Example:

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

```
int main ()  
{
```

```

/* local variable definition */
int a =100;

/* check the boolean condition */
if( a <20)
{
/* if condition is true then print the following */
printf("a is less than 20\n");
}
else
{
/* if condition is false then print the following */
printf("a is not less than 20\n");
}
printf("value of a is : %d\n", a);

return 0;
}

```

When the above code is compiled and executed, it produces the following result:

```

a is not less than 20;
value of a is : 100

```

NESTED IF....ELSE STATEMENT:- when a series of decisions are involved, we may have to use more than one if.....else statement in nested form as follows:

Syntax:

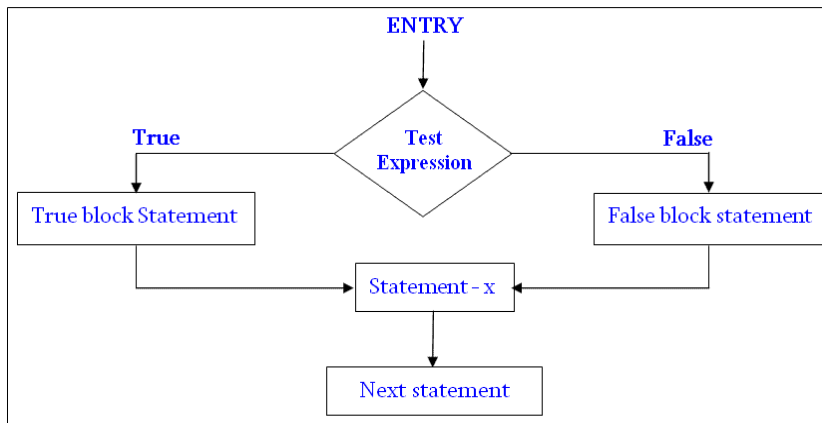
```

if (expression 1)
{
    if (expression 2)
    {
        .....
    }
    If (expression 3)
    {
        statement-1;
    }
    else
    {
        statement-2;
    }
    else
    {
        statement-3;
    }
    Statement 4;
}

```

if statement may be nested as deeply as you need to nest it. One block of code will be executed if two expression are true. expression 1 is tested first and then expression 2 is tested. The second if expression is nested in the first. The second if expression is tested only when the first expression is true else the program flow will skip to the corresponding else statement.

Nested if else Flow Chart:-



Nested if else Program:-

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

{
int A,B,C;

clrscr();
printf("Enter any three integer number here:\n");

scanf("%d%d%d",&A,&B,&C);
if(A>B)
{

if(A>C)
    printf("Number %d is largest",A);

else
    printf("Number %d is largest",C);
}
else
{

if(B>C)
    printf("Number %d is largest",B);

else
    printf("Number %d is largest",C);

}

getch();
}
```

Output:

Enter any three integer number here:
45

65
78
Number 78 is largest

IF....ELSE LADDER:-

When a series of many conditions have to be checked we may use the ladder else if statement which takes the following general form.

If....Else Ladder Syntax:-

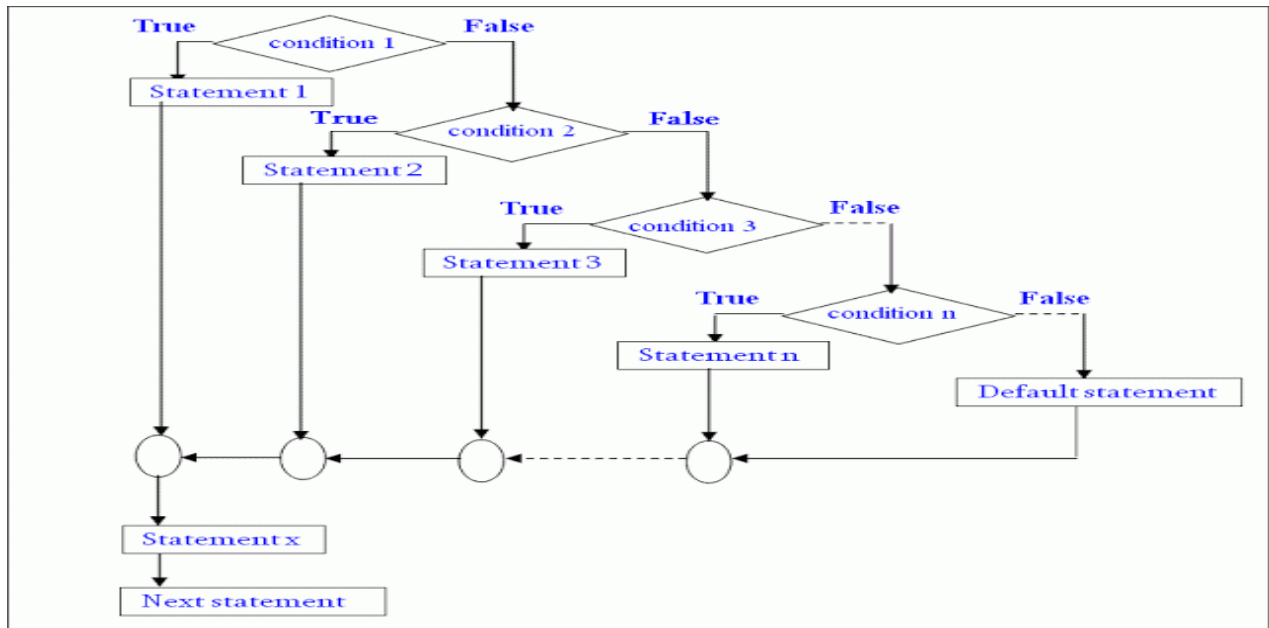
```
if (condition1)
{
statement – 1;
}
else if (condition2)
{
statement2;
}
else if (condition3)
{
statement3;
}
else if (condition)
{
statement n;
}
else
{
default statement;
}
statement-x;
```

Syntax description

This construct is known as if else construct or ladder. The conditions are evaluated from the top of the ladder to downwards. As soon on the true condition is found, the statement associated with it is executed and the control is transferred to the statement – x skipping the rest of the ladder.

When all the condition becomes false, the final else containing the default statement will be executed.

If....Else Ladder Flow chart:-



If-else ladder program (first program as per OU lab manual)

```
#include<stdio.h>

void main()
{
    int a,b,c;
    clrscr();
    printf("Enter 3 numbers:");
    scanf("%d%d%d",&a,&b,&c);
    if(a>b && a>c)
        printf("Maximum number is a = %d",a);
    else if(b>a && b>c)
        printf("Maximum number is b = %d",b);
    else
        printf("Maximum number is c = %d",c);
    if(a<b && a<c)
        printf("Minimum number is a = %d",a);
    else if(b<a && b<c)
        printf("Minimum number is b = %d",b);
    else
        printf("Minimum number is c = %d",c);
    getch();
}
```

Output:

Enter 3 numbers: 89

79

99

Maximum number is c=99

SWITCH AND BREAK STATEMENTS:-

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax:

The syntax for a **switch** statement in C programming language is as follows:

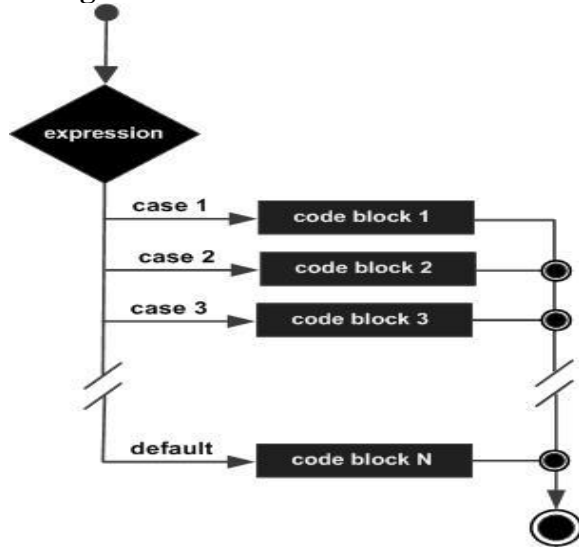
```
switch(expression)
{
case expression 1 :
block-1;
        break;/* optional */
case expression 2 :
        block-2;
        break;/* optional */

/* you can have any number of case statements */
default:/* Optional */
break;
}
statement(x);
```

The following rules apply to a **switch** statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will fall through to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram:-



Example:

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

```
int main ()
```

```
{
```

```
/* local variable definition */
```

```
char grade ='B';
```

```
switch(grade)
```

```
{
```

```
case'A':
```

```
printf("Excellent!\n");
```

```
break;
```

```
case'B':
```

```
case'C':
```

```
printf("Well done\n");
```

```
break;
```

```
case'D':
```

```
printf("You passed\n");
```

```
break;
```

```
case'F':
```

```
printf("Better try again\n");
```

```
break;
```

```
default:
```

```
printf("Invalid grade\n");
```

```
}
```

```
printf("Your grade is %c\n", grade );
```

```
return0;
```

```
}
```

Output:

```
Well done
Your grade is B
```

GOTO STATEMENTS:-

A **goto** statement in C programming language provides an unconditional jump from the goto to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

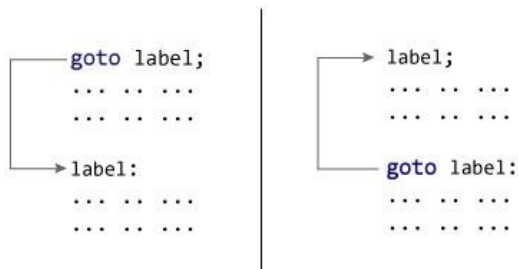
Syntax:

The syntax for a **goto** statement in C is as follows:

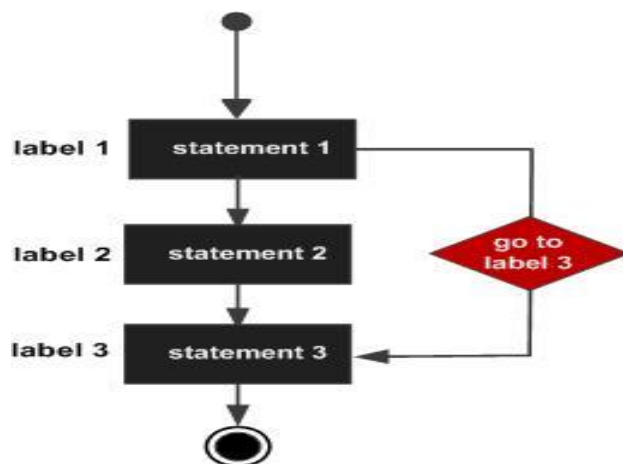
```
goto label;  
..  
.  
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

In this syntax, label is an identifier. When, the control of program reaches to goto statement, the control of the program will jump to the label: and executes the code below it.



Flow Diagram:



Example:

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



```

int number;
clrscr();
printf(www.);
goto x;
y:
printf(“expert”);
goto z;
x:
printf(“c programming”);
goto y;
z:
printf(“.com”);
getch();
}

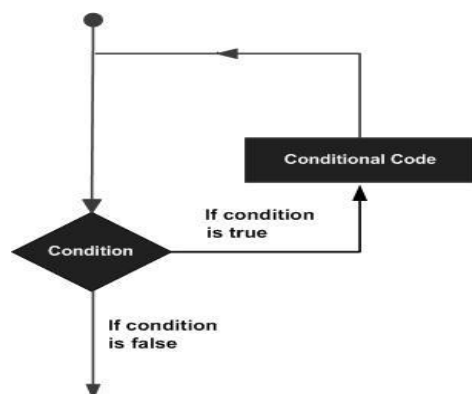
```

CONTINUE STATEMENT:- Continue statement skip the body of loop and jump on next value on a particular condition.

LOOPS/REPETITIVE STATEMENTS:-

Looping means a group of statements are executed repeatedly, until some logical condition is satisfied and C language provides three looping/iterative/repetitive statements.

Flow chart of looping:-



They are

- While do statement
- Do while statement
- For loop statement

WHILE STATEMENT:-

The while statement is also a looping statement. The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It has the following

syntax.

while (expression)

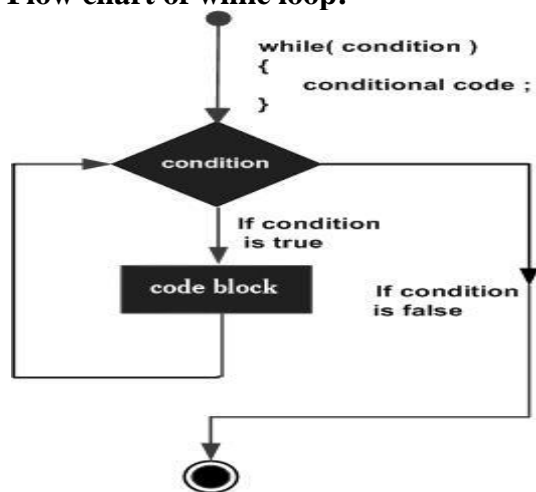
```

{
Statement 1;
Statement 2;
.....
Statement n;
}

```

Here the initialization of a loop control variable is generally done before the loop separately. The testing of the condition is done in while by evaluating the expression within the parenthesis. If the evaluation result is true value, the block of statement within calibrates is executed. If the evaluation result is false value the block of statements within the body of loop is skipped and the loop execution get terminated with the control passing to the statement immediately following the while construct. The increment or decrement of the loop control variable is generally done within the body of the loop.

Flow chart of while loop:-



The following example illustrates the use of while loop.

Program For while statement:-

```

#include<stdio.h>
#include<conio.h>
int main()
{
int i;
clrscr();
i=1;
while(i<5)
{
printf("%d\n",i);
i++;
}
}

```

```

}
getch();
return 0;
}

```

Output:-

```

1
2
3
4
5

```

DO-WHILE LOOP:-

This construct is also used for looping. In this case the loop condition is tested at the end of the body of the loop. Hence the loop is executed at least one. The do-while is an unpopular area of the language, most programmers' tries to use the straight while if it is possible.

Syntax of do while loop:-

```

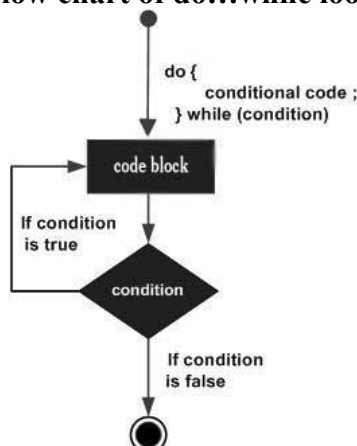
do
{
Statement 1;
Statement 2;
.....
Statement n;
}
while(expression);

```

Here the block of statement following the do is executed without any condition check. After this expression is evaluated and if it is true the block of statement in the body of the loop is executed again. Thus the block of statement is repeatedly executed till the expression is evaluated to false.

do-while construct is not used as often as the while loops or for loops in normal case of iteration but there are situation where a loop is to be executed at least one, in such cases this construction is very useful.

Flow chart of do...while loop:-



Program For do-while statement:-

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i;
clrscr();
i=1;
do
{
printf(“%d\n”,i);
i++;
}
while(i<5)
getch();
return 0;
}
```

output:-

1
2
3
4
5

FOR LOOP:-

For loop in C is the most general looping construct. The loop header contains three parts: an initialization, a continuation condition, and step.

Syntax:

```
for (initialization, condition, step)
{
Statement 1;
Statement 2;
.....
Statement n;
}
```

For statement contain three parts separated by two semicolons. The first part is known as initialization. The variable called loop control variable or index variable is initialized in this part.

The second part is known as the condition. The condition should be a value one. The condition check can be a compound expression made up of relational expression connected by logical AND, OR.

The third part is known as step. It should be an arithmetic expression. The initialization need not be contained to a single variable. If more than one variable is used for initialization they are separated by commas. The step can also applied to more than one variable separated by commas.

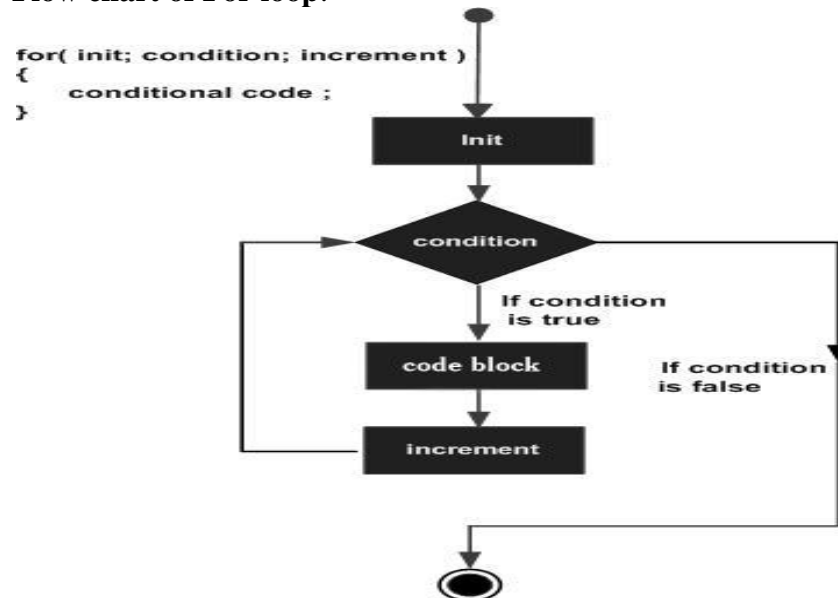
When for statement is encountered at first index variable is get initialized. This process is done only once during the execution of for statement. When condition is evaluated, if it is true the body of the loop will be executed. If more than one statement has to be executed it should end with a pair of braces. The

condition of for loop is executed each time at the beginning of the loop. After executing the body of the loop the step is executed, again the condition is executed. If the condition become false it exit from the loop and control transferred to the statement followed by the loop.

The following example executes 10 times by counting 0..9.
for (i = 0; i < 10; i++)

The following example illustrates the use of for loop.

Flow chart of For loop:-



Program For-loop statement:-

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i;
    clrscr();
    for(i=1;i<=3;i=i+1)
    printf("%d\n",i);
    for(i=1;i<5;i++)
    {
        printf("%d",i);
        printf("\\n");
    }
    getch();
    return 0;
}
```

output:-

1
2
3
4
5

Arrays: Arrays (1-D, 2-D), Character arrays and Strings:

Array can be defined as a collection of data objects which are stored in consecutive memory locations with a common variable name.

OR

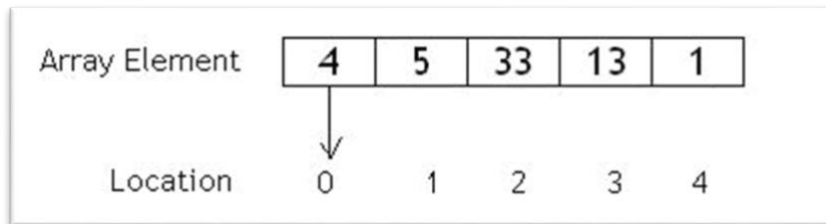
Array can be defined as a group of values referred by the same variable name.

OR

An Array can be defined as a collection of data objects which are stored in consecutive memory locations with a common variable name.

The individual values present in the array are called elements of array. The array elements can be values or variables also.

Ex:



Types of Arrays

Basically arrays can divide in two types

1. One Dimensional Array

An array with only one subscript is called as *one-dimensional array* or *1-d array*. It is used to store a list of values, all of which share a common name and are separable by subscript values

2. Two Dimensional Array

An array with two subscripts is termed as two-dimensional array. A two-dimensional array, it has a list of given variable -name using two subscripts. We know that a one-dimensional array can store a row of elements, so, a two-dimensional array enables us to store multiple rows of elements.

2.1 Initialization of Array

Array can be made initialized at the time of declaration itself. The general form of array initialization is as below type name[n] = [element1, element2, element n];

The elements 1, element2... element n are the values of the elements in the array referenced by the same.

Example1:- int codes[5] = [12,13,14,15,16];

Example2:- float a[3] = [1.2, 1.3, 1.4];

Example3:- char name [5] = ['S', 'U', 'N', 'I', 'L'];

In above examples, let us consider one, it a character array with 5 elements and all the five elements area initialized to 5 different consecutive memory locations.

name[0] = 'S'

name[1] = 'U'

name[2] = 'N'

name[3] = 'I'

name[4] = 'L'

Rules for Array Initialization

1. Arrays are initialized with constants only.
2. Arrays can be initialized without specifying the number of elements in square brackets and this number automatically obtained by the compiler.
3. The middle elements of an array cannot be initialized. If we want to initialize any middle element then the initialization of previous elements is compulsory.
4. If the array elements are not assigned explicitly, initial values will be set to zero automatically.
5. If all the elements in the array are to be initialized with one and same value, then repletion of data is needed.

2.2 Declaration of Array

The array must be declared as other variables, before its usage in a C program. The array declaration included providing of the following information to C compiler.

- The type of the array (ex. int, float or char type)
- The name of the array (ex A[],B[], etc)
- Number of subscripts in the array (i.e whether one – dimensional or Two-dimensional)
- Total number of memory locations to be allocated.

The name of the array can be kept by the user (the rule similar to naming to variable).

There is no limit one on dimensions as well as number of memory locations and it depends o the capacity of computer main memory..

The general form for array declaration is

Type name[n] ; { one dimensional array }

Ex : int marks[20];

char name[15];

float values [10];

An array with only one subscript is called as *one-dimensional array* or *1-d array*. It is used to store a list of values, all of which share a common name and are separable by subscript values.

Declaration of One-dimensional Arrays:

The general form of declaring a one-dimensional array is

data-type	array-name [size];
-----------	--------------------

Where data-type refers to any data type supported by C, array-name should be a valid C identifier; the size indicates the maximum number of storage locations (elements) that can be stored. Each element in the array is referenced by the array name followed by a pair of square brackets enclosing a subscript value. The subscript value is indexed data-type array-name [size]; from 0 to size -1. When the subscript vale is 0, first element in the array is

selected, when the subscript value is 1, second element is selected and so on.

Example

int x [6];

Here, x is declared to be an array of int type and of size six. Six contiguous memory locations get allocated as shown below to store six integer values.

1	2	3	4	5	6
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Each data item in the array x is identified by the array name x followed by a pair of square brackets enclosing a subscript value. The subscript value is indexed from 0 to 5. i.e., x[0] denotes first data item, x[1] denotes second data item and x[5] denotes the last data item.

Initialization of One-Dimensional Arrays:

Just as we initialize ordinary variables, we can initialize one-dimensional arrays also, i.e., locations of the arrays can be given values while they are declared.

The general form of initializing an array of one-dimension is as follows:

data - type array - name [size] = {list of values};

The values in the list are separated by commas.

Example

int x [6] = { 1, 2, 3, 4, 5, 6 };

as a result of this, memory locations of x get filled up as follows:

1	2	3	4	5	6
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Points to be considered during the declaration

1. If the number of values in initialization value - is less then the size of an array, only those many first locations of the array are assigned the values. The remaining locations are assigned zero.

Example: `int x [6] = { 7, 8, 6 };`

The size of the array x is six, initialization value - consists of only three locations get 0 assigned to them automatically, as follows:

7	8	6	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

2. If the number of values listed within initialization value - list for any array is greater than the size the array, compiler raises an error.

Example:

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

The size of the array x is six. But the number of values listed within the initialization - list is eight. This is illegal.

3. If a static array is declared without initialization value - list then the all locations are set to zero.

Example:

`static int x [6];`

0	0	0	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

4. If size is omitted in a 1-d array declaration, which is initialized, the compiler will supply this value by examining the number of values in the initialization value - list.

Example:

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

0	0	0	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Since the number of values in the initialization value-list for the array x is six, the size of x is automatically supplied as six.

5. There is no array bound checking mechanism built into C-compiler. It is the responsibility of the programmer to see to it that the subscript value does not go beyond size-1. If it does, the system may crash.

Example:

`int x [6];`

`x [7] = 20;`

Here, x [7] does not belong to the array x, it may belong to some other program (for example, operating system) writing into the location may lead to unpredictable results or even to system crash.

6. Array elements can not be initialized selectively.

Example:

An attempt to initialize only 2nd location is illegal, i.e.,

`int x [6] = { , 10 }` is illegal.

Similar to arrays of int type, we can even declare arrays of other data types supported by C, also.

Example

`char ch [6];`

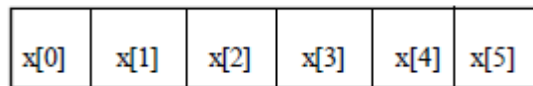
ch[0]	ch[1]	ch[2]	ch[3]	ch[4]	ch[5]

ch is declared to be an array of char type and size 6 and it can accommodate 6 characters.

Example:

```
float x [6];
```

x is declared to be an array of float type and size 6 and it can accommodate 6 values of float type. Following is the scheme of memory allocation for the array x:

**Note**

A one array is used to store a group of values. A loop (using, for loop) is used to access each value in the group.

Example

Program to illustrate declaration, initialization of a 1-d array

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

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

```
void main( )
```

```
{
```

```
int i, x [6] = { 1, 2, 3, 4, 5, 6 };
```

```
clrscr();
```

```
printf("The elements of array x \n");
```

```
for(i=0; i<6; i++)
```

```
printf("%d", x [i]);
```

```
getch();
```

```
}
```

Input – Output:

The elements of array x

1 2 3 4 5 6

4.3 Two - Dimensional Array

An array with two subscripts is termed as two-dimensional array. A two-dimensional array, it has a list of given variable -name using two subscripts. We know that a one-dimensional array can store a row of elements, so, a two-dimensional array enables us to store multiple rows of elements.

Example: Table of elements or a Matrix.

Syntax of two-dimensional arrays:

The syntax of declaring a two-dimensional array is:

data - type array - name [rowsize] [colsize];
--

Where, data-type refers to any valid C data type, array -name refers to any valid C identifier, row size indicates the number of rows and column size indicates the number of elements in each column. Row size and column size should be integer constants.

Total number of location allocated = (row size * column size).

Each element in a 2-d array is identified by the array name followed by a pair of square brackets enclosing its row-number, followed by a pair of square brackets enclosing its column-number. Row-number range from 0 to row size-1 and column-number range from 0 to column size-1.

Example: int y [3] [3];

y is declared to be an array of two dimensions and of data type Integer (int), row size and column size of y are 3 and 3, respectively. Memory gets allocated to store the array y as follows. It is important to note that y is the common name shared by all the elements of the array.

		Column numbers		
		0	1	2
Row numbers	1	x[0][0]	x[0][1]	x[0][2]
	2	x[1][0]	x[1][1]	x[1][2]
	3	x[2][0]	x[2][1]	x[2][2]

Each data item in the array y is identifiable by specifying the array name y followed by a pair of square brackets enclosing row number, followed by a pair of square brackets enclosing column number.

Row-number ranges from 0 to 2. that is, first row is identified by row number 0, second row is identified by row-number 1 and so on.

Similarly, column-number ranges from 0 to 2. First column is identified by column-number 0, second column is identified by column-number 1 and so on.

x [0] [0] refers to data item in the first row and first column

x [0] [2] refers to data item in the first row and third column

x [2] [3] refers to data item in the third row and fourth column

x [3] [4] refers to data item in the fourth row and fifth column

Initialization of Two-Dimensional Array

There are two forms of initializing a 2-d array.

First form of initializing a 2-d array is as follows:

```
data - type array - name [rowsize][colsize] = [initializer - list];
```

Where, data-name refers to any data type supported by C. Array-name refers to any valid C identifier. Row size indicates the number of rows, column size indicates the number of columns of the array. initializer-list is a comma separated list of values.

If the number of values in initializer-list is equal to the product of row size and column size, the first row size values in the initializer-list will be assigned to the first row, the second row size values will be assigned to the second row of the array and so on

Example: int x [2] [4] = { 1, 2, 3, 4, 5, 6, 7, 8 };

Since column size is 4, the first 4 values of the initializer-list are assigned to the first row of x and the next 4 values are assigned to the second row of x as shown hereinafter

1 2 3 4

5 6 7 8

Note: If the number of values in the initializer-list is less than the product of row size and colsize, only the first few matching locations of the array would get values from the initializer-list row-wise. The trailing unmatched locations would get zeros.

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

The first row of x gets filled with the values in the initializer-list. The second row gets filled with zeros.

1	2	3	4
0	0	0	0

The second form of initializing a 2-d array is as follows:

```
data-type array-name [rowsize] [colsize] = { { initializer-list1 }, { initializer-list2 }, ..... };
```

The values in initialize list 1 are assigned to the locations in the first row. The values in initializer-list2 are assigned to the locations in the second row and so on.

Example: int x [2] [4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };

data - type array - name [rowsize][colsize] = [initializer - list];

As a result of this, the array x gets filled up as follows:

1	2	3	4
5	6	7	8

Note

1. If the number of values specified in any initializer-list is less than col size of x, only those may first locations in the corresponding row would get these values. The remaining locations in that row would get 0.

Example: `int x [2] [4] = { { 1, 2, 3 }, { 4, 5, 6, 7 } };`

Since the first initializer-list has only three values, x [0] [0] is set to 1, x [0] [1] is set to 2, x [0] [2] is set to 3 and the fourth location in the first row is automatically set to 0.

1	2	3	0
4	5	6	7

2. If the number of values specified in any initializer-list is more than colsize of x, compiler reports an error.

Example: `int x [2] [4] = { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9 } };`

colsize is 4, but the number of values listed in the first row is 5. This results in compilation error.

3. Array elements can not be initialized selectively.

4. It is the responsibility of the programmer to ensure that the array bounds do not exceed the row size and colsize of the array. If they exceed, unpredictable results may be produced and even the program can result in system crash sometimes.

5. A 2-d array is used to store a table of values (matrix). Similar to 2-d arrays of int type, we can declare 2-arrays of any other data type supported by C, also.

Example: `float x [4] [5];`

x is declared to be 2-d array of float type with 4 rows and 5 columns

y is declared to be 2-d arrays of double type with 4 rows and 5 columns

Note

A two-dimensional array is used to store a table of values. Two loops (using for loops) are used to access each value in the table, first loop acts as a row selector and second loop acts as a column selector in the table.

Declare, Initialize Array of Char Type

Declaration of Array of char type: A string variable is any valid C variable name and is always declared as an array. The syntax of declaration of a string variable is:

`char string-name[size];`

The size determines the number of character in the string name.

Example: An array of char type to store the above string is to be declared as follows:

<code>char str[8];</code>							
<code>str[0]</code>	<code>str[1]</code>	<code>str[2]</code>	<code>str[3]</code>	<code>str[4]</code>	<code>str[5]</code>	<code>str[6]</code>	<code>str[7]</code>
P	r	o	g	r	a	m	\0

An array of char is also called as a string variable, since it can store a string and it permits us to change its contents. In contrast, a sequence of characters enclosed within a pair of double quotes is called a string constant.

Example: "Program" is a string constant.

Initialization of Arrays of char Type

The syntax of initializing a string variable has two variations:

Variation 1

`char str1 [6] = { 'H', 'e', 'l', 'l', 'o', '\0' };`

Here, str1 is declared to be a string variable with size six. It can store maximum six characters. The initializer – list consists of comma separated character constants. Note that the null character '\0' is clearly listed. This is required in this variation.

Variation 2

`char str2 [6] = { "Hello" };`

Here, str2 is also declared to be a string variable of size six. It can store maximum six characters including null character.

The initializer-list consists of a str[0] str[1] str[2] str[3] str[4] str[5] str[6] str[7]

P r o g r a m \0

string constant. In this variation, null character '\0' will be automatically added to the end of string by the compiler.

In either of these variations, the size of the character array can be skipped, in which case, the size and the number of characters in the initializer-list would be automatically supplied by the compiler.

Example

```
char str1 [ ] = { "Hello" };
```

The size of str1 would be six, five characters plus one for the null character '\0'.

```
char str2 [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The size of str2 would be six, five characters plus one for null character '\0'.

Example 1

Program to sort a list of numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int x [6], n, i, j, tmp;
clrscr( );
printf ("Enter the no. of elements \n");
scanf ("%d", & n);
printf ("Enter %d numbers \n", n);
for (i=0; i<n; i++)
scanf ("%d", & x [i]);
/* sorting begins */
for (i=0; i<n; i++)
for (j=i + 1; j<n; j++)
if (x[i]>x[j])
{
tmp = x [i];
x [i] = x [j];
x [j] = tmp;
}
/* sorting ends */
printf ("sorted list \n");
for (i=0; i<n; i++)
printf ("%d", x [i]);
getch( );
}
```

Input – Output

Enter the no. of elements

5

Enter 5 numbers

10 30 20 50 40

Sorted list

10 20 30 40 50

Example 2 :

C program for addition of two matrices.

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

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

```

#include<process.h>
void main( )
{
int x [5] [5], y [5] [5], z [5] [5], m, n, p, q, i, j;
clrscr ( );
printf ("Enter number of rows and columns of matrix x \n");
scanf ("%d %d", &m, &n);
printf ("Enter number of rows and columns of matrix y \n");
scanf ("%d %d", &p, &q);
if ((m !=p) || (n!=q))
{
printf ("Matrices not compatible for addition \n");
exit(1);
}
printf ("Enter the elements of x \n");
for (i=0; i<m; i++)
for (j=0; j<n; j++)
scanf ("%d", &x [i][j]);
printf ("Enter the elements of x \n");
for (i=0; i<p; i++)
for (j=0; j<n; j++)
scanf ("%d", &y [i] [j]);
/* Summation begins */
for (i=0; j<m; i++)
for (j=0; j<n; j++)
z[i] [j] = x [i] [j] + y [i] [j];
/* summation ends */
printf ("Sum matrix z \n");
for (i=0; i<m; i++)
{
for (j=0; j<n; j++)
printf ("%d", z[i] [j]);
printf ("\n");
}
getch( );
}

```

Example3

Write a program for multiplication of two matrices.

```

#include<stdio.h>
#include<conio.h>
#include<process.h>
void main( )
{
int x [5] [5], y [5] [5], z [5] [5], m, n, p, q, I, j, k;
clrscr ( );
printf ("Enter number of rows and columns of matrix x \n");
scanf ("%d %d", &m, &n);
printf ("Enter number of rows and columns of matrix y \n");
scanf ("%d %d", &p, &q);
if (n!=p)
{
printf ("Matrices not compatible for multiplication \n");
exit (1);
}

```

```

}
printf ("Enter the elements of x \n");
for (i=0; i<m; i++)
for (j=0; j<n; j++)
scanf ("%d", &x [i] [j]);
printf ("Enter the elements of y \n");
for (i=0; i<p; i++)
for (j=0; j<q; j++)
scanf ("%d", &x [i] [j]);
printf ("Enter the elements of y \n");
for (i=0; i<p; i++)
for (j=0; j<q; j++)
scanf ("%d", &y [i] [j]);
/* Multiplication of matrices of x 7 y ends */
for (i=0; i<m; i++)
for (j=0; j<q; j++)
{ z [i] [j] = 0;
for (k=0; k<n; k++)
z [i] [j] += x [i] [k] * y [k] [j];
}
/* Multiplication of matrices of x & y ends */
printf ("Product Matrix z \n");
for (i=0; i<m; i++)
{
for (j=0; j<q; j++)
printf ("%d", z[i] [j]);
printf ("\n");
}
getch( );
}

```

Example 4: C program to print transpose of a matrix.

[The transpose of a matrix is obtained by switching the rows and columns of matrix].

```

#include<stdio.h>
#include<conio.h>
void main ( )
{
int a[3] [3], b[3] [3], i, j;
clrscr ( );
printf ("Enter the elements of the matrix : \n");
for (i=0; i<3; i++)
for (j=0; j<3; j++)
scanf ("%d", &a[i] [j]);
printf ("given matrix is : \n");
for (i=0; i<3; i++)
{
for (j=0; j<3; j++)
printf ("%d", a[i] [j]);
printf ("\n");
}
printf ("transpose of given matrix is : \n");
for (i=0; i<3; i++)
{
for (j=0; j<3; j++)

```

```

{
b [i] [j] = a [j] [i];
printf ("%d", b[i] [j]);
printf ("\n");
}
}
getch();
}

```

Example 5

Write a 'C' program to find the average marks of 'n' students of five subjects for each subject using arrays.

Ans.

```

#include <stdio.h>
void main( )
{
int Sno, S1,S2,S3;
float tot, avg;
char sname[10];
printf("Enter student no;");
scanf("%d", & Sname);
printf("Enter subject - 1, sub - 2, sub - 3 marks;");
scanf("%d %d %d", &s1,&s2,&s3);
tot = S1+S2+S3;
avg = tot/3;
printf("total = %f", tot);
printf("Average = %f", avg);
}

```

Example 6

Write a C program to check the given word is 'Palindrome' or not.

Ans.

```

#include<stdio.h>
#include<conio.h>
void main ( )
{
char str[80], rev[80];
int k, i, j, flag = 0;
clrscr ( );
printf ("Enter any string (max. 80 chars) : \n");
gets (str);
for (i=0; str[i]!='\0'; i++);
for (j=i-1; k=0; j>=0; j--, k++)
rev[k] = str[j];
rev[k] = '\0';
for (i=0; str[i]!='\0'; i++)
{
if (str[i]!=rev[i])
{
flag=1;
break;
}
}
if (flag == 1);
printf ("Given string is not palindrome. \n");
}

```

```

else
printf ("Given string is palindrome. \n");
getch();
}

```

4.5 String Handling Functions in 'C'

To perform manipulations on string data, there are built-in-function (library function) Supported by 'c' compiler. The string functions are.

1. STRLEN (S1)
2. STRCMP (S1, S2)
3. STRCPY (S1, S2)
4. STRCAT (S1, S2)

1. STRLEN (S1): This function is used to return the length of the string name S1,

Ex: S1 = 'MOID'

STRLEN (S1) = 4

2. STRCMP (S1, S2): This is a library function used to perform comparison between two strings. This function returns a value <zero when string S1 is less than S2. The function return a value 0 when S1=S2. Finally the function return a value > 0 when S1>S2.

3. STRCPY (S1, S2): This is a library function used to copy the string S2 to S1.

4. STRCAT (S1, S2): This is a library function used to join two strings one after the other. This function concatenates string S2 at the end of string S1.

Example5 : C program to concatenate the given two strings and print new string.

```

#include<stdio.h>
#include<conio.h>
main ()
{
char s1[10], s2[10], s3[10],
int i,j,k;
printf ("Enter the first string : \n");
scanf ("%s",s1);
printf ("Enter the second string : \n");
scanf ("%s",s2);
i = strlen(s1);
j = strlen(s2);
for (k=0,k<i,k++)
s3[k] = s1[k];
for (k=0,k<j,k++)
s3[i+k] = s2[k];
s3[i+j] = '\0';
printf (" The new string is \n".s3);
}

```


UNIT III

Basic Algorithms: Searching, Basic Sorting Algorithms (Bubble and Selection), Finding roots of Equations. Functions: Functions (including using built in libraries), Parameter passing in functions, call by value. Passing arrays to functions: idea of call by reference.

3.1: Basic Algorithms: An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. We can also view an algorithm as a tool for solving a well-specified computational problem.

For Example: This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the sorting problem:

Input: A sequence of n numbers a_1, a_2, \dots ,

Output: A permutation (reordering) a_1, a_2, \dots , of the input sequence such that $a_1 \leq a_2 \leq \dots \leq a_n$.

Daily use of algorithm:

1. Web search
2. Completing Project work
3. Solving some problems.

3.2: Searching: The process of identifying or finding a particular record is called **Searching**. You often spend time in **searching** for any desired item. If the data is kept properly in sorted order, then **searching** becomes very easy and efficient.

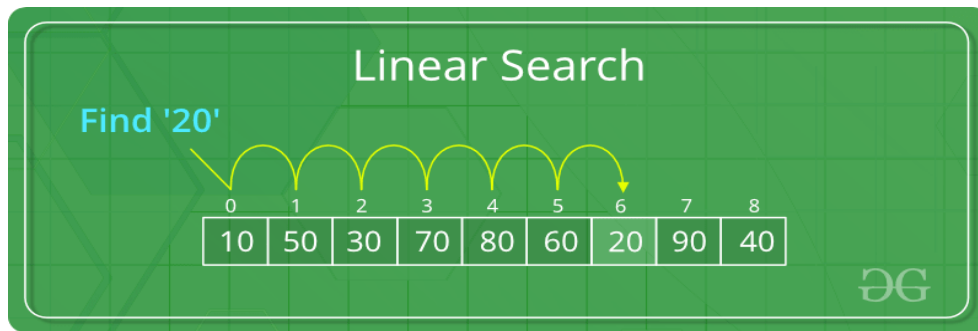
Searching is an operation or a technique that helps find the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

1. Linear Search or Sequential Search

This is the simplest method for searching. In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.



Example:

The list given below is the list of elements in an unsorted array. The array contains ten elements. Suppose the element to be searched is '46', so 46 is compared with all the elements starting from the 0th element, and the searching process ends where 46 is found, or the list ends.

The performance of the linear search can be measured by counting the comparisons done to find out an element. The number of comparison is $O(n)$.

Algorithm for Linear Search or Sequential :

It is a simple algorithm that searches for a specific item inside a list. It operates looping on each element $O(n)$ unless and until a match occurs or the end of the array is reached.

```
int linearsearch(int values[],int target, int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(values[i]==target)
        {
            return(i);
        }
    }
    return(-1);
}
```

Binary Search:

Binary search is a very fast and efficient searching technique. It requires the list to be in sorted order. In this method, to search an element you can compare it with the present element at the center of the list. If it matches, then the search is successful otherwise the list is divided into two halves: one from the 0th element to the middle element which is the center element (first half) another from the center element to the last element (which is the 2nd half) where all values are greater than the center element.

The searching mechanism proceeds from either of the two halves depending upon whether the target element is greater or smaller than the central element. If the element is smaller than the central element, then searching is done in the first half, otherwise searching is done in the second half.

Algorithm for Binary Search

```
bin_search(a,n,x)
{
```

```

low=1;
high=n;
while(low<=high)
do
{
mid[low+high]/2;
if(x<a[mid])then
high=mid-1;
elseif(x>a[mid])then
low=mid+1;
else
return mid;
}
return 0;
}

```

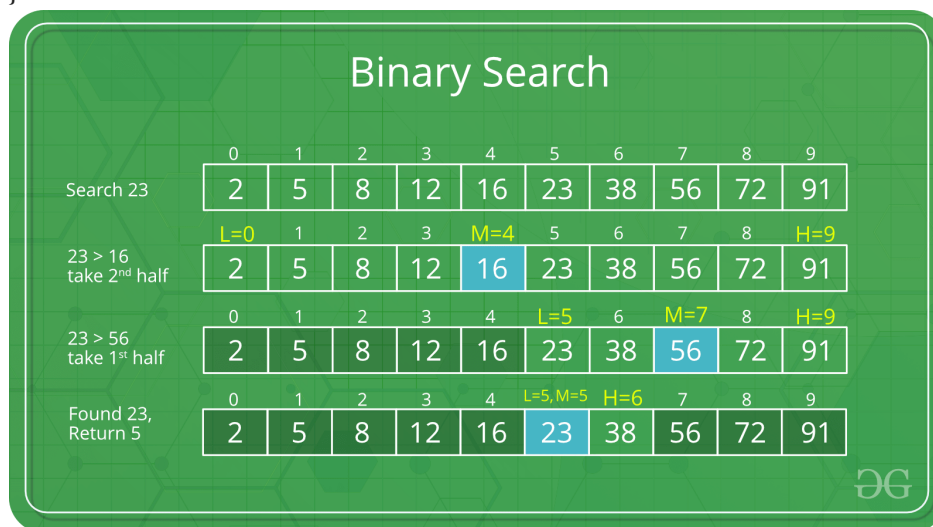


Fig : Binary search

(6. program As per OU lab manual)

Write a C program that uses non recursive function to search for a Key value in a given sorted list of integers using binary search method.

```

#include<stdio.h>
#include<conio.h>
void main()
{
int a[20], i, n, key, low, high, mid;
clrscr();
Printf("Enter the size of the array:");
Scanf("%d",&n);
printf("Enter the array elements in ascending order");
for(i = 0; i < n; i++)

```

```

{
scanf("%d", &a[i]);
}
printf("Enter the key element\n");
scanf("%d", &key);
low = 0;
high = n - 1;
while(high >= low)
{
mid = (low + high) / 2;
if(key == a[mid])
break;
else
{
if(key > a[mid])
low = mid + 1;
else
high = mid - 1;
}
}
if(key == a[mid])
printf("The key element is found at location %d", mid + 1);
else
printf("the key element is not found");
getch();
}

```

Output:

Enter the size of the array :

7

Enter the array elements in ascending order:

23

45

68

90

100

789

Enter the key element:

789

The key Element is found at location: 6

3.3: Basic Sorting Algorithms (Bubble and Selection):

Sorting:

Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search. **Sorting** arranges data in a sequence which makes searching easier.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

1. **Bubble Sort**
2. **Selection Sort**
3. Quick Sort
4. Insertion Sort
5. Merge Sort
6. Heap Sort

1. Bubble Sort:

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists. Bubble sort is not a stable sort which means that if two same elements are there in the list, they may not get their same order with respect to each other.

Step-by-step example:

- Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.
- **First Pass:**

(**5** 1 4 2 8) → (**1** 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5** 4 2 8) → (1 **4** 5 2 8), Swap since 5 > 4

(1 4 **5** 2 8) → (1 4 **2** 5 8), Swap since 5 > 2

(1 4 2 **5** 8) → (1 4 2 **5** 8), Now, since these elements are already in order (8 > 5),

algorithm does not swap them.

Second Pass:

(1 4 2 5 8) \rightarrow (1 4 2 5 8)

(1 4 2 5 8) \rightarrow (1 2 4 5 8), Swap since $4 > 2$

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Algorithm For Bubble Sort:

```
Void bubble sort(int a[],int n)
{
int i,j,temp;          int i,j,temp;
for(i=0;i<n;i++)       for(i=0;i<5;i++)
{                       then goes to the body of loop
for(j=1;j<n;j++)       for(j=1;j<5;j++)
{                       then goes to body of loop
if(a[i]>a[j])           if condition is true then it will be sorted.
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
}
```

II method:

Algorithm For Bubble Sort:

```
procedure bubbleSort( A : list of sortable items ) n = length(A)
repeat
```

```

swapped = false
for i = 1 to n-1 inclusive do
    if A[i-1] > A[i] then
        swap(A[i-1], A[i])
        swapped = true
    end if
end for
n = n - 1
until not swapped
end procedure

```

C Program For Bubble Sort:

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i,n,temp,j,arr[25];
    clrscr();
    printf("Enter the number of elements in the Array: ");
    scanf("%d",&n);
    printf("\nEnter the elements:\n\n");

    for(i=0 ; i<n ; i++)
    {
        printf(" Array[%d] = ",i);
        scanf("%d",&arr[i]);
    }

    for(i=0 ; i<n ; i++)
    {
        for(j=0 ; j<n-i-1 ; j++)
        {
            if(arr[j]>arr[j+1]) //Swapping Condition is Checked
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
    printf("\nThe Sorted Array is:\n\n");
    for(i=0 ; i<n ; i++)
    {
        printf(" %4d",arr[i]);
    }
    getch();
}

```

Output of Program:

```

Enter the number of elements in the Array: 10
Enter the elements:

```

Array[0] = 15
Array[1] = 39
Array[2] = 8
Array[3] = 97
Array[4] = 254
Array[5] = 101
Array[6] = 86
Array[7] = 53
Array[8] = 21
Array[9] = 10

The Sorted Array is:

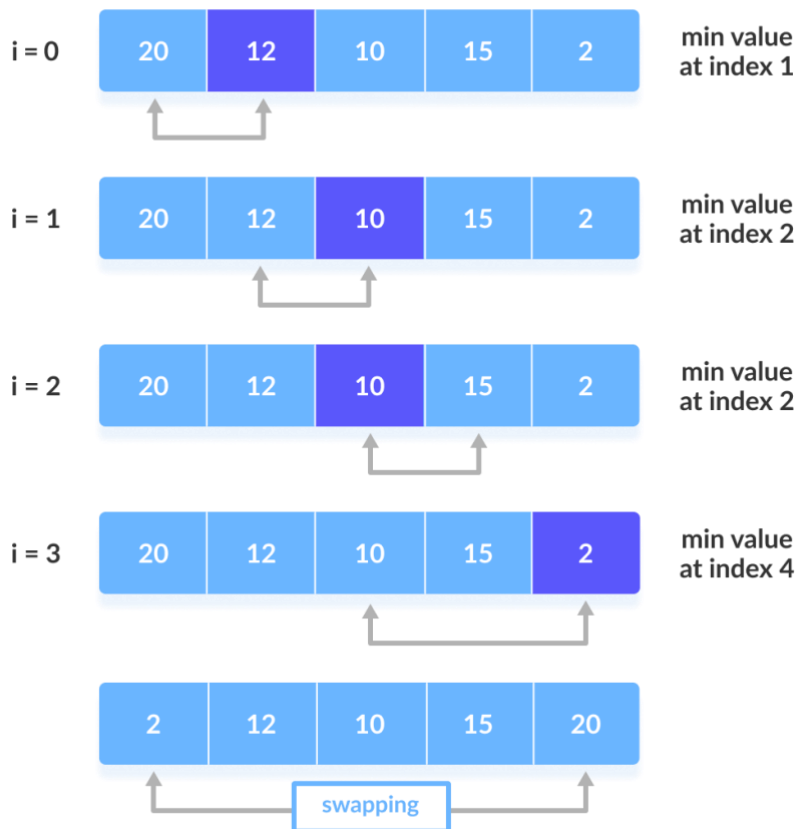
8 10 15 21 39 53 86 97 101 254

Selection Sort : A *Selection Sort* is a Sorting algorithm which finds the smallest element in the array and swaps with the first element then with the second element and continues until the entire array is sorted.

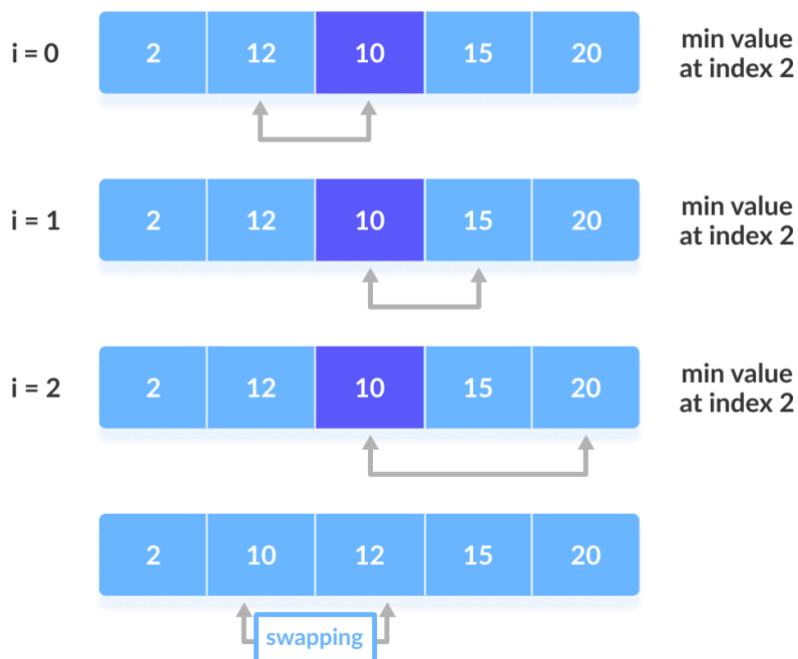
How Selection Sort Works?

At every pass, the smallest element is chosen and swapped with the leftmost unsorted element. Let us analyze the working of the algorithm with the help of the following illustration.

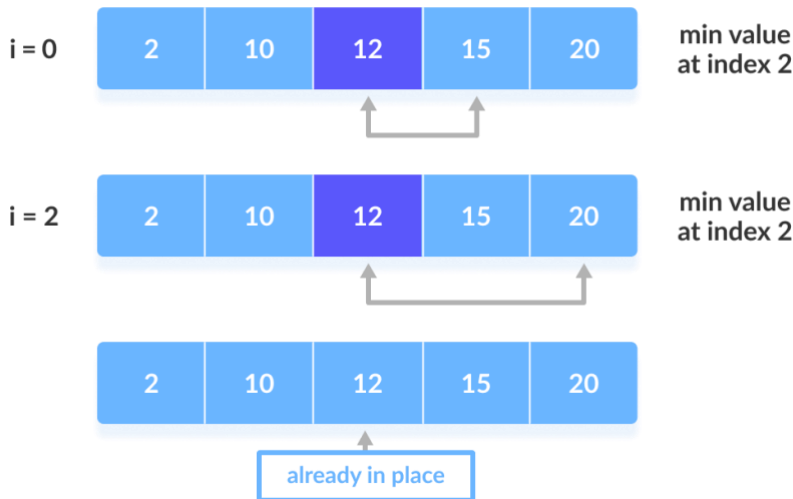
step = 0



step = 1



step = 2



step = 3



Here, size=5.

1. Inside the first loop (before entering the second loop), the leftmost unsorted element is considered as the smallest element. (min=1st element=20)
2. When step=0 (ie. the first pass of the first loop) and $i=\text{step}+1=0+1=1$ (ie. the first pass of the second loop), min and the 2nd element are compared (ie. 20 and 12 are compared)
3. When step=0 and $i=2$, min is compared with the 3rd element (ie. 12 is compared with 10). If the 3rd element is smaller than min then, min is changed to the 3rd element. (ie. min=10). If min is already the smaller one then, nothing is done.
4. When step=0 and $i=3$,
5. When step=0 and $i=4$, min=2.
6. Swap min with the leftmost unsorted element (ie. 20).
7. When step=1 and $i=2$,
8. The process continues till the array is sorted.

Algorithm of Selection Sort:

```
selection_sort(a,n)
```

```
{
```

```
int i,j,min,n,temp;
```

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

```

{
min=1;
for(j=i+1;j<n;j++)
{
if(a[j]<a[min])
{
min=j;
}
}
temp=a[j];
a[i]=a[min];
a[min]=temp;
}
}

```

Write a C program that sorts the given array of integers using selection sort in descending order

```

#include <stdio.h>
void selection_sort();
int a[30], n;
void main()
{
int i;
printf("\nEnter size of an array: ");
scanf("%d", &n);
printf("\nEnter elements of an array:\n");
for(i=0; i<n; i++)
    scanf("%d", &a[i]);
selection_sort();
printf("\n\n Descending order:\n");
for(i=0; i<n; i++)
    printf("\n%d", a[i]);
getch();
}
void selection_sort()
{
int i, j, min, temp;

```

```

for (i=0; i<n; i++)
{
    min = i;
    for (j=i+1; j<n; j++)
    {
        if (a[j] < a[min])
            min = j;
    }
    temp = a[i];
    a[i] = a[min];
    a[min] = temp;
}
}

```

OUTPUT:

Enter size of an array: 8

Enter elements of an array:

68 45 78 14 25 65 55 44

Descending order:

14

25

44

45

55

65

68

78

1.1 Finding Roots of a Quadratic Equation.

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

```
#include<math.h>
```

```
void main()
```

```
{
```

```
float a,b,c,r1,r2,d;
```

```
clrscr();
```

```
printf("Enter the values for equation:");
```

```

scanf("%f%f%f",&a,&b,&c);
/* check the condition */
if(a==0)
printf("Enter value should not be zero ");
else {
    d=b*b-4*a*c;
    /* check the condition */
    if(d>0)

    {
        r1=(-b+sqrt(d)/(2*a));
        r2=(-b-sqrt(d)/(2*a));
        printf("roots are real and unequal\n");
        printf("%f\n%f\n",r1,r2);
    }
    else
    if(d==0)

    {
        r1=-b/(2*a);
        r2=-b/(2*a);
        printf("roots are real and equal\n");
        printf("root=%f\n",r1);
        printf("root=%f\n",r2);
    }
    else
    printf("roots are imaginary");
}
getch();
}

```

Output:

1. Enter the values for equation: 1, 6, 9

Roots are real and equal

Root= -3.0000

Root= -3.0000

2. Enter the values for equation: 2, 7, 6

Roots are real and unequal

Root= -6.75

Root= -7.25

3. Enter the values for equation: 1, 2, 3

Roots are imaginary

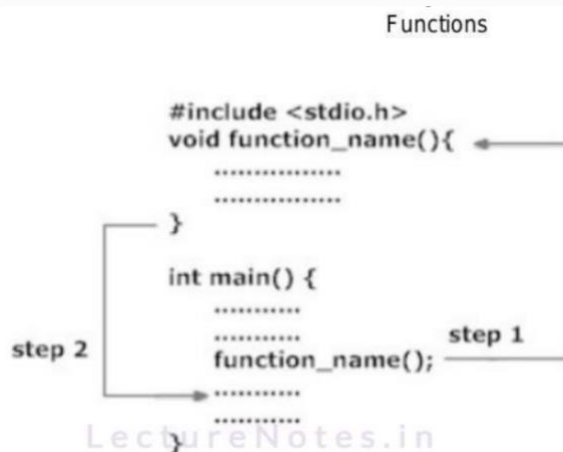
3.5: Functions:

A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by “{ }” which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

2. USES OF C FUNCTIONS:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

3.C FUNCTION DECLARATION, FUNCTION CALL AND FUNCTION DEFINITION:



There are 3 aspects in each C function. They are,

- Function declaration or prototype – This informs compiler about the function name, function parameters and return value's data type.
- Function call – This calls the actual function
- Function definition – This contains all the statements to be executed.

C functions aspects	Syntax
function definition	return_type function_name (arguments list) { Body of function; }
function call	function_name (arguments list);
function declaration	return_type function_name (argument list);

SIMPLE EXAMPLE PROGRAM FOR C FUNCTION:

- As you know, functions should be declared and defined before calling in a C program.
- In the below program, function “square” is called from main function.
- The value of “m” is passed as argument to the function “square”. This value is multiplied by itself in this function and multiplied value “p” is returned to main function from function “square”.

```
#include<stdio.h>
// function prototype, also called function declaration
float square ( float x );
// main function, program starts from here//
int main( )
{
    float m, n ;
    printf ( "\nEnter some number for finding square \n");
    scanf ( "%f", &m );
    // function call
    n = square ( m );
    printf ( "\nSquare of the given number %f is %f",m,n );
}

float square ( float x ) // function definition
{
    float p ;
    p = x * x ;
    return ( p );
}
```

OUTPUT:

```
Enter some number for finding square
2
Square of the given number 2.000000 is 4.000000
```

4. HOW TO CALL C FUNCTIONS IN A PROGRAM?

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

1. CALL BY VALUE:

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter cannot be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY VALUE):

- In this program, the values of the variables “m” and “n” are passed to the function “swap”.
- These values are copied to formal parameters “a” and “b” in swap function and used.

C

```

#include<stdio.h>
// function prototype, also called function declaration
void swap(int a, int b);

int main()
{
    int m = 22, n = 44;
    // calling swap function by value
    printf(" values before swap  m = %d \nand n = %d", m, n);
    swap(m, n);
}

void swap(int a, int b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
    printf(" \nvalues after swap m = %d\n and n = %d", a, b);
}

```

OUTPUT:

```

values before swap m = 22
and n = 44
values after swap m = 44
and n = 22

```

2. CALL BY REFERENCE:

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY REFERENCE):

- In this program, the address of the variables “m” and “n” are passed to the function “swap”.
- These values are not copied to formal parameters “a” and “b” in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.

```

#include<stdio.h>
// function prototype, also called function declaration
void swap(int *a, int *b);

int main()
{
    int m = 22, n = 44;
    // calling swap function by reference
    printf("values before swap m = %d \n and n = %d",m,n);
    swap(&m, &n);
}

```



```

}
void swap(int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
    printf("\n values after swap a = %d \nand b = %d", *a, *b);
}

```

OUTPUT:

values before swap m = 22 and n = 44
values after swap a = 44 and b = 22

Built in function:

1. C standard library functions are built in functions in C Programming.
2. A collection of reusable functions(code for building data structures, code for performing math functions and scientific calculations etc.)
3. Which will save C programmers time especially when working on large programming projects.
4. The C library is a part of ANSI (American National Standard Institute) for the C language.
5. The C programs can call on a large number of functions from the standard C library.
6. Function prototype and data definitions of these functions are written in their respective header file.
7. For example if you want to use printf() function, the header file <stdio.h> should be included.
8. If you want to find square root by just using sqrt() function which is defined under header file <math.h>

Listed below are the few built-in Library function.

1. string.h – Supports string function.
2. Stdlib.h – supports miscellaneous declarations.
3. Math.h – Supports definitions used for mathematical calculations.
4. Time.h – supports system time functions.
5. Ctype.h – support to handle characters.

stdio.h functions-

fclose()	Closes a stream
fcloseall()	Closes all open streams
feof()	Tests if end-of-file has been reached on a stream
fflush()	Flushes a stream
fgetc()	Gets a character from a stream
fgetpos()	Gets the current file pointer position
fsetpos()	Positions the file pointer of a stream
fgetchar()	Gets a character from stdin
fgets()	Gets a string from a stream
fopen()	Opens a stream
fprintf()	Sends formatted output to stream
fputc()	Outputs a character to a stream
fputs()	Outputs a string to a stream
fread()	Reads data from a stream
fscanf()	Scans and formats input from a stream.
fseek()	Sets the file pointer to a particular position.
ftell()	Returns the current position of the file pointer.
fwrite()	Writes to a stream.
getc()	gets one character.
getchar()	gets a character from stdin.
gets()	Get a string from stdin.
getw()	gets an integer from stream.
printf()	Sends the formatted output to stdout.
putc()	Outputs a character to stdout.
putchar()	Outputs a character on stdout.
puts()	Outputs string and appends a newline character.
putw()	Outputs an integer on a stream
remove()	Removes a file
rename()	Renames a file
Rewind()	Brings the file pointer to stream's beginning
scanf()	Scans and formats input from stdin.

conio.h

clrscr()	Clears text mode window
getch()	gets a character from console but does not echo to the screen
getche()	gets a character from console, and echoes to the screen
putch()	Outputs character to the text window on the screen
cgets()	Reads string from console
getchar()	Inputs a character from stdin.

stdlib.h

itoa()	converts an integer to a string.
atoi()	Converts string of digits to integer.
Random()	Returns a random number between 0 and number - 1
randomize()	Initializes random number generator.
exit()	Terminates the program.
min()	Returns the smallest of two numbers.
max()	Returns the largest of two numbers.
ltoa()	converts a long to a string
ultoa()	converts an unsigned long to a string
atof()	converts a string to a floating point
_atold()	converts a string to a long double

math.h

abs()	gets the absolute value of an integer
acos()	Calculates the inverse of cos Accepts the angle value in radians
asin()	Calculates the inverse of sin Accepts the angle value in radians
atan()	Calculates the inverse of tan Accepts the angle value in radians
ceil()	Returns the largest integer in given list.
cos()	Calculates the cosine Accepts the angle value in radians

string.h

string.h

strcat()	Function to concatenate(merge) strings.
strcmp()	Function to compare two strings.
strcpy()	Function to copy a string to another string
stricmp()	Function to compare two strings ignoring their case.
strlen()	Function to calculate the length of the string
strlwr()	Converts the given string to lowercase
strrev()	Function to reverse the given string.
strupr()	Converts the given string to uppercase
strdup()	Duplicates a string.
strnicmp()	Compares the first n characters of one string to another without being case sensitive.
strncat()	Adds the first n characters at the end of second string.
strncpy()	Copies the first n characters of a string into another.
strchr()	Finds the first occurrence of the character.
strrchr()	Finds the last occurrence of the character.
strstr()	Finds the first occurrence of string in another string.
strset()	Sets all the characters of the string to a given character.
strnset()	Sets first n characters of the string to a given character.

Parameter Passing in C

When a function gets executed in the program, the execution control is transferred from calling-function to called function and executes function definition, and finally comes back to the calling function. When the execution control is transferred from calling-function to called-function it may carry one or number of data values. These data values are called as **parameters**.

Parameters are the data values that are passed from calling function to called function.

In C, there are two types of parameters and they are as follows...

- **Actual Parameters**
- **Formal Parameters**

The **actual parameters** are the parameters that are specified in calling function. The **formal parameters** are the parameters that are declared at called function. When a function gets executed, the copy of actual parameter values are copied into formal parameters.

In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

- **Call by Value**
- **Call by Reference**

Call by Value

In **call by value** parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. **The changes made on the formal parameters does not effect the values of actual parameters.** That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example consider the following program...

Example Program

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

void main()
{
    int num1, num2 ;
    void swap(int,int) ; // function declaration
    clrscr() ;
    num1 = 10 ;
    num2 = 20 ;
    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
    swap(num1, num2) ; // calling function
    printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
    getch() ;
}

void swap(int a, int b) // called function
{
    int temp ;
    temp = a ;
    a = b ;
    b = temp ;
}
```

Output:

```
"C:\Users\User\Desktop\New folder\parameter_passing\bin\Debug\parameter_passing.exe"
Before swap: num1 = 10, num2 = 20
After swap: num1 = 10
num2 = 20
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

In the above example program, the variables **num1** and **num2** are called actual parameters and the variables **a** and **b** are called formal parameters. The value of **num1** is copied into **a** and the value of **num2** is copied into **b**. The changes made on variables **a** and **b** does not effect the values of **num1** and **num2**.

Passing Array to Function in C

Array Definition :

Array is collection of elements of similar data types .

Passing array to function :

Array can be passed to function by two ways :

1. **Pass Entire array**
2. **Pass Array element by element**

Let us discuss both of them one by one :

1 . Pass Entire array

- Here entire array can be passed as a argument to function .
- Function gets **complete access** to the original array .
- While passing entire array Address of first element is passed to function , any changes made inside function , directly **affects the Original value** .
- Function Passing method : **“Pass by Address”**

2 . Pass Array element by element

- Here individual elements are passed to function as argument.
- Duplicate **carbon copy of Original variable** is passed to function .
- So any changes made inside function **does not affects the original value**.
- Function doesn't get complete access to the original array element.
- Function passing method is **“Pass by Value”** array can also be passed to a function as an argument . In this guide, we will learn how to pass the array to a function using call by value and call by reference methods.

Passing array to function using call by value method

As we already know in this type of function call, the actual parameter is copied to the formal parameters.

```
#include <stdio.h>
void disp( char ch)
{
    printf("%c ", ch);
}
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
    for (int x=0; x<10; x++)
```

```

{
    /* I'm passing each element one by one using subscript*/
    disp (arr[x]);
}

return 0;
}

```

Output:

a b c d e f g h i j

Passing array to function using call by reference

When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

```

#include <stdio.h>
void disp( int *num)
{
    printf("%d ", *num);
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    for (int i=0; i<10; i++)
    {
        /* Passing addresses of array elements*/
        disp (&arr[i]);
    }

    return 0;
}

```

Output:

1 2 3 4 5 6 7 8 9 0

How to pass an entire array to a function as an argument?

In the above example, we have passed the address of each array element one by one using a for loop in C. However you can also pass an entire array to a function like this:

Note: The array name itself is the address of first element of that array. For example if array name is arr then you can say that **arr** is equivalent to the **&arr[0]**.

```

#include <stdio.h>
void myfuncn( int *var1, int var2)
{
    /* The pointer var1 is pointing to the first element of
    * the array and the var2 is the size of the array. In the
    * loop we are incrementing pointer so that it points to
    * the next element of the array on each increment.
    */
    for(int x=0; x<var2; x++)
    {
        printf("Value of var_arr[%d] is: %d \n", x, *var1);
        /*increment pointer for next element fetch*/
        var1++;
    }
}

```

```
}  
  
int main()  
{  
    int var_arr[] = {11, 22, 33, 44, 55, 66, 77};  
    myfuncn(var_arr, 7);  
    return 0;  
}
```

Output:

Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77

Idea Of Call by Reference:

We often write large programs and it is a good idea to split large steps into smaller procedures. These small procedure blocks are known as functions. Functions are often required to do repetitive jobs. We can define a function and call them from anywhere we need. This is a good choice for code reuse and code size optimization.

Unit - IV

Recursion: Recursion, as a different way of solving problems. Example programs, such as Finding Factorial, Fibonacci series. **Structure:** Structures, Defining structures and Array of Structures

Recursion:

One of the special features of C language is its support to recursion. Very few computer languages will support this feature.

Recursion can be defines as the process of a function by which it can call itself. The function which calls itself again and again either directly or indirectly is known as recursive function.

The normal function is usually called by the main () function, by means of its name. But, the recursive function will be called by itself depending on the condition satisfaction.

For Example,

```
main ( )
{
f1( ) ; ——— Function called by main
_____
_____
}
f1( ) ; ——— Function definition
{
_____
_____
_____
f1( ) ; ——— Function called by itself
}
```

In the above, the main () function s calling a function named f1() by invoking it with its name. But, inside the function definition f1(), there is another invoking of function and it is the function f1() again.

Example programs on Recursion

Example 1 : Write a program to find the factorial of given non-negative integer using recursive function.

```
#include<stdio.h>
main ( )
{
int result, n;
printf( “ Enter any non-negative integer\n”);
scanf ( “ %d”, & n);
result = fact(n);
printf ( “ The factorial of %d is %d \n”, n, result);
}
fact( n )
int n;
{
int i ;
i = 1;
if ( i == 1) return ( i);
else
{
i = i * fact ( n - 1);
return ( i );
}
}
```

Example 2: Write ‘C’ program to generate Fibonacci series up to a limit

```

using recursion function. .
#include<stdio.h>
#include<conio.h>
int Fibonacci (int);
void main ( )
{
int i, n;
clrscr ( );
printf ("Enter no. of Elements to be generated" \n)
scanf ("%d", &n);
for (i=1; i<n; i++)
printf ("%d", Fibonacci (i));
getch( );
}
int Fibonacci (int n)
{
int fno;
if (n==1)
return 1;
else
if (n==2);
return 1;
else
fno=Fibonacci (n-1) + Fibonacci (n-2);
return fno;
}

```

Definition of Structure

A group of one or more variables of different data types organized together under a single name is called **Structure**.

Or

A collection of heterogeneous (dissimilar) types of data grouped together under a single name is called a **Structure**.

A structure can be defined to be a group of logically related data items, which may be of different types, stored in contiguous memory locations, sharing a common name, but distinguished by its members.

Hence a structure can be viewed as a heterogeneous user-defined data type. It can be used to create variables, which can be manipulated in the same way as variables of built-in data types. It helps better organization and management of data in a program.

When a structure is defines the entire group s referenced through the structure name. The individual components present in the structure are called structure members and those can be accessed and processed separately.

Structure Declaration

The declaration of a structure specifies the grouping of various data items into a single unit without assigning any resources to them. The syntax for declaring a structure in C is as follows:

```

struct Structure Name
{
Data Type member-1;
Data Type member-2;
.... ....
DataType member-n;
};

```


The structure declaration starts with the structure header, which consists of the keyword '**struct**' followed by a tag. The tag serves as a structure name, which can be used for creating structure variables. The individual members of the structure are enclosed between the curly braces and they can be of the similar or dissimilar data types. The data type of each variable is specified in the individual member declarations.

Example:

Let us consider an employee database consisting of employee number, name, and salary. A structure declaration to hold this information is shown below:

```
struct employee
{
int eno;
char name [80];
float sal;
};
```

The data items enclosed between curly braces in the above structure declaration are called structure elements or structure members.

Employee is the name of the structure and is called structure tag. Note that, some members of employee structure are integer type and some are character array type.

The individual members of a structure can be variables of built – in data types (int, char, float etc.), pointers, arrays, or even other structures. All member names within a particular structure must be different. However, member names may be the same as those of variables declared outside the structure. The individual members cannot be initialized inside the structure declaration.

Note

Normally, structure declarations appear at the beginning of the program file, before any variables or functions are declared.

They may also appear before the main (), along with macro definitions, such as #define.

In such cases, the declaration is global and can be used by other functions as well.

Structure Variables

Similar to other types of variables, the structure data type variables can be declared using structure definition.

```
struct
{
int rollno;
char name[20];
float average;
a, b;
}
```

In the above structure definition, a and b are said to be structure type variables. 'a' is a structure type variable containing rollno, name average as members, which are of different data types. Similarly 'b' is also a structure type variable with the same members of 'a'.

Structure Initialization

The members of the structure can be initialized like other variables. This can be done at the time of declaration.

Example 1

```
struct
{
int day;
int month;
int year;
}
date = { 25,06,2012};
i.e
date. day = 25
```

```
date. month = 06
date. year = 2012
```

Example 2

```
struct address
```

```
{
char name [20];
char design [10];
char place [10];
} ;
```

i.e

```
struct address my-add = { 'Sree', 'AKM', 'RREDDY' };
```

i.e

```
my-add . name = 'Sree'
```

```
my-add . design = AKM
```

```
my-add . place = RREDDY
```

As seen above, the initial values for structure members must be enclosed with in a pair of curly braces. The values to be assigned to members must be placed in the same order as they are specified in structure definition, separated by commas. If some of the members of the structure are not initialized, then the c compiler automatically assigns a value 'zero' to them.

Accessing of Structure Members

As seen earlier, the structure can be individually identified using the period operator (.). After identification, we can access them by means of assigning some values to them as well as obtaining the stored values in structure members.

The following program illustrates the accessing of the structure members.

Example: Write a C program, using structure definition to accept the time and display it.

```
/* Program to accept time and display it */
# include <stdio.h>
main( )
{
struct
{
int hour, min;
float seconds;
} time;
printf ( "Enter time in Hours, min and Seconds\n");
scanf ( " %d %d %f", &time . hour, & time . min, & time . seconds);
printf ( " The accepted time is %d %d %f " , time . hour, time . min, time. seconds "");
}
```

Nested Structures

The structure is going to certain number of elements /members of different data types. If the members of a structure are of structure data type, it can be termed as structure with structure or nested structure.

Example

```
struct
{
int rollno;
char name[20];
float avgmarks;
struct
{
int day, mon, year;
} dob'
```

```
} student;
```

In the above declaration, student is a variable of structure type consisting of the members namely rollno, name, avgmarks and the structure variable dob.

The dob structure is within another structure **student** and thus structure is nested. In this type of definitions, the elements of the require structure can be referenced by specifying appropriate qualifications to it, using the period operator (.) .

For example, **student.dob.day** refers to the element day of the inner structure dob.

Structures and Arrays

Array is group of identical stored in consecutive memory locations with a single / common variable name. This concept can be used in connection with the structure in the following ways.

a. Array of structures

b. structures containing arrays (or) arrays within a structure

c. Arrays of structures contain arrays.

Array of Structures

Student details in a class can be stored using structure data type and the student details of entire class can be seen as an array of structure.

Example

```
struct student
```

```
{  
int rollno;  
int year;  
int tmarks;  
}
```

```
struct student class[40];
```

In the above class [40] is structure variable accommodating a structure type student up to 40.

The above type of array of structure can be initialized as under

```
struct student class [2] = { 001,2011,786},{ 002, 2012, 710}};
```

i.e class[0] . rollno = 001

class[0] . year = 2011

class[0] . tmarks = 777 and

class[1] . rollno = 002

class[1] . year = 2012

class[1] . tmarks = 777 .

Structures containing Arrays

A structure data type can hold an array type variable as its member or members. We can declare the member of a structure as array data type similar to int, float or char.

Example

```
struct employee
```

```
{  
char ename [20];  
int eno;  
};
```

In above, the structure variable employee contains character array type ename as its member. The initialization of this type can be done as usual.

```
struct employee = { ' Rajashekar', 7777};
```

Arrays of Structures Contain Arrays

Arrays of structures can be defined and in that type of structure variables of array type can be used as members.

Example

```
struct rk
```

```
{  
int empno;
```

```
char ename[20];
flat salary;
} mark[50];
```

In the above, mark is an array of 50 elements and such element in the array is of structure type rk. The structure type rk, in turn contains ename as array type which is a member of the structure. Thus mark is an array of structures and these structures in turn holds character names in array ename.

The initialization of the above type can be done as:

```
{
7777, ' Prasad' , 56800.00}
};
i.e mark[0] . empno = 7777;
mark[0] . ename = 'Prasad';
mark[0] . salary = 56800.00
```

Program

Write a C program to accept the student name, rollno, average marks present in the class of student and to print the name of students whose average marks are greater than 40 by using structure concept with arrays.

```
# include <stdio.h>
main( )
{
int i, n,
struct
{
char name [20];
int rollno;
float avgmarks;
}
class [40];
printf (" Enter the no. of students in the class\n");
scanf ( " %d", & n );
for ( i = 0, i < n, i++)
{
print ( " Enter students name, rollno, avgmarks\n");
scanf ( " %s %d", &class[i].name, &class[i].rollno, &class[i].avgmarks)
}
printf (" The name of the students whose average");
printf ( " marks is greater than 40 \n");
for ( i = 0, i < n, i++)
if ( class[i].avgmarks > 40)
printf (" %s", class[i].name);
}
```

Advantages of Structure Type over Array Type Variables

1. Using structures, we can group items of different types within a single entity, which is not possible with arrays, as array stores similar elements.
2. The position of a particular structure type variable within a group is not needed in order to access it, whereas the position of an array member in the group is required, in order to refer to it.
3. In order to store the data about a particular entity such as a 'Book', using an array type, we need three arrays, one for storing the 'name', another for storing the 'price' and a third one for storing the 'number of pages' etc., hence, the overhead is high. This overhead can be reduced by using structure type variable.
4. Once a new structure has been defined, one or more variables can be declared to be of that type.
5. A structure type variable can be used as a normal variable for accepting the user's input, for displaying the output etc.,

6. The assignment of one 'struct' variable to another, reduces the burden of the programmer in filling the variable's fields again and again.
7. It is possible to initialize some or all fields of a structure variable at once, when it is declared.
8. Structure type allows the efficient insertion and deletion of elements but arrays cause the inefficiency.
9. For random array accessing, large hash tables are needed. Hence, large storage space and costs are required.
10. When structure variable is created, all of the member variables are created automatically and are grouped under the given variable's name.

Unit - V

Pointers - Idea of pointers, Defining pointers, Use of Pointers in self-referential structures, notion of linked list (no implementation), **Introduction to File Handling**.

Idea of pointer:

- Pointer is Special Variable used to Reference and de-reference memory.

Pointers:

Pointer have a name, just like other variables. Pointers are a powerful tool and should be used carefully. Whenever you need to use it, you have to define them first. In simple words, a pointer is an address. It is a derived data type that stores the memory address. A pointer can also be used to refer another pointer, function. A pointer can be incremented/ decremented, i.e., to point to the next/ previous memory location. Syntax: data_type * pointer_variable_name;

How does Pointer Work?

If we declare a variable v of type int, v will actually store a value.

```
int v = 0;
```

v is equal to zero now. However, each variable, apart from value, also has its address (or, simply put, where it is located in the memory). The address can be retrieved by putting an ampersand (&) before the variable name.

```
&v
```

If you print the address of a variable on the screen, it will look like a totally random number (moreover, it can be different from run to run).

```
#include <stdio.h>

int main() {
    int v = 0;
    printf("%d\n", &v);
    return 0;
}
```

The output of this program is 2001.

Now, what is a pointer? Instead of storing a value, a pointer will store the address of a variable.

Self Referential Structures

Structures can have members which are of the type the same structure itself in which they are included, This is possible with pointers and the phenomenon is called as self referential structures.

A self referential structure is a structure which includes a member as pointer to the present structure type.

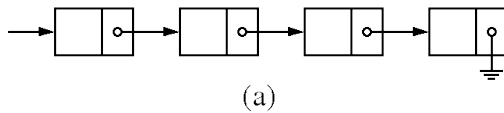
The general format of self referential structure is

struct parent

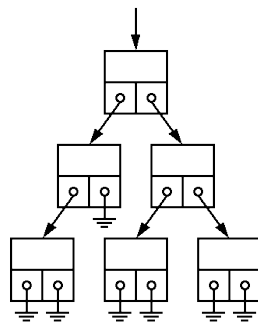
```
{
    memeber1;
    memeber2;
    _____;
    _____;
```

```
struct parent *name;
```

};



(a)



(b)

The structure of type parent is contains a member, which is pointing to another structure of the same type i.e. parent type and name refers to the name of the pointer variable.

Here, name is a pointer which points to a structure type and is also an element of the same structure.

Example

struct element

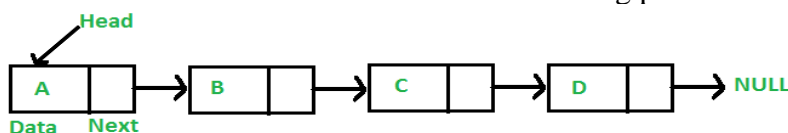
```
{
char name{20};
int num;
struct element * value;
}
```

Element is of structure type variable. This structure contains three members

- a 20 elements character array called **name**
 - An integer element called **num**
 - a pointer to another structure which is same type called **value**. Hence it is self referential structure.
- These structure are mainly used in applications where there is need to arrange data in ordered manner. A self referential structure is used to create data structures like linked lists, stacks, etc.

Notation to linked list:

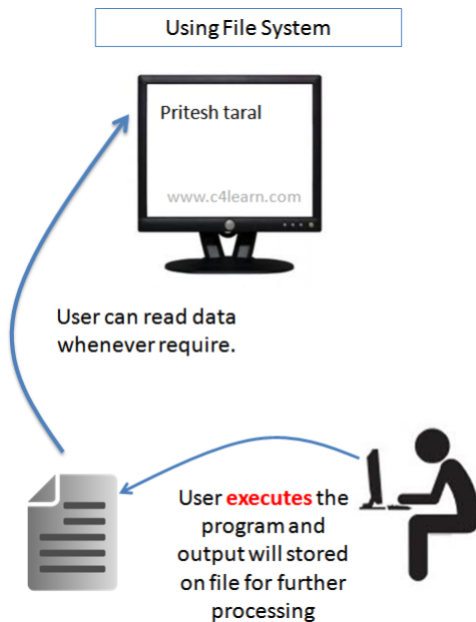
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

File Handling:

1. New way of dealing with **data is file handling**.
2. Data is **stored onto the disk** and can be retrieve whenever require.
3. Output of the program may be stored onto the disk
4. In C we have many **functions that deals with file handling**
5. A file is a collection of bytes stored on a **secondary storage device** (generally a disk)
6. Collection of byte may be **interpreted as** –
 - o Single character
 - o Single Word
 - o Single Line
 - o Complete Structure.



File: the file is a permanent storage medium in which we can store the data permanently.

Types of file can be handled

we can handle three type of file as

- (1) sequential file
- (2) random access file
- (3) binary file

File Operation

opening a file:

Before performing any type of operation, a file must be opened and for this fopen() function is used.

syntax:

```
file-pointer=fopen("FILE NAME ","Mode of open");
```

example:

```
FILE *fp=fopen("ar.c","r");
```

If fopen() unable to open a file than it will return NULL to the file pointer.

File-pointer: The file pointer is a pointer variable which can be store the address of a special file that means it is based upon the file pointer a file gets opened.

Declaration of a file pointer:-

```
FILE* var;
```

Modes of open

The file can be open in three different ways as

Read mode 'r'/rt

Write mode 'w'/wt

Appened Mode 'a'/at

Reading a character from a file

getc() is used to read a character into a file

Syntax:

```
character_variable=getc(file_ptr);
```

Writing a character into a file

putc() is used to write a character into a file

```
puts(character-var,file_ptr);
```

CLOSING A FILE

fclose() function close a file.

fclose(file-ptr);

fcloseall () is used to close all the opened file at a time

File Operation

1. Generally we used to open **following types of file in C** –

File Type	Extension
C Source File	.c
Text File	.txt
Data File	.dat

The following file operation carried out the file

(1)creation of a new file

(3)writing a file

(4)closing a file

Before performing any type of operation we must have to open the file.c, language communicate with file using A new type called **file pointer**.

Operation with fopen()

File pointer=fopen(“FILE NAME”,”mode of open”);

If **fopen()** unable to open a file then it will return **NULL** to the file-pointer.

Reading and writing a characters from/to a file

fgetc() is used for reading a character from the file

Syntax:

character variable= fgetc(file pointer);

fputc() is used to writing a character to a file

Syntax:

fputc(character,file_pointer);

/*Program to copy a file to another*/

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

```
void main()
```

```
{
```

```
FILE *fs,*fd;
```

```
char ch;
```

```
If(fs=fopen(“scr.txt”,”r”)==0)
```

```
{
```

```
printf(“sorry....The source file cannot be opened”);
```

```
return;
```

```
}
```

```
If(fd=fopen(“dest.txt”,”w”)==0)
```

```
{
```

```
printf(“Sorry.....The destination file cannot be opened”);
```

```
fclose(fs);
```

```
return;
```

```
}
```

```
while(ch=fgets(fs)!=EOF)
```

```
fputc(ch,fd);
```

```
fcloseall();
```

```
}
```

Reading and writing a string from/to a file

getw() is used for reading a string from the file

Syntax:

gets(file pointer);

putw() is used to writing a character to a file

Syntax:

fputs(integer,file_pointer);

#include<stdio.h>

#include<stdlib.h>

void main()

{

FILE *fp;

int word;

/*place the word in a file*/

fp=fopen("dgt.txt","wb");

If(fp==NULL)

{

printf("Error opening file");

exit(1);

}

word=94;

putw(word,fp);

If(ferror(fp))

printf("Error writing to file\n");

else

printf("Successful write\n");

fclose(fp);

/*reopen the file*/

fp=fopen("dgt.txt","rb");

If(fp==NULL)

{

printf("Error opening file");

exit(1);

}

/*extract the word*/

word=getw(fp);

If(ferror(fp))

printf("Error reading file\n");

else

printf("Successful read:word=%d\n",word);

/*clean up*/

fclose(fp);

}

Reading and writing a string from/to a file

fgets() is used for reading a string from the file

Syntax:

fgets()(string, length, file pointer);

fputs() is used to writing a character to a file

Syntax:

fputs()(string,file_pointer);

#include<string.h>

#include<stdio.h>

void main(void)

{

FILE*stream;

```
char string[]="This is a test";
char msg[20];
/*open a file for update*/
stream=fopen("DUMMY.FIL","w+");
/*write a string into the file*/
fwrite(string,strlen(string),1,stream);
/*seek to the start of the file*/
fseek(stream,0,SEEK_SET);
126 *Under revision
/*read a string from the file*/
fgets(msg,strlen(string)+1,stream);
/*display the string*/
printf("%s",msg);
fclose(stream);
}
```

stdio.h functions-

fclose()	Closes a stream
fcloseall()	Closes all open streams
feof()	Tests if end-of-file has been reached on a stream
fflush()	Flushes a stream
fgetc()	Gets a character from a stream
fgetpos()	Gets the current file pointer position
fsetpos()	Positions the file pointer of a stream
fgetchar()	Gets a character from stdin
fgets()	Gets a string from a stream
fopen()	Opens a stream
fprintf()	Sends formatted output to stream
fputc()	Outputs a character to a stream
fputs()	Outputs a string to a stream
fread()	Reads data from a stream
fscanf()	Scans and formats input from a stream.
fseek()	Sets the file pointer to a particular position.
ftell()	Returns the current position of the file pointer.
fwrite()	Writes to a stream.
getc()	gets one character.
getchar()	gets a character from stdin.
gets()	Get a string from stdin.
getw()	gets an integer from stream.
printf()	Sends the formatted output to stdout.
putc()	Outputs a character to stdout.
putchar()	Outputs a character on stdout.
puts()	Outputs string and appends a newline character.
putw()	Outputs an integer on a stream
remove()	Removes a file
rename()	Renames a file
Rewind()	Brings the file pointer to stream's beginning
scanf().	Scans and formats input from stdin.