LECTURE NOTES

ON

INTRODUCTION TO PROGRAMMING

(Common to All Branches of Engineering)

2023 – 2024

B.Tech I-Year I-Semester

(Autonomous-AK23)

ANNAMACHARYA INSTITUTE OF TECHNOLOGY & SCIENCES, TIRUPATI

(Autonomous)

Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapuramu.

Two B.Tech Programmes (CSE & ECE) are accredited by NBA, New Delhi.

INTRODUCTION TO PROGRAMMING
(Common to All Branches of Engineering)

<u>UNIT 1</u>

Introduction to Programming and Problem Solving

History of Computers, Basic organization of a computer: ALU, input-output units, memory, program counter, Introduction to Programming Languages, Basics of a Computer Program-Algorithms, flowcharts (using Dia Tool), pseudo code. Introduction to compilation and Execution, Primitive Data Types, Variables and Constants, Basic Input and Output, Operations, Type conversion and Casting.

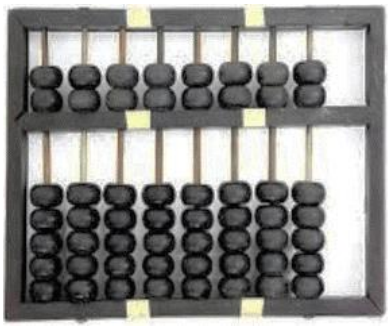
Problem solving techniques: Algorithmic approach, characteristics of algorithm, Problem solving strategies: Top-down approach, Bottom-up approach, Time and space complexities of algorithms.

History of Computers (or) Evolution of Computers:

The history of computer begins with the birth of abacus which is believed to be the first computer. It is said that Chinese invented Abacus around 4,000 years ago. let us consider the development of a computer through various stages.

1. Abacus:

* It was a wooden rack which has metal rods with beads mounted on them.
* The beads were moved by the abacus operator according to some rules to perform arithmetic calculations.
* Abacus is still used in some countries like China, Russia and Japan.


1. Abacus

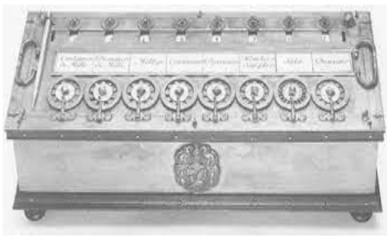
2. Napier's Logs and Bone's :

* Napier's Bones was a manually operated calculating device and invented by John Napier.
* This device used 9 different ivory strips (bones) marked with numbers to multiply and divide for calculation.
* First machine to use the decimal point system for calculation.

2. Napier's Bones

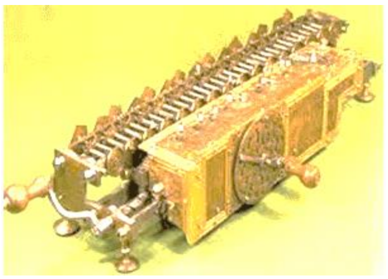
3. Pascal's Adding machine :

* Pascaline is also known as Arithmetic Machine or Adding Machine.
* First mechanical and automatic calculator by a French mathematician-philosopher Biaise Pascal.
* It was a wooden box with a series of gears and wheels. When a wheel is rotated one revolution, it rotates the neighbouring wheel. A series of windows is given on the top of the wheels to read the totals.


3. Pascaline

4. Leibnitz wheel calculator :

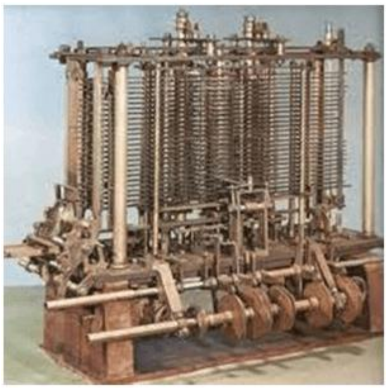
* It was developed by a German mathematician-philosopher Gottfried Wilhelm Leibnitz in 1673.
* He improved Pascal's invention to develop this machine. It was a digital mechanical calculator which was called the stepped reckoner as instead of gears it was made of fluted drums.
* It was able to perform multiplication & division.


4. Stepped Reckoner or Leibnitz wheel

5. Babbage's Difference Engine :
* Charles Babbage developed a machine called difference engine In early 1820s.
* These machines calculate logarithmic tables to a high degree of precision.

5. Babbage's Difference Engine

6. Babbage's Analytical Engine :
* Charles Babbage designs an analytical engine which begins a real model of modern days computer in 1830.
* He included the concept of central process storage area, memory and input/output.
* Because of inventions of difference engine and analytical engine, Charles Babbage has been considering as "Father of modern Computers".
* It was a mechanical computer that used punch-cards as input. It was capable of solving any mathematical problem and storing information as a permanent memory.


6. Babbage's Analytical Engine

7. Tabulating (or) Hollerith's Machine :
* Herman Hollerith developed electro mechanical punched card in 1890, that is used for input, output and storing of instructions.
* It could tabulate statistics and record or sort data or information. This machine was used in the 1890 U.S. Census.
* Hollerith also started the Hollerith's Tabulating Machine Company which later became International Business Machine (IBM) in 1924.

7. Tabulating (or) Hollerith's Machine

8. Mark-1 :
* Howard Aiken contracted an electro mechanical computer named mark-1 In the year 1944.
* It could multiply two 10-digit number in 5-seconds.
* It was also the first programmable digital computer marking a new era in the computer world.

8. Mark-1

9. ENIAC :
* Electronic Numerical Integrator and Calculator were built by Prof. Eckerit and Mauchly.
* It used about 19000 vacuum tubes and can perform about 300 multiplications per second.

9. ENIAC (Electronic Numerical Integrator And Computer)

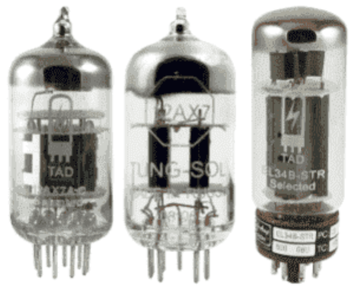
10. EDSAC :
* Electronic Delay Storage Automatic Computer was developed by Maurice willies.
* It has ability to input, output, staring the data.
* It also able to perform and control arithmetic calculations.
11. EDVAC :
* Electronic Discrete Variable Automatic Computer was developed by Prof. Eckerit and Mauchly.
* In this both the data and instruction can be stored in binary form instead of decimal number system.
12. UNIVAC :
* Universal Automatic Computer developed by Remington.
* It was cable of performing to access both numeric and alphabetic information.
13. UNIVAC-1:
* It is the first computer which is used for commercial purpose in 1954.

Generations of Computers: .

Evolution of Modern Computers from the olden days is known as "Generation of Computers." Generations of computers are broadly classified based on the following characteristics:

* Increasing in storage capacity.
* Increasing in processing speed.
* Increasing reliability.

There are totally 5 generations of computers till today. They are,

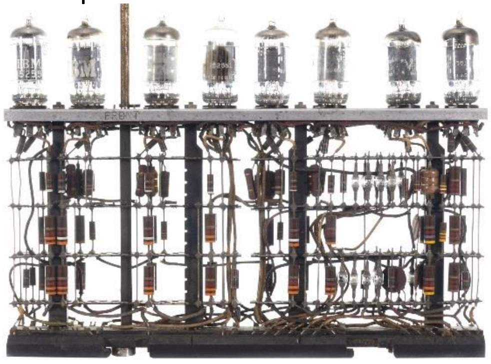


First Generation Computers:

Period: (1945-1955)

Technology: Vacuum tubes

* The ENIAC was the first computer developed in this generation.
* It was capable of performing 5000 additions (or) 350 multiplications per second.



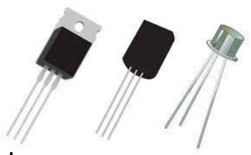
* It uses about 18000 vacuum tubes and it consumes 150 kw/hr.

Limitations:

The limitations of first-generation computers are as follows,

* Less operating capacity.
* High power consumption.
* Very large space requirement.
* Produce high temperature.
* Very costly.

Second Generation Computers:



Period: (1955-1965)

Technology: Transistors

* The second-generation computers use transistors as the main component in CPU.
* They are very small in size when compared to vacuum tubes and produce less heat.

* They are fast and reliable.
* By this invention of Transistors, the size, maintenance cost, power consumption has decreased.

* During this period magnetic storage devices have been started their development. Because of this, speed and storage capacity has been increased.
* They are capable to perform 20,000 to 50,000 additions per second.

Third Generation Computers:

Period: (1965-1975)

Technology: Integra ted Circuits (IC)

* In this generation the computers used integrated circuits instead of Transistors
* IC is a miniature form of an electronic circuit made of silicon and enclosed in a metal package.
* These ICs are caused "chips." The cost and size of the computers were greatly reduced.
* The magnetic disk technology has improved rapidly.
* It has capable to perform 10 million additions per second.

Fourth Generation Computers:

Period: (1975-1990)

Technology: Very Large Scale Integrated Circuit (VLSI)

* ICs packing nearly 10,000 transistors are grouped in to a single silicon chip known as "microprocessor".
* The computers which use microprocessors are called "Micro Computers".

* Intel Corporation invented the microprocessor in the year 1980 with this development the cost of a computer has reduced a lot.
* The floppy disk technology was developed during this generation.

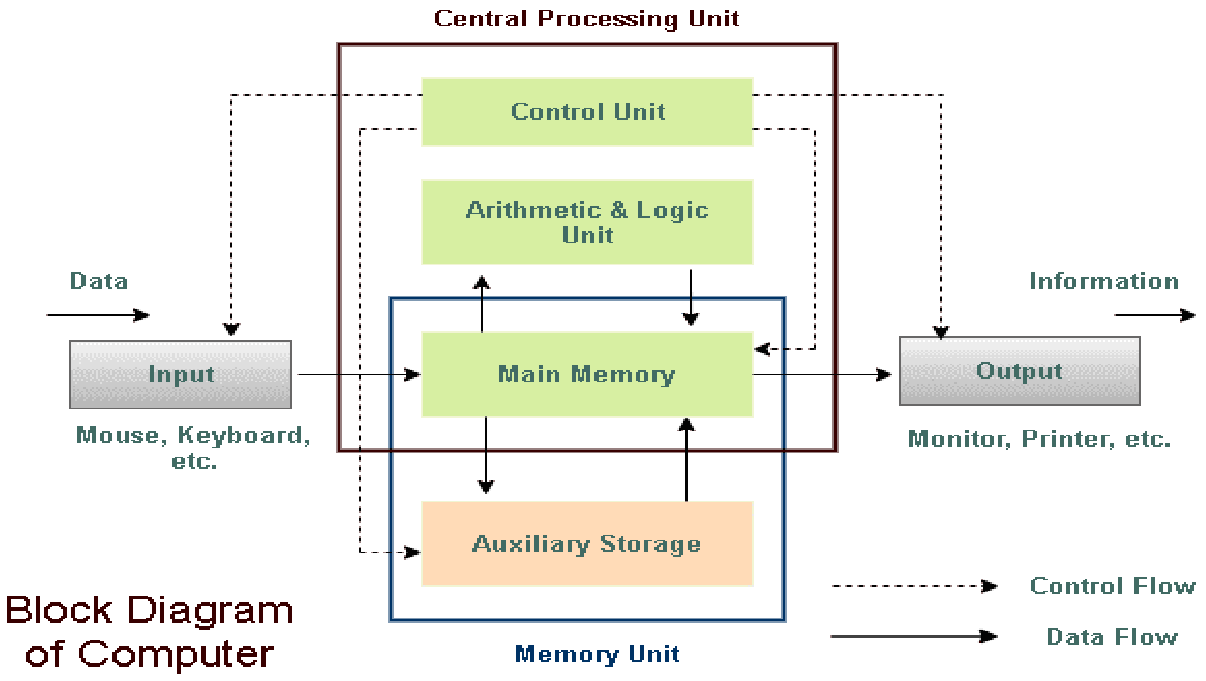
Fifth Generation Computers:

Period: (1990- till date)
Technology: Artificial Intelligence
* Artificial Intelligence is a technique by which we make the computer to think and take decisions in its own.
* These computers are under research.
* Artificial Intelligence can be achieved by means of problem solving, Game playing, and Expert systems.

Block Diagram of a Computer (or) Basic organization of a computer:

The word computer has been derived from the word "Compute" which means to calculate. It is an electronic device which inputs the data, stores the data, processes the data and gives the result accurately at a very high speed according to the instructions provided. (or)

A computer is an electronic machine that can be programmed to carry out sequences of arithmetic or logical operations automatically that accepts raw data as input and processes it with a set of instructions (a program) to produce the result as output.



Input Unit:

Computers need to receive data and instructions in order to solve any problem.

- The input unit consists of one or more input devices like keyboard, mouse, joystick etc.
- Regardless of the type of the input device used in a computer system, all input devices perform the following functions.
    - Accept the data
    - Convert data to computer understandable form.
    - Supply converted data for further processing.

CPU:

The actual processing of the data is carried out in the Central Processing Unit (CPU), which is the brain of computer. The CPU stores the data and instructions in the primary memory of the computer, called the Random Access Memory (RAM) and processes them from this location.

- CPU stands for “Central Processing Unit.”
- CPU is like a computer Brain. It performs the following operations.
    - o It performs all calculations.
    - o It takes all decisions.
    - o It controls all units of computer.
- Control unit, Memory unit and Arithmetic logic unit of the computers are together known as central processing unit.

The Control Unit (CU) and the Arithmetic Logic Unit (ALU) are the two subcomponents of the CPU. The ALU carries out the arithmetic and logical operations while the CU retrieves the information from the storage unit and interprets this information. The CPU also consists of circuitry devices called cache and registers.

Control Unit:

- The control unit controls all other units in the computer.
- The control unit instructs the input unit where to store data after receiving it from the user.
- It also controls the flow of data and instructions from the memory unit to “ALU”.
- It controls the flow of results from ALU to output unit.

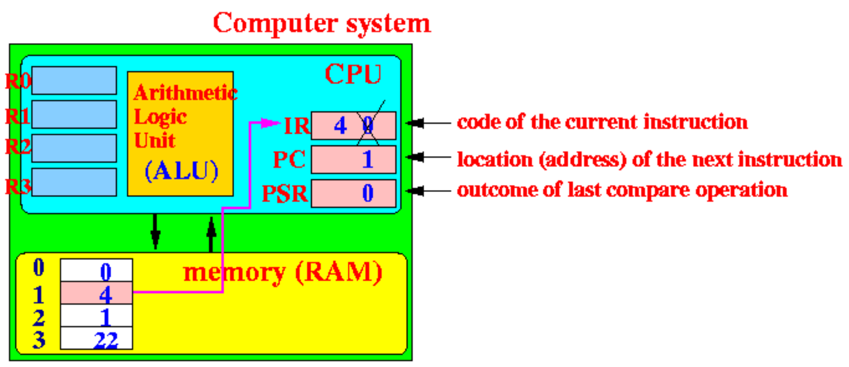
Arithmetic Logic Unit:

- All calculations are performed in ALU.
- It does comparisons and takes decisions.
- It can perform Basic Arithmetic operations like addition, subtraction as well as logical operations like less than greater than.
- After performing calculations, the result is stored in memory unit.

Program Counter:

A program counter is a register in a computer processor that contains the address (location) of the instruction being executed at the current time.

As each instruction gets fetched, the program counter increases its stored value by 1.



Computer system
CPU
R0
R1
R2
R3
Arithmetic Logic Unit (ALU)
IR 4 0
PC 1
PSR 0
code of the current instruction
location (address) of the next instruction
outcome of last compare operation
memory (RAM)
0 0
1 4
2 1
3 22

Memory Unit:

- Memory Unit of the computer holds data and instructions that we enter through the input unit.
- It is also used to preserve intermediate and final results before they are sent to the output unit.
- It is used to preserve the data for later usage.
- The various storage devices used for storage can be classified in to a categories namely,
    - Primary Memory
    - Secondary Memory
- Primary memory stores and provides information very fast but it loses the contents when we switch off the computer.
- Secondary memory stores the data permanently. The program that we want to run on the computer is first transferred to the primary memory from secondary.

Output Unit:

- The output unit of a computer provides information and results of an operation to the outside world.
- The output unit also converts Binary data to a form that uses can understand.
- The commonly used to output devices are Monitors, Printers, and Plotters.

Introduction to Programming Languages:

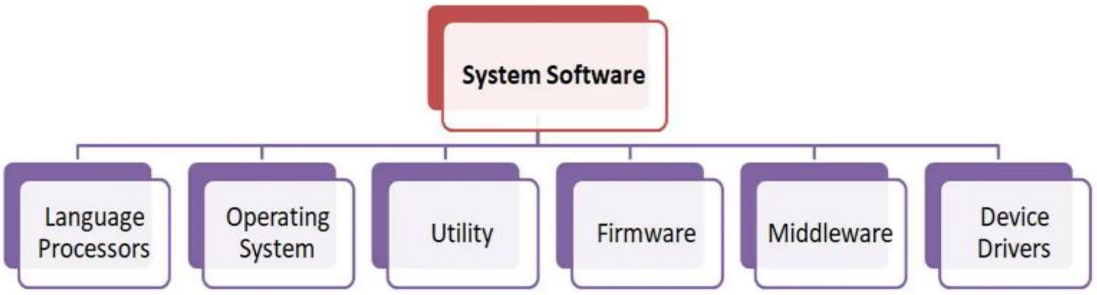
A programming language is a set of symbols, grammars and rules with the help of which one is able to translate algorithms to programs that will be executed by the computer. The programmer communicates with a machine using programming languages.

Program & Programming

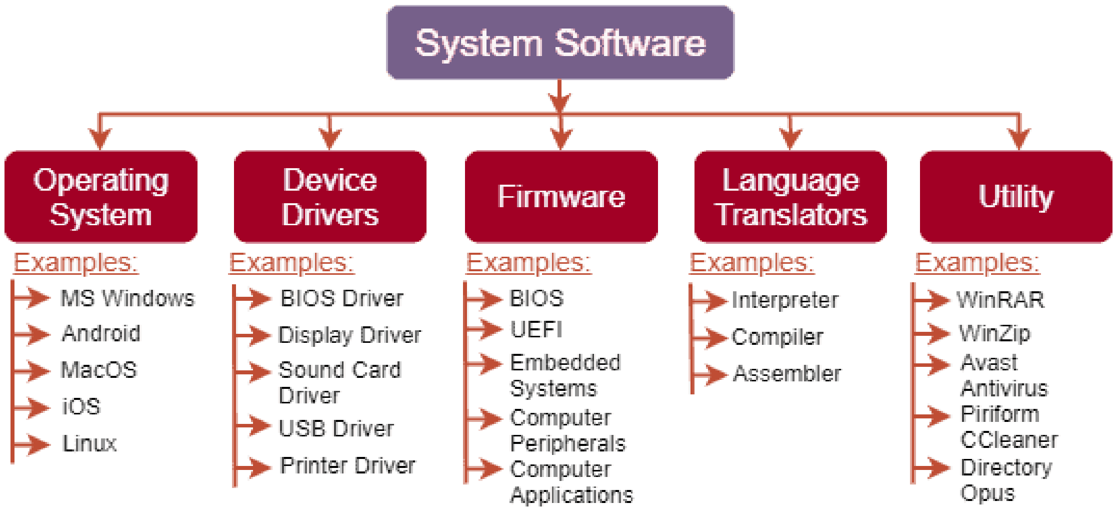
• A program is a set of logically related instructions that is arranged in a sequence that directs the computer in solving a problem.
• The process of writing a program is called programming.
• Software is a collection of computer programs and related data that provides the instructions for telling a computer what to do and how to do it.
• Computer software broadly classified into two categories: a) System software and b) Application software

a) System Software:

• System software is a collection of programs that interfaces with the hardware.
• Categories of system software:

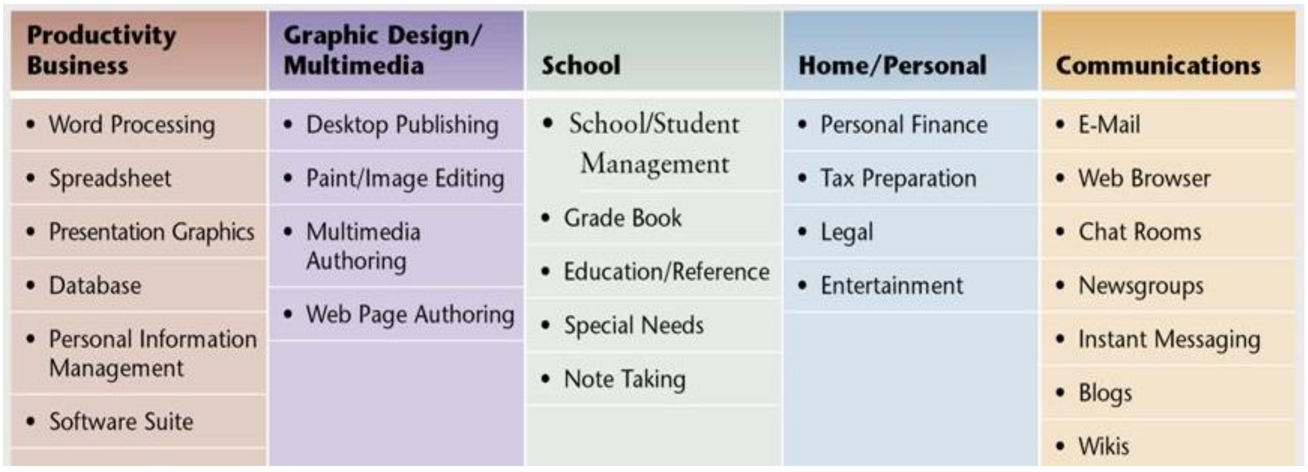


(or)



b) Application Software:

• Application software is written to enable the computer to solve a specific data processing task.
• Categories of application software:



Productivity Business	Graphic Design/ Multimedia	School	Home/Personal	Communications
• Word Processing	• Desktop Publishing	• School/Student Management	• Personal Finance	• E-Mail
• Spreadsheet	• Paint/Image Editing	• Grade Book	• Tax Preparation	• Web Browser
• Presentation Graphics	• Multimedia Authoring	• Education/Reference	• Legal	• Chat Rooms
• Database	• Web Page Authoring	• Special Needs	• Entertainment	• Newsgroups
• Personal Information Management		• Note Taking		• Instant Messaging
• Software Suite				• Blogs
				• Wikis

Programming Language:

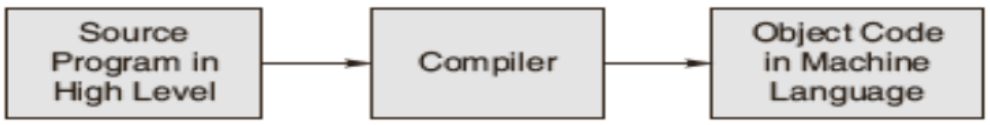
- A programming language is composed of a set of instructions in a language understandable to the programmer and recognizable by a computer.
- Programming languages can be classified as,
  - (a) High-level language - BASIC, COBOL & FORTRAN (application programs)
  - (b) Middle level language - C (application & system programs)
  - (c) Low level language - assembly language (system programs)
- System programming language:
  - – It is designed to make the computer easier to use.
  - – For ex.: OS which control many other programs that control i/o devices, multiple tasks.
- Application programming language:
  - – It is designed for specific computer application such as payroll, inventory etc.
  - – Its 2 sub categories are:
    - Business program
    - Scientific Program

Low level & High level Languages:

- *Machine Language (LOW):*
  - – In these a sequence of instructions written in the form of binary number consisting of 1's and 0's to which the computer responds directly. (it is also called as first generation language 1GL).
  - – For ex: 0011 1100 means, load A register with 7 value.
- *Assembly language (LOW):*
  - – When symbols such as letters, digits or special characters' are employed for operations, operands & other parts of instruction code, the representation is called an assembly language instruction. It is also 2GL.
  - – For Ex:- LD A 7 means, load A register with 7 value
- *High Level Language:*
  - – Having instruction that are similar to human languages and have a set grammar which makes programmer to write program and identify and correct error in them.
  - – For Ex:- int a=7 means, a value is 7

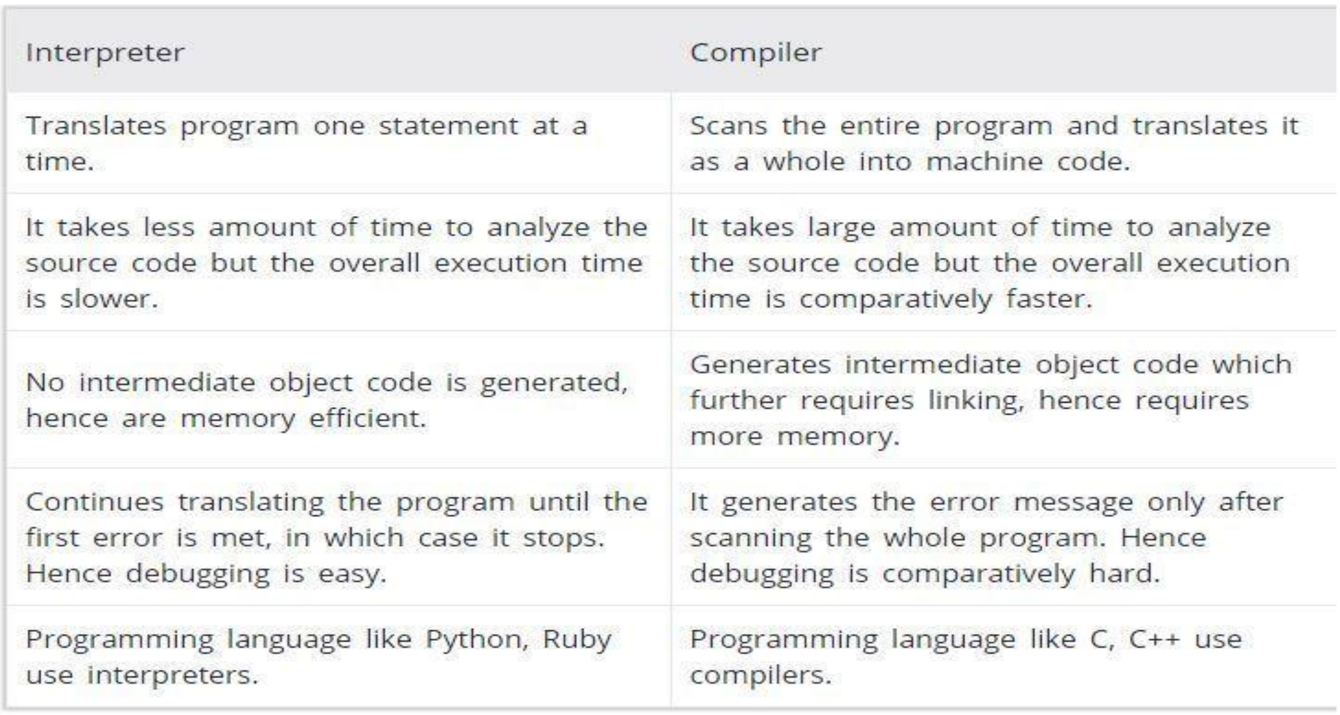
COMPILER:

- For executing a program written in a high-level language, it must be first translated into a form the machine can understand. This is done by a software called the compiler.
- The compiling process consists of two steps:
  - a) The analysis of the source program
  - b) Check for syntax error
- Compiler action:



INTERPRETER:

- During the process of translation There is another type of software that also does translation. This is called an interpreter.
- Differences between compiler and interpreter:



Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Third, Forth & Fifth Generation Languages:

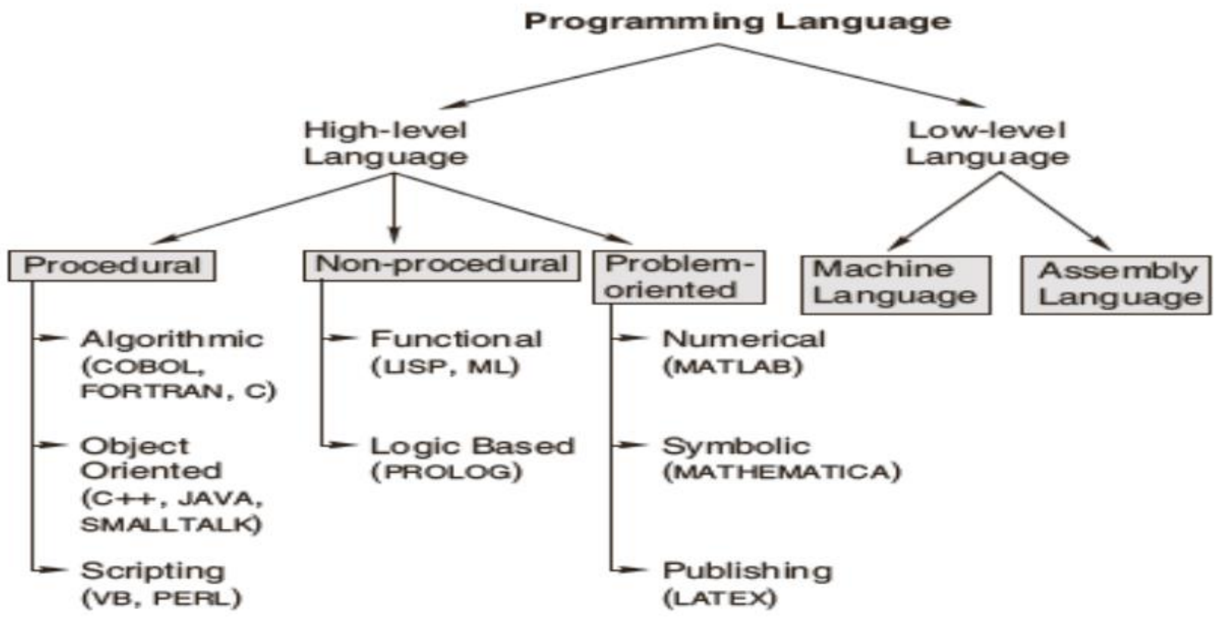
* The 3GLs are procedural in nature i.e., these languages specify "how to do". Hence 3GLs require the knowledge of complete steps to solve a problem.

* In case of 4GLs, they are non-procedural i.e., they specify "what to do". Hence in 4GLs we need to specify only what we required and rest gets done on its own

Advantages:

* Smaller code
* Reduced development time and maintenance cost
* Easy to programmers
* Doesn't require a high knowledge on program GUI Based Languages
* With the invention of GUI based interfaces, many programming languages are developed that help programmer to develop GUI applications.
* Some of the GUI based languages are,
  * Visual Basic (VB)
  * C#
  * VB.Net

Classification of Programming Languages:



Basics of a Computer Program:

Computer Programming is a set of instructions, that helps the developer to perform certain tasks that return the desired output for the given valid inputs.

Example: Given below is a Mathematical Expression,

- Z = X + Y, where X, Y, and Z are the variables in a programming language.
- If X = 550 and Y = 450, value of X and Y are the input values that are called literals.
- To calculate the value of X+Y, which results in Z, i.e. the expected output.

A computer is a machine that processes information and this information can be any data that is provided by the user through devices such as keyboards, mice, scanners, digital cameras, joysticks, and microphones. These devices are called Input Devices and the information provided is called input.

The computer requires storage to store this information and the storage is called Memory. Computer Storage or Memory is of Two Types. (i) Primary Memory or RAM (Random Access Memory): This is the internal storage that is used in the computers and is located on the motherboard. RAM can be accessed or modified quickly in any order or randomly. The information that is stored in RAM is lost when the computer is turned off called as Volatile. (ii) Secondary Memory or ROM (Read-Only Memory): Information (data) stored in ROM is read-only, and is stored permanently called as Non-Volatile. The ROM stored instruction is required to start a computer.

Processing: Operations done on this information (input data) is called Processing. The Processing of input is done in the Central Processing Unit which is popularly known as CPU.

Output Devices: These are the computer hardware devices that help in converting information into human-readable form. Some of the output devices include Visual Display Units (VDU) such as a Monitor, Printer, Graphics Output devices, Plotters, Speakers, etc.

Developers should have essential knowledge on the following concepts to become skilled in Computer Programming,

1) Algorithm: It is a set of steps or instruction statements to be followed to accomplish specific tasks.
2) Source code: Source code is the actual text used to construct the program using C language.

```
void main()
{
        // Steps to be performed
}
```

3) Compiler: Compiler is a software program that helps in converting the source code into binary code.
4) Data Type: Data of different types, integer, floating-point, character.
5) Variable: Variable is a space holder to store value in memory used in program. Example, int age = 25.
6) Conditionals: a certain condition should execute only if true or false to exit.
7) Array: Array is the variable that stores elements of a similar data type. Example: int a[10]
8) Loop: Loop is used to execute the series of code until the condition is true. Example, for, while, do-while.
9) Function: Functions or methods are used to accomplish a task in programming, a function can take parameters and process them to get the desired output. Functions are used to reuse them whenever required at any place repeatedly.

Algorithms:

Definition:

An algorithm is defined as a finite set of steps that provide a chain of actions for solving a problem. (or) An algorithm is a sequence of instructions that are carried out in a predetermined sequence in order to solve a problem. (or) A step-by-step procedure used to solve a problem is called Algorithm. (or) An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conducting a sequence of specified actions.

Procedure for writing an algorithm:

An Algorithm is a well-organized and textual computational module that receives one or more input values and provides one or more output values.

- These well-defined steps are arranged in a sequence that processes given input into output.
- The steps of the algorithm are written using English like statements which are easy to understand.
- This will enable the reader to translate each step into a program.
- Every step is known as instruction.
- An algorithm is set to be accurate only when it provides the exact required output.
- If the procedure is lengthy then sub divided into small parts as it makes easy to solve the problem.

Analyzing an Algorithm:

When one writes an algorithm, it is essential to know how to analyses the algorithm.

- Analyzing the algorithm refers to calculating the resources such as computer memory, processing time, logic gates and so on …..
- Time is most important resource because the program developed should be faster in processing.
- The analysis can also be made by reading the algorithm for logical accuracy, tracing the algorithm and checking with the data.

Categories of Algorithm:

The steps in an algorithm can be divided into three categories, namely

- Sequence
- Selection and
- Iteration

Sequence

- The steps described in an algorithm are performed successively one by one without skipping any step.
- The sequence of steps defined in an algorithm should be simple and easy to understand.

Example:
```
// adding two timings
Step 1: start
Step 2: read h1, m1, h2, m2
Step 3: m=m1+m2
Step 4: h= h1 + h2 + m/60
Step 5: m=m mod 60
Step 6: write h and m
Step 7: stop
```

Selection

- We understand that the algorithms written in sequence fashion are not reliable. There must be a procedure to handle operation failure occurring during execution.
- The selection of statements can be shown as follows,

```
if(condition)
        Statement-1;
else
        Statement-2;
```

- The above syntax specifies that if the condition is true, statement-1 will be executed otherwise statement-2 will be executed.
- In case the operation is unsuccessful. Then sequence of algorithm should be changed / corrected in such a way that the system will re-execute until the operation is successful.

Example:
```
// Person eligibility for vote
Step 1: start
Step 2: read age
Step 3: if age > = 18 then step_4 else step_5
Step 4: write "person is eligible for vote"
Step 5: write "person is not eligible for vote"
Step 6: stop
```

<u>Iteration</u>

- In a program, sometimes it is very necessary to perform the same action for a number of times.
- If the same statement is written repetitively, it will increase the program code.
- To avoid this problem, iteration mechanism is applied.
- The statement written in an iteration block is executed for a given number of times based on certain condition.

Example:
```
// sum of individual digits in the given number
Step 1: start
Step 2: read n
Step 3: repeat step 4 until n>0
Step 4:  (a) r=n mod 10
         (b) s=s+r
         (c) n=n/10
Step 5: write s
Step 6: stop
```

Some more examples:

<u>Algorithm to add two numbers:</u>	***<u>Algorithm to find largest among three numbers:</u>***	***<u>Algorithm to reverse the number:</u>***
Step 1: START Step 2: Declare integers a, b & c Step 3: Define values of a & b Step 4: add values of a & b Step 5: store output of step 4 to c Step 6: print c Step 7: STOP ***<u>Algorithm to find odd or even number:</u>*** Step 1: Start Step 2: [Take Input] Read: Number Step 3: Check: If Number%2 == 0 Then Print : N is an Even Number. Else Print : N is an Odd Number. Step 4: Exit	Step 1: Start Step 2: Declare variables a,b and c. Step 3: Read variables a,b and c. Step 4: If a > b If a > c Display a is the largest. Else Display c is the largest. Else If b > c Display b is the largest. Else Display c is the largest. Step 5: Stop	

Flowcharts (using Dia Tool):

A flowchart is an alternative technique for solving a problem. Instead of descriptive steps, we use pictorial representation for every step.

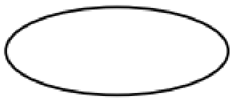
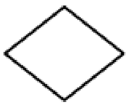
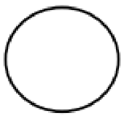
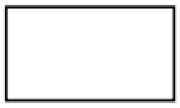
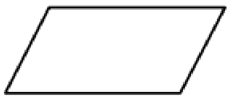
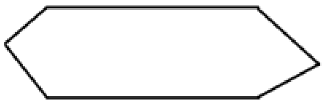
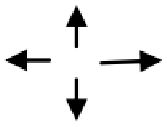
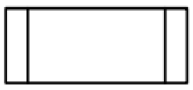
Definition:

Flowchart is a diagrammatic or pictorial representation of various steps involved in the Algorithm.

A complete flowchart enables us to organize the problem into a plan of actions i.e. it specifies what comes first, second, third, ...

* Flowchart also represents the flow of data.
* It is an easy way to solve the complex problems because it makes the reader to flow the process quickly from the flowchart incited of going through text.
* A flowchart is a set of symbols that indicates various operations in a program.
* For every process there is a corresponding symbol in the flowchart.
* Once the algorithm is written, its pictorial representation can be done using flowchart symbol.

Some of the commonly used flowchart symbols are listed below,

Symbol	**Description**
	Start/Stop
	Decision
	Connector
	Process
	Input/output
	Loop
	Data Flow Direction Lines
	Predefined Process

Note: (Name of the Symbols)

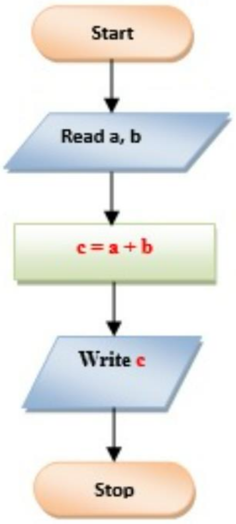
Oval, Diamond, Circle, Square (or) Rectangle, Parallelogram, hexagon shaped symbol, arrows (Data Flow lines), Subroutine Symbol (rectangle with a line at each end of the shape).

Flowchart Example:

• Problem 1: Draw a flowchart to find the sum of two numbers.

Algorithm **Flowchart** **Program**

1. Start
2. Read a, b
3. c = a + b
4. Print or display c
5. Stop

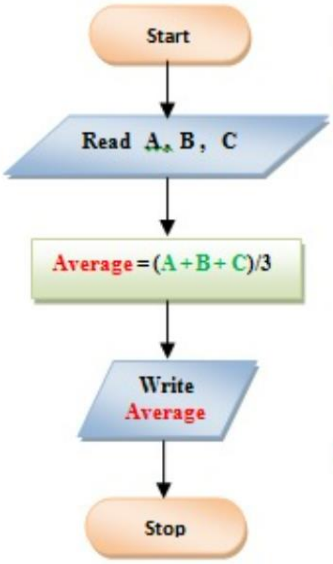


```
#include<stdio.h>

int main()
{
    int a, b, c;

    printf("Enter value of a: ");
    scanf("%d", &a);

    printf("Enter value of b: ");
    scanf("%d", &b);
    c = a+b;

    printf("Sum of given two numbers is: %d", c);

return 0;
}
```

• Problem 2: Draw a flowchart to calculate the average of three two numbers.

Algorithm **Flowchart** **Program**

1. Start
2. Read 3 numbers A, B, C
3. Calculate the average by the equation:
   Average = (A + B + C)/3
4. Print average
5. Stop

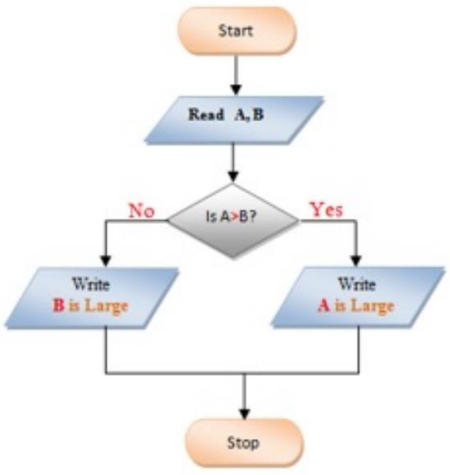


```
#include<stdio.h>

int main()
{

    int A, B, C;
    float Average;

    printf("Enter values of A, B, C: \n");
    scanf("%d %d %d", &A, &B, &C);

    Average = (A+B+C)/3;

    printf("Average of given 3 numbers is: %f", Average);

    return 0;
}
```

• Problem 3: Draw a flowchart to find the greatest of two numbers.

Algorithm **Flowchart** **Program**

```
1. Start
2. Read A,B
3. If A > B then
        Print A is large
   else
        Print B is large
4. Stop
```



```
#include<stdio.h>

int main()
{

    int A, B;

    printf("Enter values of A, B: ");
    scanf("%d %d", &A, &B);

    if (A>B)
    printf("A is Larger");
    else
    printf("B is Larger");

    return 0;
}
```

• Problem 4: Draw a flowchart to swap two numbers.

Algorithm

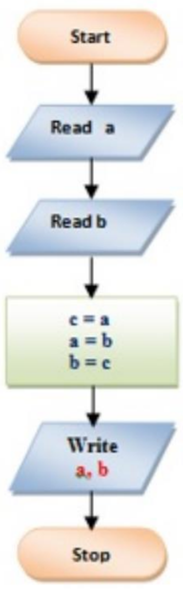
1. Start
2. Read two values into two variables a, b
3. Declare third variable, c

   c = a

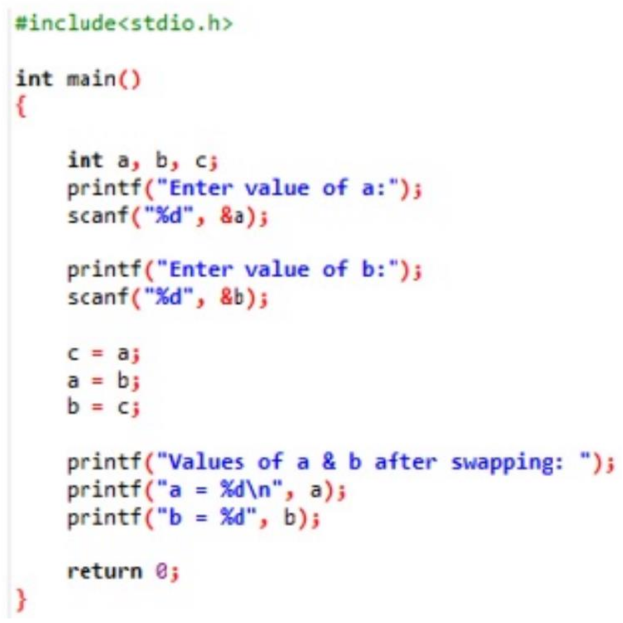
   a = b

   b = c
4. Print or display a, b
5. Stop

Flowchart



Program



```c
#include<stdio.h>

int main()
{

    int a, b, c;
    printf("Enter value of a:");
    scanf("%d", &a);

    printf("Enter value of b:");
    scanf("%d", &b);

    c = a;
    a = b;
    b = c;

    printf("Values of a & b after swapping: ");
    printf("a = %d\n", a);
    printf("b = %d", b);

    return 0;
}
```

Pseudo Code:

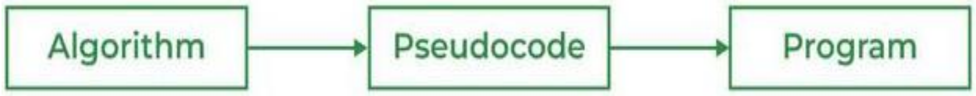
A Pseudocode is defined as a step-by-step description of an algorithm. Pseudocode does not use any programming language in its representation instead it uses the simple English language text as it is intended for human understanding rather than machine reading.

Pseudocode is the intermediate state between an idea and its implementation (code) in a high-level language.

Algorithm and Pseudocode are the two related terms in computer programming. The basic difference between algorithm and pseudocode is that an algorithm is a step-by-step procedure developed to solve a problem, while a pseudocode is a technique of developing an algorithm.

What is the need for Pseudocode

Pseudocode is an important part of designing an algorithm, it helps the programmer in planning the solution to the problem as well as the reader in understanding the approach to the problem. Pseudocode is an intermediate state between algorithm and program that plays supports the transition of the algorithm into the program.



Pseudocode is an intermediate state between algorithm and program

Five important rules for writing pseudocode are:

1. Write one statement per line.
2. Initial keywords should be represented in capital case (READ, WRITE, IF, WHILE, UNTIL).
3. Indentation of pseudocode should be similar to the actual program to show hierarchy.
4. Ending the multiline structure is necessary.
5. Keep statements in simple language (English).

Difference between Algorithm and Pseudocode:

Algorithm	**Pseudocode**
An Algorithm is used to provide a solution to a problem in form of a well-defined step-based form.	A Pseudocode is a step-by-step description of an algorithm in code-like structure using English text.
An algorithm only uses simple English words	It uses reserved keywords like if-else, for, while, etc.
These are a sequence of steps of a solution to a problem	These are fake codes as the word pseudo means fake, using code like structure and plain English text
There are no rules to writing algorithms	There are certain rules for writing pseudocode
Algorithms can be considered pseudocode	Pseudocode cannot be considered an algorithm
It is difficult to understand and interpret	It is easy to understand and interpret

Example 1:

To implement a simple program that calculates the average of three numbers using pseudo code in C:
Pseudo code:

1. Start
2. Input three numbers
3. Calculate the sum of the three numbers
4. Divide the sum by 3 to get the average
5. Display the average
6. End
7.

Example 2:

To find the Factorial value of the given number using pseudo code in C:

Algorithm to find Factorial of n:

```
Step 1: start
Step 2: initialize fact = 1
Step 3: input the user value n
Step 4: for i=1 to i <= n repeat
        the process
Step 5: fact = fact * i
Step 6: i++ [increment i by one]
Step 7: print fact value
Step 8: stop
```

Pseudocode to Factorial of n:

```
Start program
Declare fact and n
Enter number for n
for i=1 to i <=n
Perform fact = fact * i
Display fact
End program
```

C program to find Factorial of n:

```
#include<stdio.h>
void main()
{
int n, fact=1,i;
printf("enter value for n: ");
scanf("%d", &n);
for(i=1; i<=n; i++)
{  fact=fact*i;  }
printf("\n factorial is: %d", fact);
}
```

Example 3:

To find the Sum of Natural Numbers up to the given number using pseudo code in C:

Algorithm to find the Sum of Natural Numbers *up to n:*

```
Step 1: start
Step 2: declare and initialize n,
        sum = 0 and i
Step 3: Input number n
Step 4: for i=1 to i<=n
Step 5: sum = sum + i
Step 6: i++ [increment i by one]
Step 7: print sum
Step 8: stop
```

pseudo-code to find the Sum of Natural Numbers up to n:

```
Start program
Declare variables n, sum=0 and i
Enter the number for n
For i=1 to i<=n
Perform operation sum = sum + i
Increment i value by one
Print sum
End program
```

C program to find the Sum of Natural Numbers *up to n:*

```
#include<stdio.h>
void main()
{
int n, sum=0, i;
printf("enter no of terms: ");
scanf("%d", &n);
for(i=1; i<=n; i++)
{    sum = sum+i;    }
printf("sum of series=%d",sum);
}
```

INTRODUCTION TO COMPILATION AND EXECUTION:

(i) Compilation of the High Level Language:

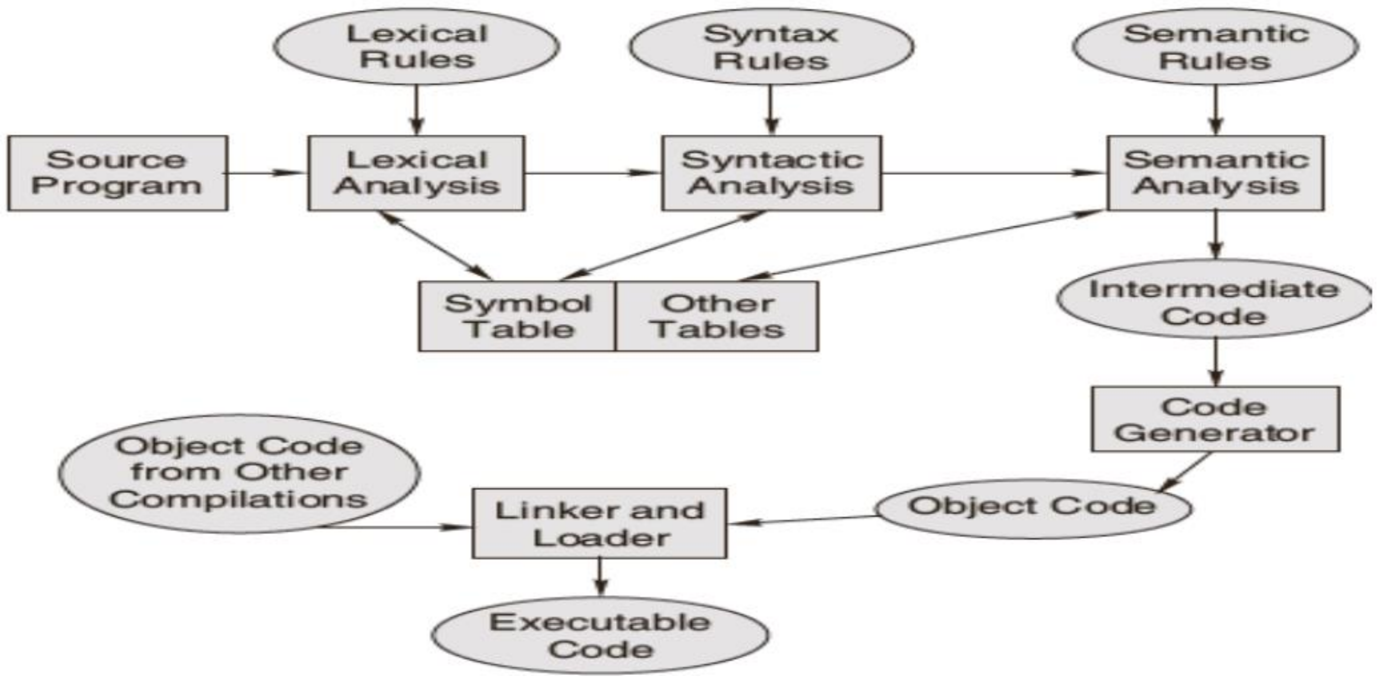
Compilation process in C is also known as the process of converting Human Understandable Code (C Program) into a Machine Understandable Code (Binary Code). Compilation process in C involves four steps: pre-processing, compiling, assembling, and linking.

- *The preprocessor tool* helps in comments removal, macros expansion, file inclusion, and conditional compilation. These commands are executed in the first step of the compilation process.
- *Compiler software* helps boost the program's performance and translates the intermediate file to an assembly file.
- *Assembler* helps convert the assembly file into an object file containing machine-level code.
- *Linker* is used for linking the library file with the object file. It is the final step in compilation to generate an executable file.

The compiling process consists of two steps: i) The analysis of the source program and ii) The synthesis of the object program in the machine language of the specified machine.

• The analysis phase uses the precise description of the source programming language.
• A source language is described using a) lexical rules, b) syntax rules, and () semantic rules.

The Process of Compilation:



Block diagram of the Process of Compilation

(ii) Execution Steps of a Program:

The following are the execution steps of a program:

1. ***Translation of the program*** resulting in the object program.
2. ***Linking of the translated program*** with other object programs needed for execution, thereby resulting in a binary program.
3. ***Relocation of the program to execute*** from the specific memory area allocated to it.
4. ***Loading of the program in the memory*** for the purpose of execution.

Linker:

- Linking resolves symbolic references between object programs.
- It makes object programs known to each other.
- Linking makes the addresses of programs known to each other so that transfer of control from one subprogram to another or a main program takes place during execution.
- In FORTRAN / COBOL, all program units are translated separately.

Relocation:

- Relocation is more than simply moving a program from one area to another in the main memory.
- Relocation means adjustment of all address-dependent locations, such as address constant, correspond to the allocated space, which means simple modification of the object program so that it can be loaded at an address different from the location originally specified.

Loader:

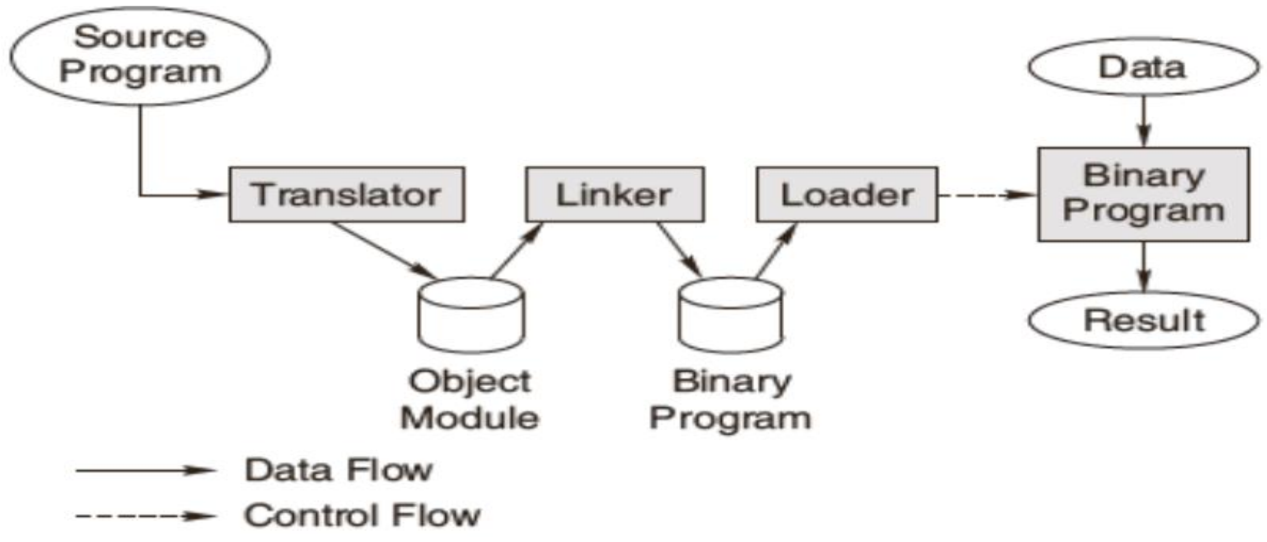
- Loading means physically placing the machine instructions and data into main memory, also known as primary storage area.
    a. Assignment of load-time storage area to the program
    b. Loading of program into assigned area
    c. Relocation of program to execute properly from its load time storage area
    d. Linking of programs with one another

Program Execution:

- When a program is compiled and linked, each instruction and each item of data is assigned an address.
- At execution time, the CPU finds instructions and data from these addresses.
- The program counter, is a CPU register that holds the address of the next instruction to be executed in a program.

The CPU has random access capability to any and all words of the memory, no matter what their addresses

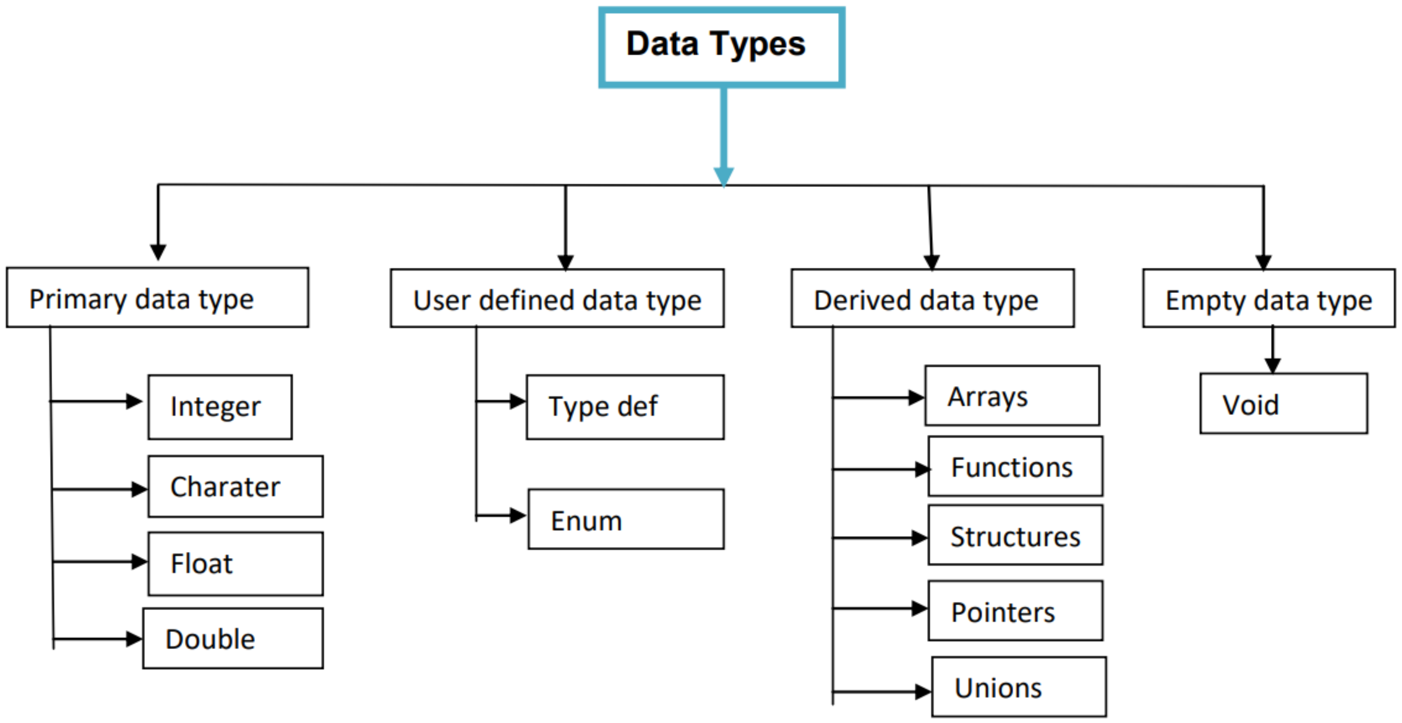
The Process of Program Execution:



Block Diagram of the Process of Program Execution

PRIMITIVE DATA TYPES:

Data Types:



Primary Data Type	Size (in bytes)	Range
int	**2**	**-32768 to 32767**
unsigned int	2	0 to 65535
signed	2	-32768 to 32767
short	**2**	**-32768 to 32767**
unsigned short	2	0 to 65535
signed short	2	-32768 to 32767
long	**4**	**-2147483648 to 2147483647**
unsigned long	4	0 to 4294967295
signed long	4	-2147483648 to 2147483647
char	**1**	**-128 to 127**
unsigned char	1	0 to 255
signed char	1	-128 to 127
float	**4**	**3.4E-38 to 3.4E +38**
double	**8**	**1.7E -308 to 1.7E +308**
long double	**10**	**34E -4932 to 34E +4932**

Note:

- Float is a single precision value. (After decimal 7 digits has been considered)
- Double is a double precision value. (After decimal 14 digits has been Considered)
- In-Built data Types are supported by “C” compiler by default.

1. Character (Denoted as "char" in C programming language)
Description: A character denotes any alphabet, digit or special symbol used to represent in formation and it is used to store a single character.

Storage space: 1 byte
Format: %c
Range of Values: ASCII Character Set.

2. Integer (Denoted as "int" in C programming language)
Description: Integer type is used to store positive and negative integer.

Storage space: 2 bytes.
Format: %d
Range of values: -327687 to +32767.
(The storage space may vary depending on the operating system)

3. Float point (Denoted as "float" in C programming language)
Description: It is used to store real number, with single precision floating point number (precision of 6 digits after decimal points.)

Storage space: 4 bytes.
Format: %f
Range of values: -3.4*1038 to +3.4*1038.

4. Double (Denoted as "double" in C programming language)
Description: It stores real numbers with double precision. The use of double doesn't guarantee to double the number of significant digits in our result, but it improves the accuracy of the arithmetic and reduces the accumulation of rounding errors.

Storage Space: 8 bytes.
Format: %ld
Range of values: -1.7*10308 to +1.7*10308.

5. void:
Description: It is a special data type used for,

- To specify that a function doesn’t returns any value.
- To specify a function takes no arguments.
- To create generic pointers.

Eg: 1. void print (void)
1. void *ptr

<u>VARIABLES AND CONSTANTS:</u> .

<u>VARIABLES:</u>
Variable Names:

- A variable is an identifier that is used to store data elements.
- The data may be numerical quantity (or) a character constant.
- The data item must be assigned a value to the variable at some point in the program. Then data item can be accessed by referring to the variable name.
- In other words, a variable is an identifier which changes its value during the execution of the program.

Rules for Constructing a Variable:

- A variable name is a combination of alphabets, digits and underscore.
- Other than the underscore no special characters is allowed.
- The first character in the variable name must be an alphabet.
- The variable name should not be of keyword.

Declaration of a Variable:

- Any variable which is used in the program should be declared first.
- A declaration of a variable specifies two things:
    - It tells the compiler about the name of the variable.
    - It specifies about the type of data that a variable can hold.

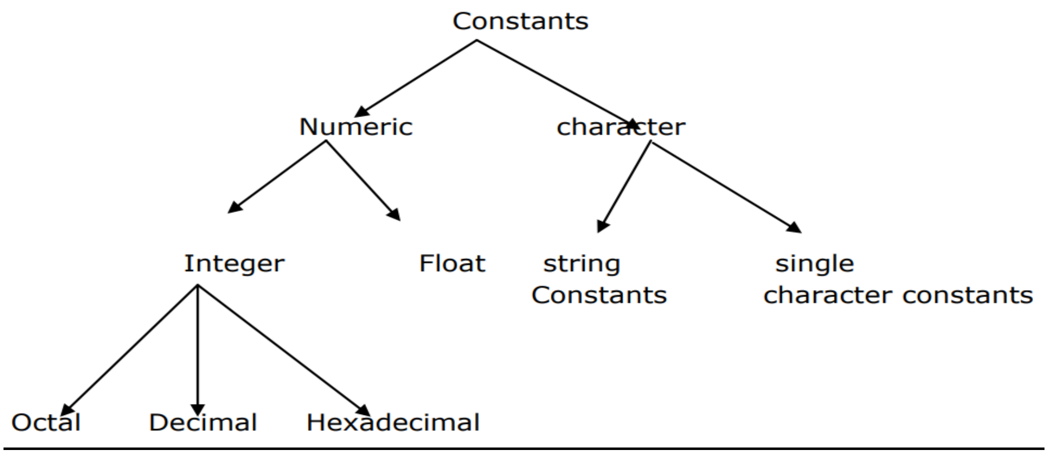
Syntax: data type variable 1, variable 2 , ... variable;
Eg: int a, b, c;

- The above declaration results in declaring the variables a, b, c as integer data type.

CONSTANTS:

Constants and its types:

- These are fixed values that will not change during the execution of program.
- There are 4 basic types of constants in 'C'. They are



Integer Constant:

An integer constant is an integer valued number. Rules for Constructing Integer Constant:

- It shouldn't have a decimal point.
- An integer constant must have at least one digit.
- Commas, Blank spaces, special characters can't be included with in the constants.
- The constant can be preceded by a minus sign if desired.
- The range of integer constants is -32,768 to 32,767

Types of Integer Constants:

- Integer Constants can be written in 3 different number systems, namely
    - Decimal {Base = 10}
    - Octal {Base = 8}
    - Hexa Decimal {Base = 16}

Decimal Numbers:

- A Decimal integer constant can consist of any combinations of digits taken from the set 0-9.
- If the constant contains 2 (or) more digits, the first digit must be something other than zero.

Eg: 0, 1, 723, 32156

Octal Constants:

- An Octal integer constant can consist of any combination of digits taken from the set 0-7.
- The first digit must be zero in order to identify the constant as an octal number.

Eg: 0, 01, 0723.

Hexa Decimal Numbers:

- A Hexa Decimal integer constant must begin with either 0x, (or) 0x.
- It can consist of the digits taken from the set 0-9, and a-f {upper case / lower case}.
- The letters a-f, represent the quantities 10-15 respectively.

Eg: 0x, 0x, 0x7.

Unsigned & Long Integer Constants:

- An Unsigned integer constant can be identified by appending the letter 'u'. {either upper or lower case}
- Long interval constants can be identified by appending the letter 'L' {either upper or lower case} to the end of the constant.
- An unsigned long interval constant can be specified by appending the letters UL {either upper case (or) lower case} to the end of the constants.
- Note that 'u' must precede the 'L'.

Eg: 234U – Unsigned Decimal Constant
0427UL – Unsigned Long Octal Constant
0X72FL – Long Hexa Decimal Constant

Floating Point Constants:

- A floating-point constant is a decimal number that contains a decimal point.

Rules:

- Commas and Blank spaces can't be included.
- It can contain of either decimal point (or) Exponent.
- The constant can be preceded by minus sign it desired.

Eg: 0, 1.0, 12.3.

Scientific Notation:

- When we want to represent very large values (or) very small values, instead of representing the value of sequence of digits we can represent a floating-point constant value in a scientific notation as follows,

  Mantissa E (or) e exponent.
  Eg: 1.2E + 3 => 1.2 * (10^3)
- Note that if no sign is represented after 'E' it will be treated as positive.
- Floating point constants are identified by appending the letter F {either upper case (or) lower case} to the end of the constants.
  Eg: 3E5F = 300000F.

Character Constants:

- A character constant is a single character enclosed in single quotation marks. The Max length of character constant is "one". The single character can be either alphabet (or) Digit (or) a special symbol.
- Eg: 'a', '1', '?'
  Note: Most computers make use of the "ASCII" { American Standard Code for Information Interchange }, character set, in which each individual character is numerically encoded.
- ASCII code is a 7-bit code. ASCII values ranges from 0-127.

String Constants:

- A string constant is an array of characters that has a fixed value enclosed within double quotation marks (" ").
- String constant is a sequence of characters enclosed within a pair of double quotes.
- For example, "DataFlair", "Hello world!"

BASIC INPUT AND OUTPUT:

C language has standard libraries that allow input and output in a program. The stdio.h or standard input output library in C that has methods for input and output.

Formated Input:

scanf()

The scanf() method, in C, reads the value from the console as per the type specified.

Syntax: scanf(“%X”, &variable of X Type);

where %X is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and & is the address operator in C, which tells the compiler to change the real value of this variable, stored at this address in the memory.

Formated Output:

printf()

The printf() method, in C, prints the value passed as the parameter to it, on the console screen.

Syntax: printf(“%X”, variable of X Type);

where %X is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and & is the address operator in C, which tells the compiler to change the real value of this variable, stored at this address in the memory.

The basic type in C includes types like int, float, char, double. Inorder to input or output the specific type, the X in the above syntax is changed with the specific format specifier of that type.

Example: C program to show input and output

```
#include <stdio.h>
int main()
{
        int num;                            // Declare the variables
        char ch;
        float frac;
        printf("Enter the integer: ");      // Integer – i/o
        scanf("%d", &num);
        printf("\n Entered integer is: %d", num);

        printf("\n Enter the float: ");     // Float – i/o
        scanf("%f", &f);
        printf("\n Entered float is: %f", frac);

        printf("\n Enter the Character: "); // Character – i/o
        scanf("%c", &ch);
        printf("\n Entered char is: %c", ch);

        return 0;
}
```

Output:

```
        Enter the integer: 10
        Entered integer is: 10

        Enter the float: 2.5
        Entered float is: 2.500000

        Enter the Character: A
        Entered Character is: A
```

OPERATORS: .

An operator is a symbol that informs to the computer to perform a particular task.

- The data items that operators act upon are called "operands".
- If the operator requires only one operant then it is called "unary operator". If it requires two then it is called "Binary Operator".

'C' language supports a different set of operators,

i) Arithmetic Operators:

Operator	**Purpose**	***Expression***	**Examples**
+	Addition	a + b	7 + 2 =
-	Subtraction	a - b	7 – 2 =
*	Multiplication	a * b	7 * 2 =
/	Division	a / b	7 / 2 =
%	Modulo Division	a % b	7 % 2 =

Example 1: Write a C program to perform arithmetic operations.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a, b, c;

        printf(" Enter a, b values: ");
        scanf("%d%d", &a, &b);

        c = a + b;
        printf("sum= %d \n", c);













}
```

Output:

```
Enter a, b values:
sum=
sub=
mul=
div=
mod=
```

ii) Relational Oprators:

- Relational Operators are used to perform comparison between two values.
- These operators‟ returns true (1) if the comparison condition is true otherwise false (0).
- The operators used for comparison in ‘C’ are listed below,

Operator	**Description or Action**	**Expression**	**Example**
<	less than	a < b	5 < 10
<=	less than or equal to		
>	greater than		
>=	greater than or equal to		
==	equal to		
!=	not equal to		

Example 1:

35 < 5	- false (0)
10 <= 10	- true (1)
45 > 5	-
11 >= 5	-
35 == 10	-
35 != 10	-

(10+5) == (5*3) - true (1)

Example 2: Write a program to use various relational operators and display their return values?

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

        printf("\n Condition \t : Return values \n ");

        printf("\n 10 != 10 \t : %5d", 10 != 10);









        getch();

}
```

Output:

```
10 != 10      :   0
10 == 10      :
10 >= 10      :
10 > 5        :
10 <= 100     :
10 < 9        :   0
```

iii) Logical Oprators:

- To take a decision based on 2 (or) more conditions then use logical operators.
- There are three logical operators available in 'C'

Operator	Purpose	Expression		Example
&&	Logical AND	if(a<b) && (a<c)	- false(0)	a=5; b=10; c=3
||	Logical OR	if(a<b) || (a<c)	-	a=5; b=10; c=3
!	Logical NOT	if(a != b)	-	a=5; b=10; c=3

Truth Table

A	B	A && B	A	B	A || B	A	! A
F	F	F	F	F	F	F	T
F	T	F	F	T	T	T	F
T	F	F	T	F	T		
T	T	T	T	T	T		

Example 1: Write an example program using logical operators?

```
#include<stdio.h>
#include<conio.h>
void main()
{
        clrscr();
        printf("condition \t \t : Return values\n");
        printf("\n 9>5 && 8<10 : %5d", 9>5 && 8<10);
        printf("\n 10<9 && 3>7 : %5d", 10<9 && 3>7);
        printf("\n 9>6 || 6<2 : %5d", 9>6 || 6<2);
        printf("\n 14 != 14 : %5d", 14 != 14);
        printf("\n 9 != 10 : %5d", 9 != 10);
        getch();
}
```

Output:

```
Condition                : Return values
```

iv) Increment / Decrement Operator (++ and - -)

- Increment operator increases value by 1. Decrement operator decreases value by 1.
- This operator can be used in two forms namely Prefix and Postfix.
- In case of Prefix notation, the operator precedes the operant {++a}.
- In case of Postfix notation, the operator is followed by operand {a++}.
- In Prefix notation increment operation will be given the highest priority i.e., first the value is incremented by 1 and other operations are performed later.
- In Postfix notation, increment operator will be given the least priority i.e., the increment operation is performed at last after all operations are performed.
- The above points are applicable to the decrement operator also.

Example:	i/p	int a = 5, c=0;	int a = 5, c=0;
	process	c = ++a;	c = a++;
	o/p	c=6, a=6	c=5, a=6.

v) Bitwise Operators:

- Bitwise operators are one of the salient features of 'C' language.
- These are specially designed to manipulate the data at Bit level.
- The Bitwise operators are not applicable for float (or) Double data type.
- The following are the some of the Bit wise operators available in 'C' language.

Operator	**Meaning**	
&	Bitwise Logical AND	
|	Bitwise Logical OR	
^	Bitwise Logical XOR	
<<	Left Shift	
>>	Right Shift	
~	Bitwise Compliment	[2's Complement = Bitwise 1's Compliment number + 1]

vi) Assignment Operators:

- This operator is used to assign a constant value (or) the result of an expression to a variable.
- In this operation Right hand side expression is evaluated first and then the result is assigned to left hand side variable.

Operator	*Meaning*
=	- Assign Right Hand value to LHS value
+=	- Value of LHS add to value of RHS and assign it back to the variable in LHS Eg: a += 2; which means, a = a + 2;
-=	- Value of RHS variable will be subtracted from the value of LHS and assign it back to the variable in LHS. Eg: a -= 2; which means, a = a – 2;
*=	- Value of LHS variable will be multiplied to the value of RHS and Assign it back to the variable in LHS. Eg: a *= 2; which means, a = a * 2;
/=	- Value of LHS variable will be divided by the value of RHS and Assign it back to the variable in LHS. Eg: a /= 2; which means, a = a * 2;
%=	- The Remainder will be stored back to the LHS after integer, division Is carried out between the LHS variable and RHS variable. Eg: a %= 2 which means, a = a % 2.

Example:

```
#include <stdio.h>
int main()
{
        int a = 5, c;
        c = a;
        printf("c = %d\n", c);          // c is 5
        c += a;
        printf("c = %d\n", c);          // c is 10
        c -= a;
        printf("c = %d\n", c);          // c is 5
        c *= a;
        printf("c = %d\n", c);          // c is 25
        c /= a;
        printf("c = %d\n", c);          // c is 5
        c %= a;
        printf("c = %d\n", c);          // c = 0
        return 0;
}
```

Output:

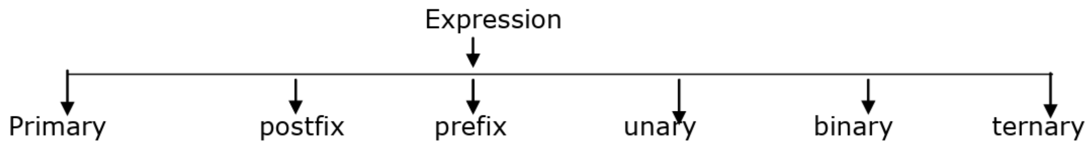
```
c = 5    c = 10    c = 5     c = 25     c = 5      c = 0
```

Expressions:

An expression is a combination of operators and operands which reduces to a single value. An operator indicates an operation to be performed on data that yields a value. An operand is a data item on which an operation is performed.

A simple expression contains only one operator for example 3+5 is a simple expression which yields a value 8, -a is also a single expression. A complex expression contains more than one operator. An example of complex expression is 6+2*7.

An expression can be divided into six categories based on the number of operators, positions of the operands and operators, and the precedence of operator.



Primary Expressions:

In C, the operand in the primary expression can be a Name, a Constant, or a Parenthesized Expression. Name is any identifier for a variable. A constant is the one whose value cannot be changed during program execution. Any value enclosed within parenthesis must be reduced to single value. A complex Expression can be converted into primary expression by enclosing it with parenthesis.

The following is an example,

(3 * 5 + 8);   (c = a = 5 * c);

Postfix Expressions

The postfix expression consists of one operand and one operator. Example: A Function Call, The function name is operand and parenthesis is the operator.

The other examples are post increment and post decrement. In post increment the variable is increased by 1, a++ results in the variable increment by 1.

Similarly in post decrement, the variable is decreased by 1, a-- results in the variable decreased by 1.

Prefix Expressions

Prefix Expressions consists of one operand and one operator, the operand comes after the operator. Examples of prefix expressions are prefix increment and prefix decrement i.e., ++a, --a.

The only difference between postfix and prefix operators is, in the prefix operator, the effect take place before the expression that contains operators is evaluated. It is other way in case of postfix expressions.

Unary Expressions

A unary expression is like a prefix expression consists of one operand and one operator and the operand comes after the operator.

Example: +a; -b; -c; +d;

Binary Expressions

Binary Expressions are the combinations of two operands and an operator. Any two variables added, subtracted, multiplied or divided is a binary expression.

Example: a + b; c * d;

Ternary Expressions

Ternary Expressions is an expression which consists of a ternary operator pair " ? : " In the example, if exp1 is true exp2 is executed else exp3 is executed.

Syntax: exp1 ? exp2 : exp3;

Example 1:

```
#include <stdio.h>
int main()
{
	int n1 = 5, n2 = 10, max;
	max = (n1 > n2) ? n1 : n2;
	printf("Largest number between %d and %d is: %d. ", n1, n2, max);
	return 0;
}
```

Output:

```
Largest number between 5 and 10 is: 10.
```

Example 2:

```
#include <stdio.h>
int main()
{
	int age;
	printf("Enter your age: ");
	scanf("%d", &age);
	(age >= 18) ? printf("You can vote") : printf("You cannot vote");
	return 0;
}
```

Output:

```
Enter your age: 12
You cannot vote
```

Example 3:

```
#include <stdio.h>
int main()
{
	char operator = '+';
	int num1 = 8;
	int num2 = 7;
	int result = (operator == '+') ? (num1 + num2) : (num1 - num2);
	printf("%d", result);
	return 0;
}
```

Output:

```
15
```

PRECEDENCE AND ASSOCIATIVITY:

- Every operator has a precedence value. An expression containing more than one operator is known as a complex expression.
- Complex expressions are executed according to precedence of operators.
- Associativity specifies the order in which the operators are evaluated with the same precedence in a complex expression.
- Associativity is of two ways i.e., left-to-right and right-to-left.
- Left-to-right associativity evaluates an expression stating from left and moving towards right.
- Right-to-left associativity proceeds from right to left.

The precedence and associativity of various operators in C are as shown in the following table,

Operators	**Operation**	**Associativity**	**Precedence**
() [] **->** .	Function call Array expression or square bracket Structure Operator Structure Operator	Left to right	1st
+ _ ++ -- ! ~ * & Sizeof Type	Unary plus Unary minus Increment Decrement Not operator One's complement Pointer operator Address operator Size of an object Type cast	Right to left	2nd
* / %	Multiplication Division Modular division	Left to right	3rd
+ -	Addition Subtraction	Left to right	4th
<< >>	Left shift Right shift	Left to right	5th
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right	6th
== !=	Equality Inequality	Left to right	7th
&	Bitwise AND	Left to right	8th
^	Bitwise XOR	Left to right	9th
|	Bitwise OR	Left to right	10th
&&	Logical AND	Left to right	11th
||	Logical OR	Left to right	12th
?:	Conditional Operator	Right to Left	13th
=, *=, -=, &=, += ^=, |=, <<=, >>=	Assignment Operators	Right to Left	14th
,	Comma Operator	Left to right	15th

TYPE CONVERSION AND CASTING:

Type Conversion:

- In some situations, some variables are declared as integers and while performing an operation on these variables we require the result as floating point type. In such situations we use type conversion.
- The type conversion is used to convert the set of declared data types to some other required types. Conversion can be carried out in 2 ways,
    - Converting by using assignment (Implicit casting).
    - Using cast operator (Explicit casting).

Converting by Assignment:

- The usual way of converting a value from one data type to another is by using the assignment operator.

```
int a;
float b;
a=b;
```

- In this method, if a larger data type value is assigned to smaller data type then the value will be truncated {cut off}. This method is called "Narrowing".

Cast Operator:

- The cast operator is a technique to forcefully to convert one data type to another.
- The operator used to force this conversion is known as "Cast Operator" and the process is known as type casting.

```
Syntax:   x= (cast) expression
          (or)
          X= cast (expression)

Eg:       int a, b;
          float f;
          f= (float) a/b;
          (or)
          f= float (a)/b;
```

Example of Cast Operator:

```
#include <stdio.h>
int main()
{
        int sum= 17, count = 5;
        float mean;
        mean =  (float) sum / count;
        printf("Value of mean : %f\n", mean );
}
```

O/p: *Value of mean : 3.400000*

Explanation: *The **cast operator** has precedence over division, so the value of sum is first converted to **float** data type and finally it gets divided by count yielding a **float value**.*

gets only integer output: wrong	***gets float output: but wrong***	***gets float output w/o CAST: Correct***

#include <stdio.h>
int main()
{
  int sum = 17, count = 5;
  int mean;
  mean = sum / count;
  printf("Value of mean : %d\n", mean);
}
O/p:
Value of mean : 3
 	
The actual output needed is 3.400000, but the result is 3. So to get the correct output one way is to change the data type of a given variable.
	#include <stdio.h>
	int main()
	{
	  int sum = 17, count = 5;
	  float mean;
	  mean = sum / count;
	  printf("Value of mean : %f\n", mean);
	}
	O/p:
	Value of mean : 3.000000
	
	The actual output needed is 3.400000, but the result is 3.000000. So to get the correct output one way is to change the data type of a given variable.
		#include <stdio.h>
		int main()
		{
		  float sum= 17; int count = 5;
		  float mean;
		  mean = sum / count;
		  printf("Value of mean : %f\n", mean);
		}
		O/p:
		Value of mean : 3.400000
		
		The actual output needed is 3.400000, also the result is 3.400000. *Getting the correct output by declaring the correct data type of a given variable. (without TYPE CONVERSION or CASTING)*

PROBLEM SOLVING TECHNIQUES:

An algorithm is a step-by-step procedure to solve a problem. A good algorithm should be optimized in terms of time and space. Different types of problems require different types of algorithmic techniques to be solved in the most optimized manner. There are many types of algorithms but the most important and fundamental algorithms are under an Algorithmic approach.

Algorithmic approach:

There may be several methods under an approach. Example, Sorting is an approach and different sorting techniques are methods each. An algorithm is a sequence of logical steps to solve a problem.

There are many types of algorithms but the most important and fundamental algorithms that you must are discussed in this article.

- Brute Force Algorithm
- Recursive Algorithm
  - Divide and Conquer Algorithm
  - Dynamic Programming Algorithms
  - Greedy Algorithm
  - Backtracking Algorithm
- Randomized Algorithm
- Sorting Algorithm
- Searching Algorithm
- Hashing Algorithm

1. Brute Force Algorithm:

This is the most basic and simplest type of algorithm. A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem. More technically it is just like iterating every possibility available to solve that problem.

Example: If there is a lock of 4-digit PIN. The digits to be chosen from 0-9 then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN. In the worst case, it will take 10,000 tries to find the right combination.

2. Recursive Algorithm:

This type of algorithm is based on recursion. In recursion, a problem is solved by breaking it into subproblems of the same type and calling own self again and again until the problem is solved with the help of a base condition.

Example: Some common problem that is solved using recursive algorithms are Factorial of a Number, Fibonacci Series, Tower of Hanoi, DFS for Graph, etc.

a) Divide and Conquer Algorithm:

In Divide and Conquer algorithms, the idea is to solve the problem in two sections, the first section divides the problem into subproblems of the same type. The second section is to solve the smaller problem independently and then add the combined result to produce the final answer to the problem.

Example: Some common problem that is solved using Divide and Conquers Algorithms are Binary Search, Merge Sort, Quick Sort, Strassen's Matrix Multiplication, etc.

b) Dynamic Programming Algorithms:

This type of algorithm is also known as the memoization technique because in this the idea is to store the previously calculated result to avoid calculating it again and again. In Dynamic Programming, divide the complex problem into smaller overlapping subproblems and store the result for future use.

Example: The following problems can be solved using the Dynamic Programming algorithm Knapsack Problem, Weighted Job Scheduling, Floyd Warshall Algorithm, etc.

c) Greedy Algorithm:

In the Greedy Algorithm, the solution is built part by part. The decision to choose the next part is done on the basis that it gives an immediate benefit. It never considers the choices that had been taken previously.

Example: Some common problems that can be solved through the Greedy Algorithm are Dijkstra Shortest Path Algorithm, Prim's Algorithm, Kruskal's Algorithm, Huffman Coding, etc.

d) Backtracking Algorithm:

In Backtracking Algorithm, the problem is solved in an incremental way i.e., it is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.

Example: Some problems can be solved through the Backtracking Algorithm are the Hamiltonian Cycle, M-Coloring Problem, N Queen Problem, Rat in Maze Problem, etc.

3. Randomized Algorithm:

In the randomized algorithm, we use a random number.it helps to decide the expected outcome. The decision to choose the random number so it gives the immediate benefit

Example: Some common problems that can be solved through the Randomized Algorithm are Quicksort: In Quicksort we use the random number for selecting the pivot.

4. Sorting Algorithm:

The sorting algorithm is used to sort data in maybe ascending or descending order. It's also used for arranging data in an efficient and useful manner.

Example: Some common problems that can be solved through the sorting Algorithm are Bubble sort, insertion sort, merge sort, selection sort, and quick sort are examples of the Sorting algorithm.

5. Searching Algorithm:

The searching algorithm is the algorithm that is used for searching the specific key in particular sorted or unsorted data.

Example: Some common problems that can be solved through the Searching Algorithm are Binary search or linear search is one example of a Searching algorithm.

6. Hashing Algorithm:

Hashing algorithms work the same as the Searching algorithm but they contain an index with a key ID i.e. a key-value pair. In hashing, we assign a key to specific data.

Example: Some common problems can be solved through the Hashing Algorithm in password verification.

Characteristics of an Algorithm:

There are some important characteristics which every algorithm should follow. They are as follows:

1. Clear and Unambiguous:
The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

2. Well-Defined Inputs:
If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.

3. Well-Defined Outputs:
The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.

4. Finiteness:
The algorithm must be finite, i.e., it should terminate after a finite time. An algorithm must terminate after a finite number of steps in all test cases.

5. Feasible:
The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.

6. Language Independent:
The Algorithm designed must be language-independent, i.e., it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

7. Effectiveness:
An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

8. Modularity and reusability:
Algorithms may be modular, they may be divided into smaller subproblems or features that may be reused in unique parts of the algorithm or in other algorithms.

9. Understandability:
Algorithms need to be designed with clarity and ease in mind, making them easy to apprehend and implement.

PROBLEM SOLVING STRATEGIES: .

Structured programming is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines. It is possible to do structured programming in any programming language, though it is preferable to use something like a procedural programming language. Example: ALGOL, Pascal, PL/I and Ada.

There are three main principles of structured programming,

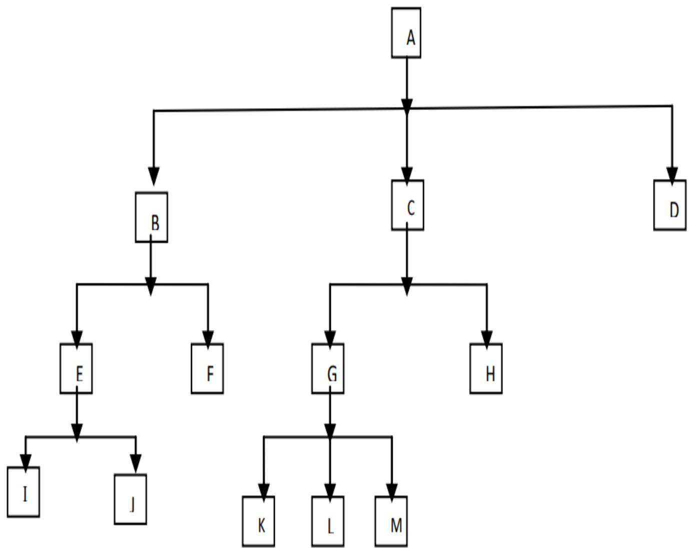
1. Program design using ***top-down or bottom-up approach***
2. Decomposition of program into components i.e. modular programming
3. Structuring of control flow

Top-down approach:

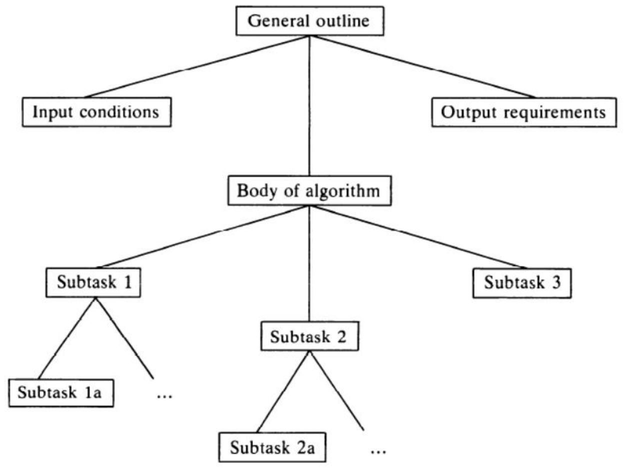
- Program design concentrates on planning the solution as a collection of sub solutions. During top-down design the divide and conquer policy is followed.
- The problem is divided into smaller sub problems and these sub problems are further divided into even smaller sub problems.
- In the top-down approach, the calling components is always designed before its sub component.
- Thus top-down design represents a successive refinement of functionality of the program.

- It allows us to build solutions to a problem in a stepwise fashion.

For example, the following figure shows top-down approach,



(or)



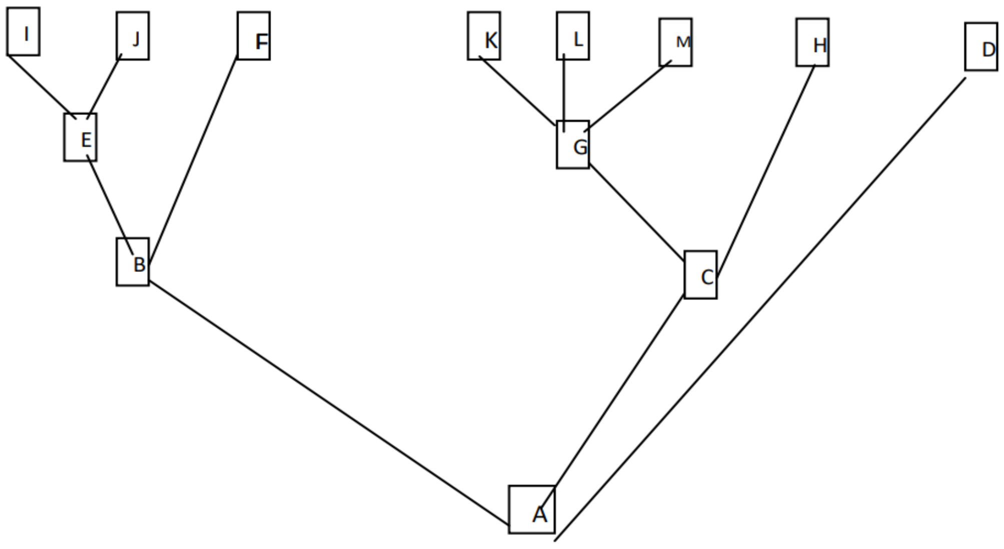
g. 1.1 Schematic breakdown of a problem into subtasks as employed in top-down design.

Bottom-up approach:

The bottom-up design is the reverse of the top-down approach.

- Here the process starts with the identification of the smallest sub component of the total program which can be easily implemented.
- Such smallest components are combined to reach to a more abstract level and plan components of the higher level.
- This process is continued till the complete program does not get realized.
- The main drawback of this approach is that it is rarely possible to identify smallest sub component needed for the program, especially for bigger program.

For example, the following figure shows bottom-up approach,



Deference between top-down and bottom-up approaches are,

Sl.	TOP DOWN APPROACH	BOTTOM UP APPROACH
1.	In this approach We focus on breaking up the problem into smaller parts.	In this approach, we solve smaller problems and integrate it as whole and complete the solution.
2.	Mainly used by structured programming language such as COBOL, Fortran, C, etc.	Mainly used by object oriented programming language such as C++, C#, Python.
3.	Each part is programmed separately therefore contain redundancy.	Redundancy is minimized by using data encapsulation and data hiding.
4.	In this the communications is less among modules.	In this module must have communication.
5.	It is used in debugging, module documentation, etc.	It is basically used in testing.
6.	In top down approach, decomposition takes place.	In bottom up approach composition takes place.
7.	In this top function of system might be hard to identify.	In this sometimes we can not build a program from the piece we have started.
8.	In this implementation details may differ.	This is not natural for people to assemble.

Time and Space complexities of Algorithms: .

An algorithm is defined as complex based on the amount of Space and Time it consumes. Hence the Complexity of an algorithm refers to the measure of the time that it will need to execute and get the expected output, and the Space it will need to store all the data (input, temporary data, and output). Hence these two factors define the efficiency of an algorithm.

The two factors of Algorithm Complexity are:

- **Time Factor**: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor**: Space is measured by counting the maximum memory space required by the algorithm to run/execute.

Therefore the complexity of an algorithm can be divided into two types:

1. Space Complexity: The space complexity of an algorithm refers to the amount of memory required by the algorithm to store the variables and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?

The space complexity of an algorithm is calculated by determining the following 2 components:

- **Fixed Part:** This refers to the space that is required by the algorithm. For example, input variables, output variables, program size, etc.
- **Variable Part:** This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

Therefore, Space complexity **S(P)** of any algorithm P is **S(P) = C + SP(I)**, where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

2. Time Complexity: The time complexity of an algorithm refers to the amount of time required by the algorithm to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

How to Calculate, Time Complexity?

The time complexity of an algorithm is also calculated by determining the following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, arithmetic operations, etc.
- **Variable Time Part:** Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.

Therefore, Time complexity of any algorithm P is **T(P) = C + TP(I)**, where C is the constant time part and TP(I) is the variable part of the algorithm, which depends on the instance characteristic I.

The analysis of Algorithms:

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

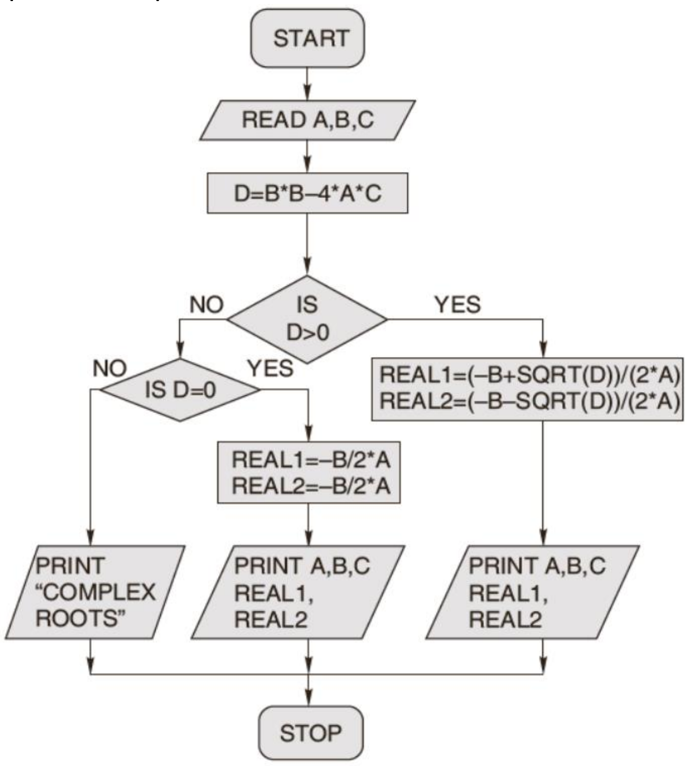
Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance.

Generally, we perform the following types of analysis –

- **Worst-case** − The maximum number of steps taken on any instance of size a.
- **Best-case** − The minimum number of steps taken on any instance of size a.
- **Average-case** − An average number of steps taken on any instance of size a.
- **Amortized** − A sequence of operations applied to the input of size an averaged over time.

Some Important sort outs:

Problem: Draw a flowchart to find the roots of a quadratic equation



Problem: Write the PSEUDO CODE for checking whether a given number is a prime number or not.

```
START
PRINT "ENTER THE NUMBER"
INPUT N
IF N = 2 THEN
 PRINT "CO-PRIME" GOTO STEP 12
D ← 2
Q ← N/D (Integer division)
R ← N – Q*D
IF R = 0 THEN GOTO STEP 11
D ← D + 1
IF D <= N/2 THEN GOTO STEP 6
IF R = 0 THEN
 PRINT "NOT PRIME"
ELSE
 PRINT "PRIME"
STOP
```

UNIT 2

Control Structures

Simple sequential programs, Conditional Statements (if, if-else, Switch). Loops (for, while, do-while), Break and Continue.

Simple Sequential Programs:

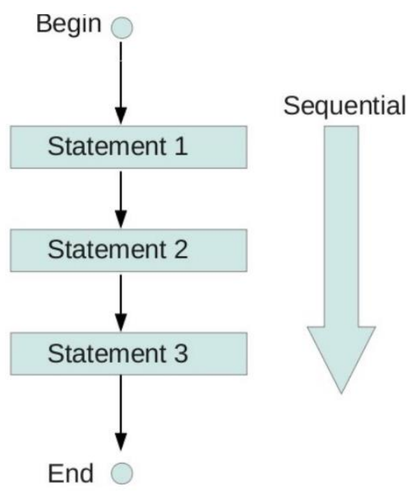
Sequence of statements are written in order to accomplish a specific activity. So statements are executed in the order they are specified in the program. This way of executing statements sequentially is known as Sequential control statements.

There is an advantage that is no separate control statements are needed in order to execute the statements one after the other.

Disadvantage is that there is no way to change the sequence. The solution for this is branching and loop control structures.

Example: / * Program to illustrate sequential flow of execution */

```
#include <stdio.h>
#include <conio.h>
int main()
{
        float farenhite, celsius;

        printf("Enter degree in farenhite\n");
        scanf("%f", &farenhite);

        celsius=(float)5/9*(farenhite-32);

        printf("result=%f\n", celsius);
        getch();
        return 0;
}
```



Explanation: Execution begins from the main. We enter a farenhite degree value. Control moves to the formula which will convert farenhite into celsius and then equivalent celsius value is output.

Some more Examples:

/* Prog. to print Multiplication Table */	***/* Prog. to do arithmetic operations */***	***/* Prog. to scan diff. i/p and print o/p */***

#include <stdio.h>
int main()
{
    int n;
    printf("Which Mul. Table to print: “);
    scanf("%d", &n);
    printf(“ 1 * %d = %d \n”, n, 1*n);
    printf(“ 2 * %d = %d \n”, n, 2*n);
    printf(“ 3 * %d = %d \n”, n, 3*n);
    printf(“ 4 * %d = %d \n”, n, 4*n);
    printf(“ 5 * %d = %d \n”, n, 5*n);
    return(0);
}
	#include<stdio.h>
	int main()
	{
	    int a,b,c;
	    printf(“enter a,b values\n”);
	    scanf(“%d%d”,&a,&b);
	    c=a+b;
	    printf(“sum=%d\n”,c);
	    c=a-b;
	    printf(“sub=%d\n”,c);
	    c=a*b;
	    printf(“mul=%d\n”,c);
	    c=a/b;
	    printf(“div=%d\n”,c);
	    c=a%b;
	    printf(“mod=%d\n”,c);
	    return 0;
	}
		#include <stdio.h>
		int main()
		{
		    int num;
		    char ch;
		    float f;
		    printf("Enter the integer: ");
		    scanf("%d", &num);
		    printf("\nEntered integer is: %d", num);
		    printf("\n\nEnter the float: ");
		    scanf("%f", &f);
		    printf("\nEntered float is: %f", f);
		    printf("\n\nEnter the Character: ");
		    scanf("%c", &ch);
		    printf("\nEntered integer is: %c", ch);
		    return 0;
		}

Conditional Statements (if, if-else, switch):

Selection Control Statements also called as Decision control statements and Conditional Statements allow the computer to take decision. And force to work under given condition. A Selection Control Statement gives the control to the computer for taking the decisions.

Four selection control instruction which are implemented in C are following:

a) if statement

b) if-else statement

c) Nested if-else statement

d) if-else Ladder

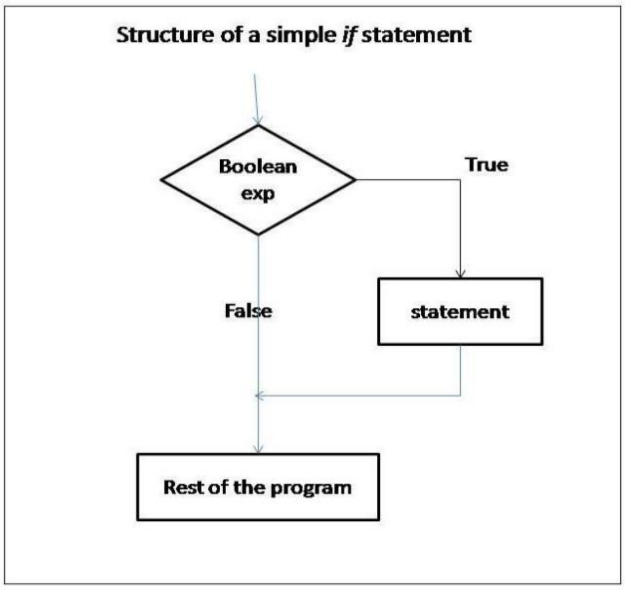
e) switch

a) Simple if statement:

The simple if (or) if only statement takes care of true condition only. It has only one block.

Syntax:

```
        if (condition is true)
        {

            block of executable Statements;

        }
```



Example:

```
    #include<stdio.h>
    #include<conio.h>
    int main( )
    {
        int m,n;
        clrscr();
        printf(“Enter any two number: ”);
        scanf(“%d %d”, &m, &n);

        if( m-n == 0 )
        {
            printf(“ The entered numbers are equal. \n”);
        }

        getch();
        return 0;
    }
```

Output:

```
        Enter any two number: 50 50
        The entered numbers are equal.
```

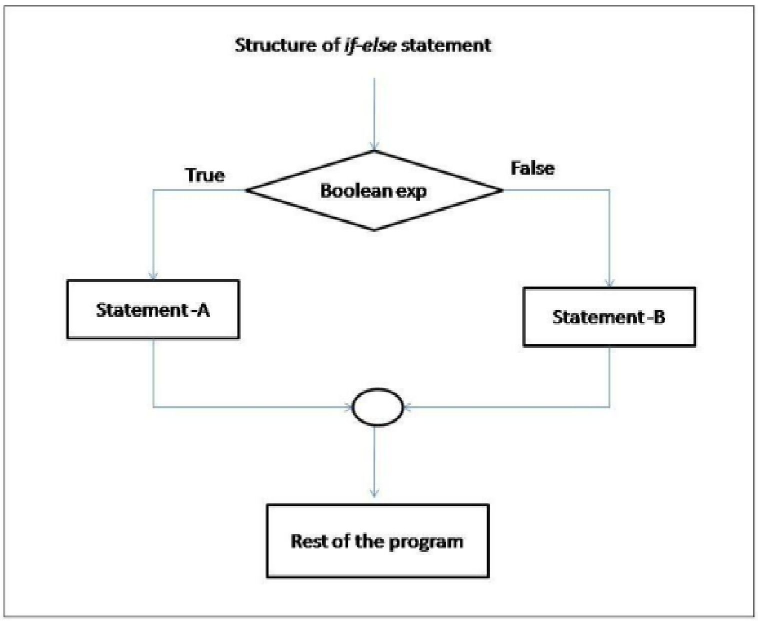
b) if-else statement:

The if-else statement is an extension of the if statement. The if-else statement takes care of true as well as false conditions. It has two blocks.

- One block is for if and it is executed when the condition is true.
- The other block is of else and it is executed when the condition is false.
- The else statement cannot be used without if.
- No multiple else statements are allowed without one if.

Syntax:

```
if(expression is true)
{
        Statement-A;
}
else
{
        Statement-B;
}
```



Example:

```
#include<stdio.h>
#include<conio.h>
int main( )
{
        int age;
        clrscr( );
        printf("Enter candidate age: \n");
        scanf("%d", &age);

        if( age >= 18 )
            printf(" Eligible to cast vote \n");
        else
            printf(" Not Eligible to cast vote "\n);

        getch();
        return 0;
}
```

Output:

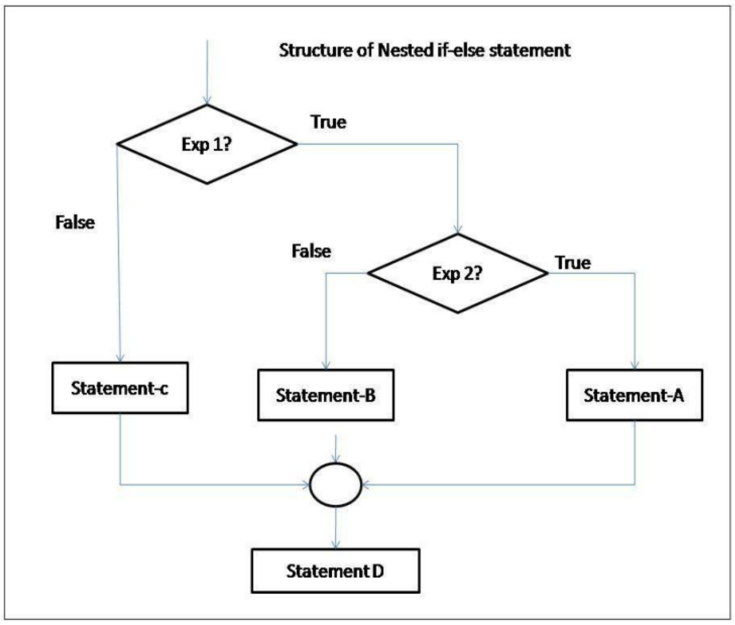
```
Enter candidate age: 20
Eligible to cast vote

Enter candidate age: 17
Not Eligible to cast vote
```

c) Nested if-else statement:

The numbers of Logical conditions are checked for executing various statements by using nested if-else statements.

Syntax:

```
if(condition)
{
        if(condition)
        {
                Statement-A;
        }
        else
        {
                Statement-B;
        }
}
else
{
        Statement-C;
}

… … …
Statement-D;
… … …
```



Example: Write a program to find largest number among three numbers using nested if.

```
#include<stdio.h>
int main()
{
        int x, y, z;
        printf("\n Enter the values of x, y, z: ");
        scanf("%d %d %d", &x, &y, &z);
        printf("\n Largest among three numbers is: ");
        if( x > y )
        {
                if( x > z )
                        printf(" x = %d \n", x);
                else
                        printf(" z = %d \n", z);
        }
        else
        {
                if( z > y )
                        printf(" z = %d \n", z);
                else
                        printf(" y = %d \n", y);
        }
        return 0;
}
```

Output:

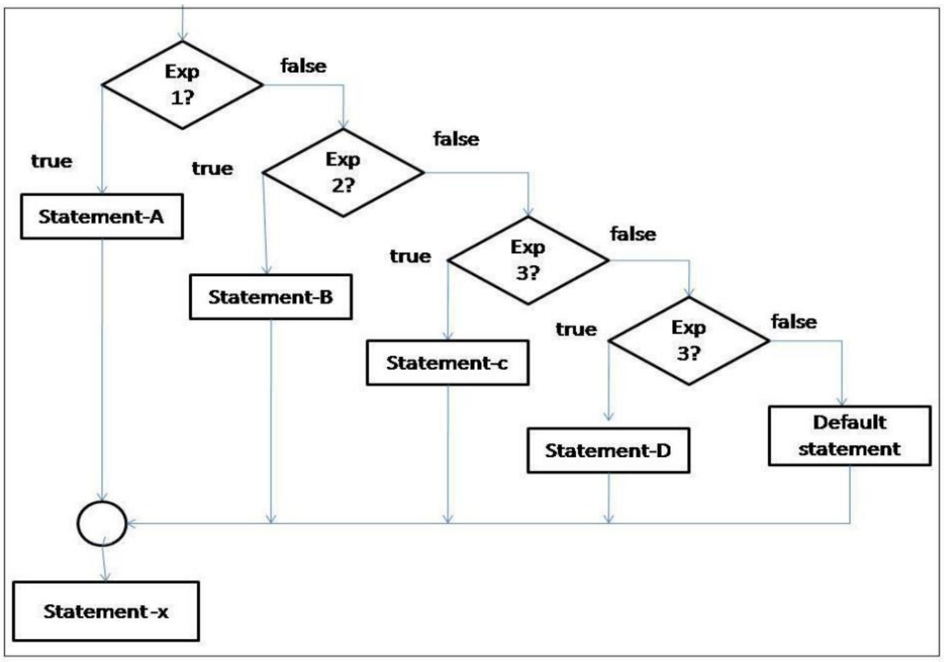
```
Enter three numbers x, y, z: 56 89 78
Largest out of three numbers is: y = 89
```

d) if-else Ladder statement:

There is another way of putting if's together when multipath decisions are involved to solve the given task. A multipath decision is a chain of if's in which the statement associated with each else is an if.

Syntax:

```
if(condition-1)
      Statement-1;
else if(condition-2)
      Statement-2;
else if(condition-3)
      Statement-3;
…  …  …
…  …  …
else if(condition-n)
      Statement-n;
else
      Default statement;
…  …  …
Statement-x;
…  …  …
```



Example: Write a C program to find the average of six subjects and display the results as follows,

AVERAGE	RESULT .
< 40	FAIL
>=40 & <50	THIRD CLASS
>=50 & <60	SECOND CLASS
>=60 & <75	FIRST CLASS
>=75 & <100	DISTINCTION

```
#include<stdio.h>
int main()
{
        int sum=0, tel, eng, math, hin, pys, chem;
        float avg;
        printf("\n: Enter marks :\n Telugu: \nEnglish: \nMaths: \nHindi: \nPhysics: \nChemistry: \n\n");
        scanf("%d%d%d%d%d%d", &tel, &eng, &math, &hin, &pys, &chem);
        sum=tel+eng+math+hin+pys+chem;
        avg=sum/6;
        printf("Total : %d \n Average: %.2f", sum, avg);

        if( tel<40 || eng<40 || math<40 || hin<40 || pys<40 || chem<40 )
        {       printf("\n Result: Fail");              }
        else if( avg>=40 && avg<50 )
        {       printf("\n Result: Third Class");       }
        else if( avg>=50 && avg<60 )
        {       printf("\nResult: Second Class");  }
        else if( avg>=60 && avg<75 )
        {       printf("\n Result: First Class");       }
        else if( avg>=75 && avg<100 )
        {       printf("\nResult: Distinction");        }

return(0);
}
```

<u>*Output:*</u>

```
        : Enter marks :
        Telugu: 96
        English: 86
        Maths: 77
        Hindi: 66
        Physics: 65
        Chemistry: 88

        Total: 478
        Average: 79.00

        Result: Distinction
```

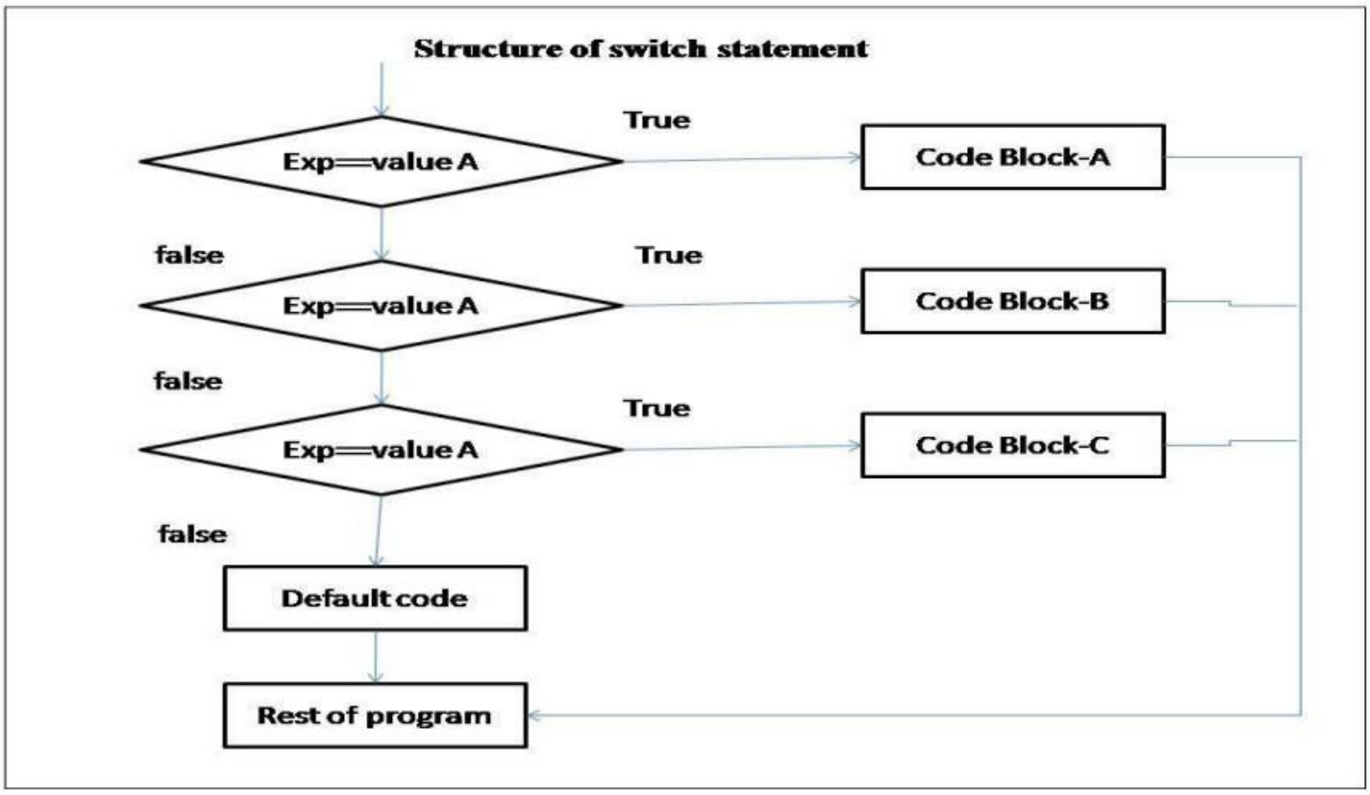
e) switch statement (or) Case control (or) switch()…case:

The Case control statements allow the computer to take decision as to be which statements are to be executed next.

- It is a multi-way decision construct facilitate number of alternatives has multi way decision statement known as "switch statement".
- First, in the option parentheses - give the condition. This condition checks to match, one by one with case constant.
- If value match then its statement will be executed.
- Otherwise, the default statement will appear.
- Every case statement terminates with " **:** "

<u>Syntax:</u>

```
        switch(option)
        {
                case option_1 :     statements;
                                    break;

                case option_2 :     statements;
                                    break;
                … … …
                … … …

                case option_n :     statements n;
                                    break;

                default :           statements;
                                    break;
        }
```



Example 1:

1. Write a program to convert years into Minutes, Hours, Days, Months, and Seconds using switch statements.

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

  printf("\n [1] MINUTES");
  printf("\n [2] HOURS");
  printf("\n [3] DAYS");
  printf("\n [4] MONTHS");
  printf("\n [5] SECONDS");
  printf("\n [0] EXIT");

  printf("\n Enter your choice:");
  scanf("%ld", &ch);

  if( ch>0 && ch<6 )
  {
      printf(" Enter how many years: ");
      scanf("%ld", &yrs);
  }
  mon=yrs*12;
  ds=mon*30;
  ds=ds+yrs*5;
  hrs=ds*24;
  min=hrs*60;
  se=min*60;
```

2. Write a program to convert Decimal into Words.

```
#include<stdio.h>
int main()
{
     int n;
     printf("\n: Enter a Digit 10 – 15: ");
     scanf("%d", &n);
     switch(n)
     {
          case 10 : printf(" Ten ");
                    break;
          case 11 : printf(" Eleven ");
                    break;
          case 12 : printf(" Twelve ");
                    break;
          case 13 : printf(" Thirteen ");
                    break;
          case 14 : printf(" Fourteen ");
                    break;
          case 15 : printf(" Fifteen ");
                    break;
          default :  printf("Invalid. Enter 10-15 only.\n");
                    break;
      }
      return 0;
}
```

```
Output:          Enter a Digit 10 – 15: 14
                 Fourteen
```

```
  switch(ch)
  {
      case 1 : printf("\n MINUTES: %ld", min);
                  break;
      case 2 : printf("\n Hours: %ld", hrs);
                  break;
      case 3 : printf("\n Days: %ld", ds);
                  break;
      case 4 : printf("\n Months: %ld", mon);
                  break;
      case 5 : printf("\n seconds: %ld", se);
                  break;
      case 0 : printf("\n Exe. Terminated ");
                  exit();
                  break;
      default : printf("\n Invalid choice ");
                  break;
  }
  getch();
  return 0;
}
```

Output:

```
              [1] MINUTES
              [2] HOURS
              [3] DAYS
              [4] MONTHS
              [5] SECONDS
              [0] EXIT
              Enter your choice: 2
              Enter how many years: 1
              Hours: 8760
```

2. Write a program to print 'a' as "A is for apple", ….. …..

```
#include<stdio.h>
int main()
{
    char ch;
    printf("\n: Enter a Char a – e: ");
    scanf("%c", &ch);
    switch(ch)
    {
        case 'a' :  printf(" A is for Apple. ");
                    break;
        case 'b' :  printf(" B is for Ball ");
                    break;
        case 'c' :  printf(" C is for Cat ");
                    break;
        case 'd' :  printf(" D is for Dog ");
                    break;
        case 'e' :  printf(" E is for Elephant ");
                    break;

         …       …       …       …
         …       …       …       …

         case 'z' : printf(" Z is for Zebra ");
                    break;
        default :   printf(" Invalid. Enter a-z only. \n");
                    break;
     }
     return 0;
}

Output:          Enter a Char a – e: a
                 A is for Apple.
```

Loops (for, while, do-while):

ITERAION CONTROL STATEMENTS:

Iteration Control Statements also called as Repetition or Loop Control Statements. This type of statements helps the computer to execute a group of statements repeatedly. This allows a set of instruction to be performed until a certain condition is reached.

What is a loop?

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

There are three types of loops in C:

a) for loop

b) while loop

c) do-while loop

a) The for loop:

There are three parts / portions / blocks of for loop:

a) Counter initialization.

b) Check condition.

c) Modification of counter.

Syntax:

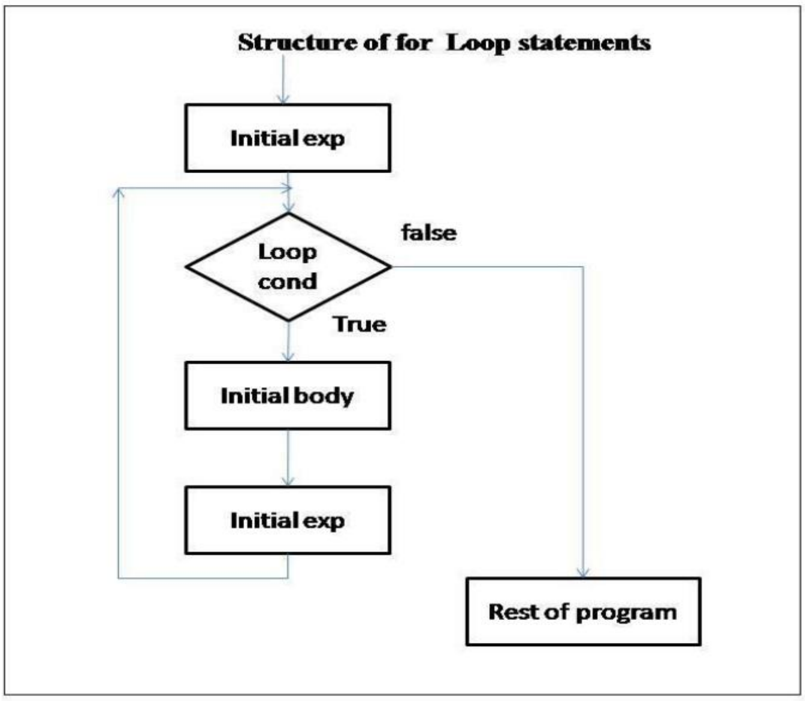
```
for (variable initialize; check condition; modify counter)
{
        statements 1;
        -----------;
        -----------;
        statements n;
 }
```

Explanation:

1. The initialization is usually an assignment that is used to set the loop control variable.
2. The condition is a relational expression that determines when the loop will exit.
3. The modify counter defines how loop control variables will change each time the loop is repeated.

* These three sections are separated by semicolon (;).
* The for loop is executed as long as the condition is true. When, the condition becomes false the program execution will resume on the statement following the block.
* Advantage of for loop over other loops:

  All three parts of for loop (i.e., counter initialization, check condition, modification of counter) are implemented on a single line.



Some more usage format about for loop:

Sl	Usage	Meaning
1.	for (p=1, n=2; n<17; n++)	we can assign multiple variables together in for loop.
2.	for (n=1, m=50; n<=m; n=n+1, m=m-1)	The increment section may also have more than one part as given.
3.	for (i=1, sum=0; i<20 && sum<100; ++I)	The test condition may have any compound relation as given.
4.	for (x=(m;n)/2; x>0; x=x/2)	It is also permissible to use expressions in the assignment statements of initialization and increment section as given.
5.	for (; m!=100 ;)	we can omit the initialization and increment section to set up time delay.
6.	for (; ;)	Infinite loop. (use ctrl + break to quit execution)
7.	for (p=1; n<=17; n++)**;**	The for loop will iterate only one time. So, it is not advisable.

Example 1: ***Write a C program to print 1 upto 10 nos using for loop.***

```
#include <stdio.h>
int main()
{
        int counter;
        for (counter =1; counter<=10; counter++)
        {
                printf("%d \n", counter);
        }
        return 0;
}
```

Output:

```
                1  2  3  4  5  6  7  8  9  10
```

Example 2: ***Write a C program to print 10th Multiplication Table upto 10 using for loop.***

```
#include<stdio.h>
int main()
{
        int i, n;
        printf(“ Enter which table to print: “);
        scanf(“%d”, &n);
        for (i =1; i<=10; i++)
        {
                printf(" %d * %d = %d \n", n, i, n*i);
        }
        return 0;
}
```

Output:

```
10 * 1 = 10
10 * 2 = 10
10 * 3 = 10
10 * 4 = 10
10 * 5 = 10
10 * 6 = 10
10 * 7 = 10
10 * 8 = 10
10 * 9 = 10
10 * 10 = 100
```

b) while loop:

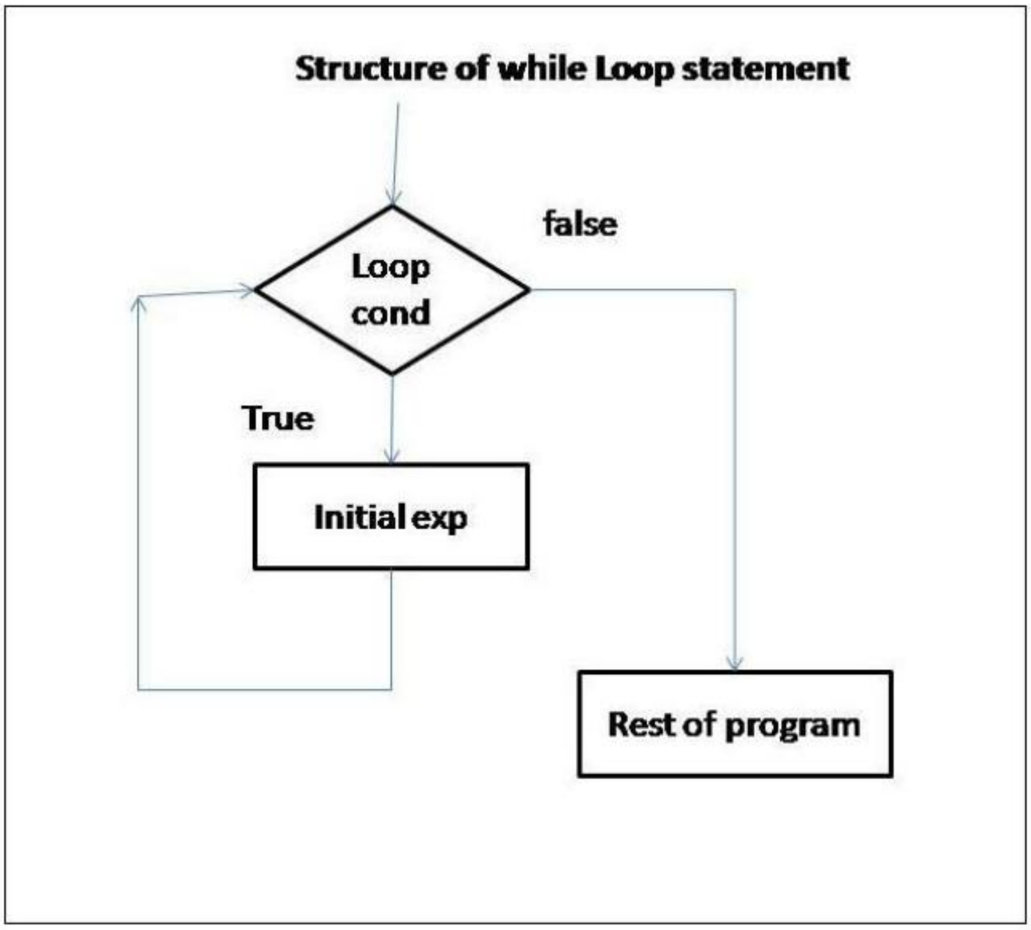
It is a primitive type looping control because it repeats the loop a fixed no. of time. It is also called entry-controlled loop statements.

Syntax:

```
while (test condition)
{
        body of loop;
}
```

Explanation:

The test condition is evaluated if the condition is true, the body of loop will be executed.



Example 1: **Write a program to print the entered number in reverse order?**

```
#include<stdio.h>
int main()
{
        int i=1, n, d, rev;

        printf(" Enter the number of digits: ");
        scanf("%d", &d);

        printf("\n Enter the number which is to be reversed: ");
        scanf("%d", &n);
```

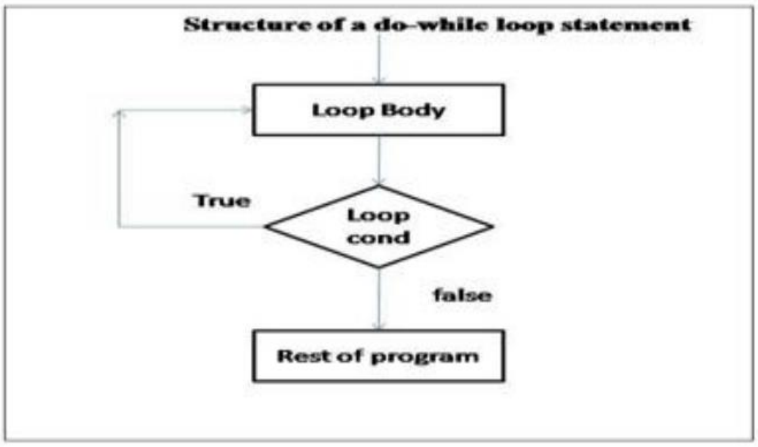
```
            printf("\n The reversed number is: ");
                                                        /*   Compare with for loop
            while( i <= d )                             for( i=1; i<=d; i++)
            {                                           {
                    rev = n%10;                                 rev = n%10;
                    printf("%d", rev);                          printf("%d", rev);
                    n = n/10;                                   n = n/10;
                    i++;                                }
            }                                                                       */

            return(0);
    }
```

Output:

```
            Enter the number of digits: 4
            Enter the number which is to be reversed: 7896
            The reversed number is: 6987

            Enter the number of digits: 4
            Enter the number which is to be reversed: 3692
            The reversed number is: 2963
```

c) do-while loop:

The minor Difference between the working of while and do-while loop is the place where the condition is tested. The while tests the condition before executing any of the statements within the while loop. As against this, the do-while loop tests the condition after having executed the statement within the loop.



syntax:

```
            do
            {
                    … … …
                    body of loop;
                    … … …

            } while (test condition);
```

Explanation:

It first executes the body of the loop, and then evaluates the test condition. If the condition is true, the body of loop will be executed again and again until the condition becomes false.

Example 1: ***Write a program to find the numbers using do while loop.***

```
    #include<stdio.h>
    #include<math.h>
    int main()
    {
            int y, x=1;
            printf("\n Print the numbers and their cubes ");
            printf(" \n ========================");
```

```
        do
        {
                y = pow(x, 3);
                printf("%5d %27d\n", x, y);
                x++;
        } while( x <= 10 );

        return(0);
}
```

Output:

```
        Print the numbers and their cubes
        ==========================
        1       1
        2       8
        3       27
        4       64
        5       125
        6       216
        7       343
        8       512
        9       729
        10      1000
```

Break and Continue:

Jumping Out of Loops:

In various scenarios, you need to either exit the loop or skip an iteration of loop when certain condition is met. So, in those scenarios are known as jumping out of the loop. There are two ways in which you can achieve the same.

i) break statement:

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, passing control to the first statement beyond the loop or a switch. With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

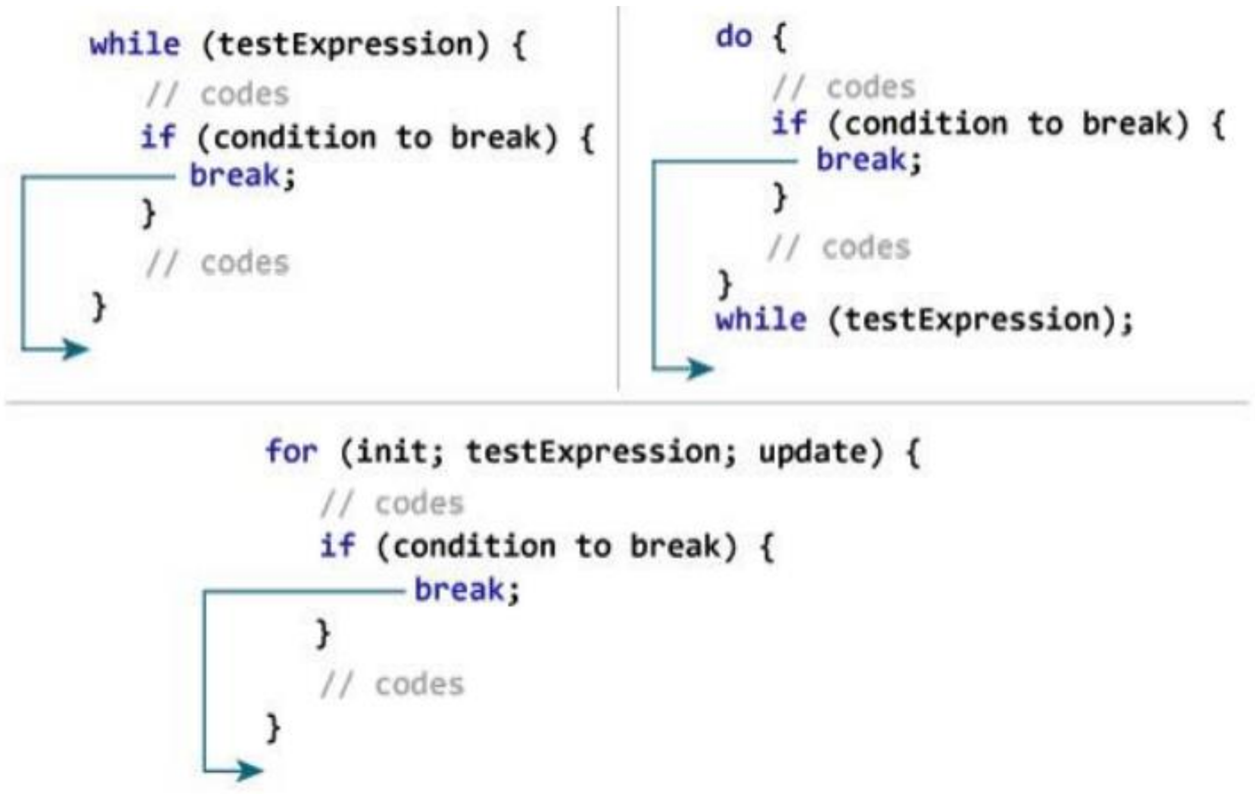
When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

In case of nested loop, if the break statement is encountered in the inner loop then inner loop is exited.

<u>Syntax:</u>

```
Statements;
… … …
break;
```

It can be used with for, while, do-while and switch statements.



```
while (testExpression) {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
```

```
do {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
   // codes
   if (condition to break) {
      break;
   }
   // codes
}
```

<u>Example: Write a C program to explain the use of " break " statement.</u>

```
#include<stdio.h>
int main()
{
        int counter;
        for (counter=1; counter<=10; counter++)
        {
                if(counter == 5)
                {
                        break;
                }
                printf("%d \n", counter);
        }
        return 0;
}
```

<u>Output:</u>

```
1  2  3  4
```

<u>ii) continue statement:</u>

Continue Statement sends the control directly to the test-condition and then continue the loop process. On encountering continue keyword, execution flow leaves the current iteration of loop, and starts with the next iteration.

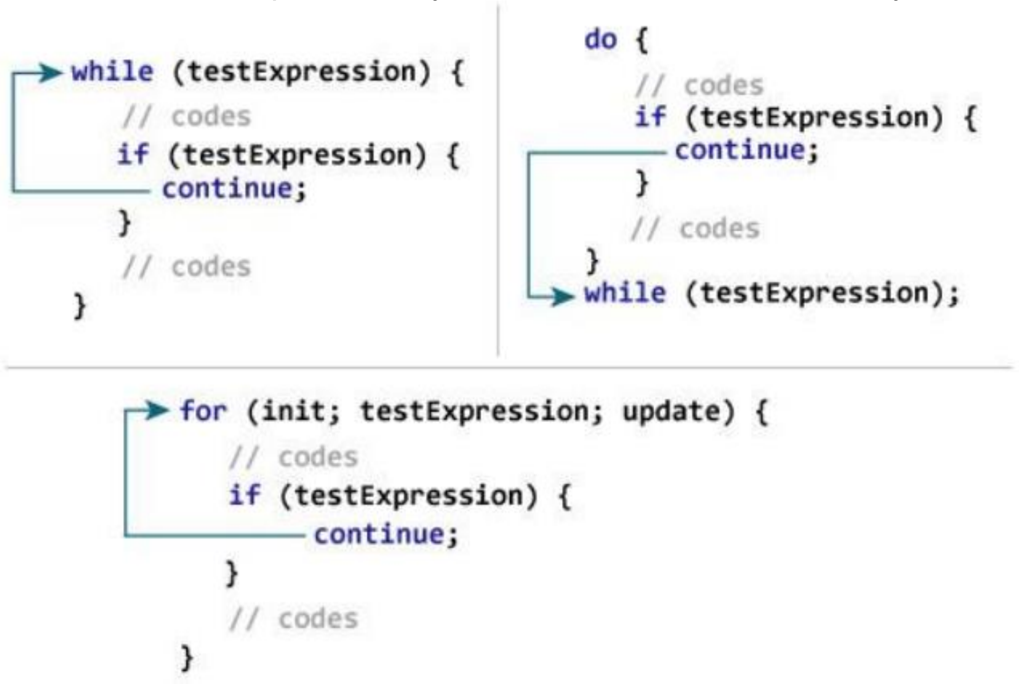
This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

<u>Syntax:</u>

```
statement 1;
continue;
statement 2;
```

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration (looping).

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.



```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

<u>Example: Write a C program to explain the use of “ continue “ statement.</u>

```
#include<stdio.h>
int main()
{
        int i;
        for ( i = 1; i <= 10; i++ )
        {
                if (i == 5)                    /*  If i is equals to 6, continue to next iteration
                {   continue;        }              without printing i value.                      */
                else
                {   printf(" %d ", i);   }     // otherwise print the value of i
        }
        return 0;
}
```

<u>Output:</u>

```
1   2   3   4   6   7   8   9   10
```

UNIT 3

Arrays and Strings

Arrays indexing, memory model, programs with array of integers, two dimensional arrays, Introduction to Strings.

ARRAYS & INDEXING: .

Introduction:

So far, we have used only single variable name for storing one data item. If we need to store multiple copies of the same data then it is very difficult for the user. To overcome the difficulty a new data structure is used called arrays.

- An array is a linear and homogeneous data structure
- Similar type of elements is stored contiguously in memory under one variable name.
- An array can be declared of any standard or custom data type.

[So far, we have used only the fundamental data types, namely char, float, int, double and variations of int and double. A variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data.

To Process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. Array is a secondary or Derived Data type. At some important areas the concept arrays are used,

- *To maintain list of employees in a company.*
- *In pharmacy to maintain list of Drugs and their cost.*
- *To maintain test scores of a class of students.*]

Definition of an Array:

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. ***(or)***

Array in C can be defined as a method of clubbing multiple entities of similar type into a larger group in contiguous memory. These entities or elements can be of int, float, char, or double data type or can be of user-defined data types. ***(or)***

An array is a data structure that stores a collection of similar data type elements, each identified by an index. The elements in an array are stored in contiguous memory locations, and the index is used to access a specific element within the array. Arrays are commonly used in computer programming to organize and manipulate sets of related data. ***(or)***

Array is defined as the set of same data items which are stored in contiguous memory by sharing same variable name with different index positions. Array permits to declare of any one of the primitive data types. Array index always starts with zero.

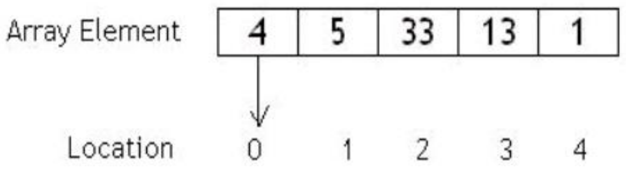
Example:

```
        int a[10];      float a[10];        double a[10];       char str[25];
```

[Suppose we have to store the roll numbers of the 100 students the we have to declare 100 variables named as roll1, roll2, roll3, ……. roll100 which is very difficult job. Concept of C programming arrays is introduced in C which gives the capability to store the 100 roll numbers in the contiguous memory which has 100 blocks and which can be accessed by single variable name.

1. C Programming Arrays is the Collection of Elements
2. C Programming Arrays is collection of the Elements of the same data type.
3. All Elements are stored in the Contiguous memory
4. All elements in the array are accessed using the subscript variable (index).

<u>Pictorial representation of C Programming Arrays</u>



The above array is declared as ***int a[5];***

i.e., a[0] = 4; a[1] = 5; a[2] = 33; a[3] = 13; a[4] = 1;

In the above figure 4, 5, 33, 13, 1 are <u>actual data</u> items. 0, 1, 2, 3, 4 are <u>index</u> variables.

<u>Index or Subscript Variable:</u>

1. Individual data items can be accessed by the name of the array and an integer enclosed in square bracket called subscript variable / index

2. Subscript Variables helps us to identify the item number to be accessed in the contiguous memory.

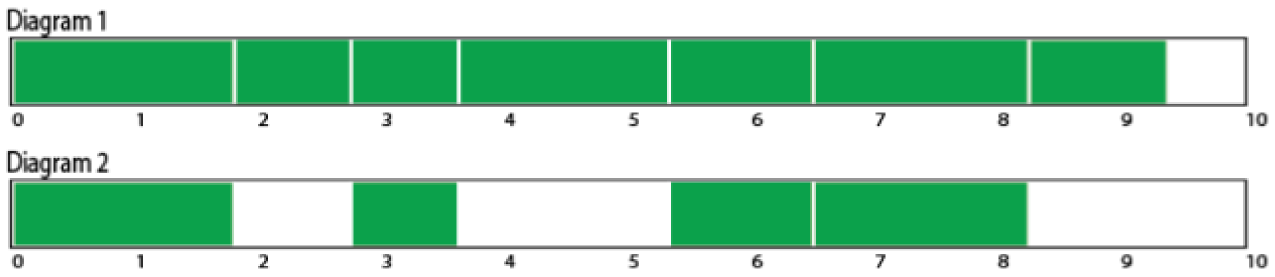
<u>What is Contiguous Memory?</u>

1. When Big Block of memory is reserved or allocated then that memory block is called as Contiguous Memory Block.
2. Alternate meaning of Contiguous Memory is continuous memory.
3. Suppose inside memory we have reserved 1000-1200 memory addresses for special purposes then we can say that these 200 blocks are going to reserve contiguous memory.

<u>Contiguous Memory Allocation:</u>

1. Two registers are used while implementing the contiguous memory scheme. These registers are base register and limit register.
2. When OS is executing a process inside the main memory then contents of each register are as,

Registers	***Content of registers***
1. Base register	*Starting address of the memory location where process execution is happening*
2. Limit register	*Total amount of memory in bytes consumed by process*



Here, <u>diagram 1</u> represents the <u>contiguous allocation of memory</u> and diagram 2 represents non-contiguous allocation of memory.

3. When process try to refer a part of the memory then it will firstly refer the base address from base register and then it will refer relative address of memory location with respect to base address.

<u>How to allocate contiguous memory?</u>

1. Using static array declaration.
2. Using calloc( ) / malloc( ) function to allocate big chunk of memory dynamically.

<u>Array Terminologies:</u>

1) Size *: Number of elements to store in an array. It is always mentioned in square brackets [ ]*
2) Type *: Refers to data type. It decides which type of element is stored in the array. It is also instructing the compiler to reserve memory according to the data type.*
3) Base *: The address of the first element is a base address. The array name itself stores address of the first element.*
4) Index *: The array name is used to refer to the array element. For example int num[x], num is array and x is index. The value of x begins from 0. The index value is always an integer value.*
5) Range *: Value of index of an array varies from lower bound to upper bound. For example, int num[100]; the range of index is 0 to 99.*

<u>Characteristics of an array:</u>

1. The declaration int a[5] is nothing but creation of five variables of integer types in memory instead of declaring five variables for five values.
2. All the elements of an array share the same name and they are distinguished from one another with the help of the element number.
3. The element number in an array plays a major role for calling each element.
4. Any particular element of an array can be modified separately without disturbing the other elements.
5. Any element of an array a[ ] can be assigned or equated to another ordinary variable or array variable of its type.
6. Array elements are stored in contiguous memory locations. *]*

<u>Array Declaration:</u>

Array has to be declared before using it in C Program. Array is nothing but the collection of elements of similar data types.

<u>Syntax:</u> <data type> array_name [size 1][size 2].....[size n];

where as,

Data Type - Data Type specifies the type of the array. We can compute the size required for storing the single cell of array.
Valid Identifier - Valid identifier is any valid variable or name given to the array. Using this identifier array_name can be accessed.
Size of Array - It is maximum size that array can have.

<u>Example:</u> int num[10]; float avg[10]; double a[10]; char str[25];

[Note: What does Array Declaration tell to Compiler?
1. Type of the Array 2. Name of the Array 3. Number of Dimension
4. Number of Elements in Each Dimension]

<u>Types of an Array:</u>

An array can be of,

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

1-D Array Declaration and Initialization:

Syntax for declaration: <data_type> <array_name> [max_size];

Examples: int iarr[10]; char carr[20]; float farr[30];

Syntax for initialization: <data_type> <array_name> [max_size] = { val_1, val_2, …, val_n };

Examples: int num[5] = { 1, 2, 3, 4, 5 };

char name[10] = “program”;

float avg[3] = { 12.5, 13.5, 14.5 };

[<u>Explanation 1:</u> int y[10] = { 205, 207, 208, 209, 210, 211 }; // values separated by comma
From the above statements the array_name ***‘y’*** *elements are stored in memory as follows,*

y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	y[8]	y[9]
205	207	208	209	210	211				

Compiler will allocate ***two bytes*** *of space for* ***10*** *each* ***integer*** *array elements, totally* ***20 bytes*** *of space is allocated for array name* ***y****.]*

[<u>Explanation 2:</u> char N[10] = “WELCOME”;
From the above statements the array_name ***‘N’*** *elements are stored in memory as follows,*

N[0]	N[1]	N[2]	N[3]	N[4]	N5]	N[6]	N[7]	N[8]	N[9]
W	E	L	C	O	M	E	\0		

Compiler will allocate ***one byte*** *of space for* ***10*** *each* ***character*** *array elements, totally* ***10 bytes*** *of space is allocated for array name* ***N****.]*

Different Methods of Initializing of 1-D Array:

1. Declare & Initialize an array by its subscripts / index positions,

```
int num[5];
num[0] = 2;     // by index positions
num[1] = 8;
num[2] = 7;
num[3] = 6;
num[4] = 0;
```

2. Two ways are there to Declare an array at Compile Time,

i) initialize array size Directly int num[5] = { 2, 8, 7, 6, 0 }; *// by list / comma separated values*

ii) initialize array size Indirectly int num[ ] = { 2, 8, 7, 6, 0 };

Ex: Array elements initialize by its subscripts / index positions

```
#include<stdio.h>      // Simple Sequential program method
int main()
{
    int num[5];

    num[0]=1; num[1]=2; num[2]=3; num[3]=4; num[4]=5;

    printf(" %d ", num[0] );
    printf(" %d ", num[1] );
    printf(" %d ", num[2] );
    printf(" %d ", num[3] );
    printf(" %d ", num[4] );

    return 0;
}

O/p:      1  2  3  4  5
```

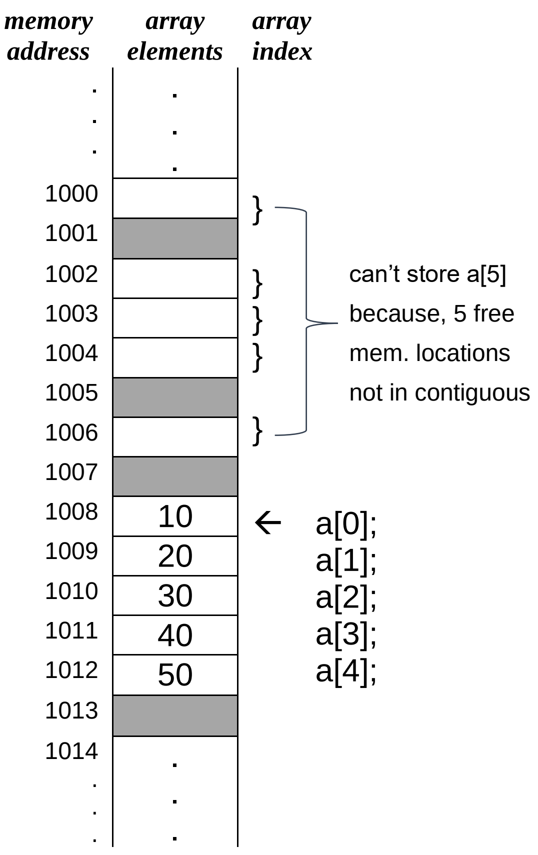
Ex: Array elements initialize by list / comma separated values

```
#include<stdio.h>      // Simple Sequential program method
int main()
{
    int num[5] = { 1, 2, 3, 4, 5 };    // int num[ ] = { 1, 2, 3, 4, 5 };

    printf(" %d ", num[0] );
    printf(" %d ", num[1] );
    printf(" %d ", num[2] );
    printf(" %d ", num[3] );
    printf(" %d ", num[4] );

    return 0;
}

O/p:      1  2  3  4  5
```

MEMORY MODEL of 1-D ARRAY:

The subscript of a column or row index will be used to access this type of array. A single subscript, in this case, represents each element. The items are saved in memory in sequential order. Array elements are stored at contiguous memory locations only. For example, A[1], A[2], …, A[n]

Representing 1-D array, **int a[5] = { 10, 20, 30, 40, 50 };** into memory as,



memory stack model

Operations on One Dimensional Array:

1. Traversing - It a process to visit each and every element of an array from first to last at least once without skipping any element.
2. Insertion - Used to insert an element at a specified position in an array.
3. Deletion - Involves deleting specified elements form an array.
3. Searching - An array element can be searched. The process of seeking specific elements in an array is called searching.
4. Merging - The elements of two arrays are merged into a single one.
6. Sorting - Arranging elements in a specific order either in ascending or in descending order.

PROGRAMS WITH 1-D ARRAY OF INTEGERS: .

List of Programs using 1-D array

1. **Initialize** and **print** 1-D array with 5 data elements.
2. **Insert a new element** at the last position in an array.
3. **Delete an element** at the last position in an array.
4. **Remove the duplicate elements** from an array.
5. **Search an element** in an array.
6. **Copy all elements** of an array **into another** array.
7. **Merge two arrays** into one array.
8. **Sum of all elements** & **Avg.** in an array.
9. **Find Smallest** element in an array.
10. **Find Biggest** element in an array.
11. **Reverse an array element** in an array.

1: Write a C program to **Initialize and Print** 1-D array with 5 data elements.

```
#include <stdio.h>
int main()
{
   int n[ ] = { 2, 8, 7, 6, 0 };
   int i;
   for (i=0; i<5; i++)
   {
     printf("\nArray Element n[%d] = %d", i, n[i]);
   }
   return 0;
}
```

Output:

```
        Array Element n[0] = 2
        Array Element n[1] = 8
        Array Element n[2] = 7
        Array Element n[3] = 6
        Array Element n[4] = 0
```

2: Write a C Program to **Insert a new element** at the last position in an array.

```
#include<stdio.h>
int main()
{
  int arr[30], num, element, i;
  printf("\nEnter no of elements: ");
  scanf("%d", &num);
  for (i = 0; i < num; i++)
  {
     scanf("%d", &arr[i]);
  }
  printf("\nEnter the element to be inserted: ");
  scanf("%d", &element);
  arr[num] = element;
  num=num+1;
  for (i = 0; i < num; i++)
  {
     printf("%d ", arr[i]);
  }
  return(0);
}
```

Output:

```
        Enter no of elements: 5
        10 20 30 40 50

        Enter element insert at last: 60
        10 20 30 40 50 60
```

3: Write a C Program to **Delete an element** at the last position in an array.

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

4: Write a C Program to delete / **Remove the duplicate elements** from an array.

```
#include<stdio.h>
int main()
{
   int arr[30], num, element, i;
   char ch;
   printf("\nEnter no of elements: ");
   scanf("%d", &num);
   for (i = 0; i < num; i++)
   {
      scanf("%d", &arr[i]);
   }
   printf("\n Delete the last element (y/n): ");
   scanf("%c", &ch);
   if(ch==’y’ || ch==’Y’)
   {
         num=num-1;
         printf(“ \n Last element is Deleted! ”);
   }
   else
         printf(“ \n Last element is NOT Deleted! ”);
   for (i = 0; i < num; i++)
   {
      printf("%d ", arr[i]);
   }
   return(0);
}
```

Output:

```
         Enter no of elements: 6
         10 20 30 40 50 60

         Delete the last element (y/n): y
         10 20 30 40 50
```

```
   int arr[20], i, j, k, size;
   printf("\nEnter array size: ");
  scanf("%d", &size);
  printf("\nEnter array elements: ");
  for (i = 0; i < size; i++)
        scanf("%d", &arr[i]);
  printf("\nArray with Unique list: ");
  for (i = 0; i < size; i++)
  {
        for (j = i + 1; j < size;)
        {
              if (arr[j] == arr[i])
              {
                    for (k = j; k < size; k++)
                    {
                          arr[k] = arr[k + 1];
                    }
                    size--;
              }
              else
                    j++;
        }
  }
  for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
  return (0);
}
```

Output:

```
Enter array size: 10
Enter array elements: 10 20 30 40 50 60 20 30 70 90

Array with Unique list: 10 20 30 40 50 60 70 90
```

5: Write a C Program to Search an element in an array.

```
#include<stdio.h>
int main()
{
   int a[30], key, num, i, flag=0;
   printf("\n Enter array size: ");
   scanf("%d", &num);
   printf("\n Enter array elements: ");
   for (i = 0; i < num; i++)
        scanf("%d", &a[i]);
   printf("\n Enter search key: ");
   scanf("%d", &key);
   for(i=0; i<num; i++)
   {
        if (key == a[i])
        {
             flag=1;
             break;
        }
   }
   if (flag=1)
      printf("Search key found at location = %d", i+1);
   else
      printf("Search key not found");
```

6: Write a C Program to Copy all elements of an array into another array.

```
#include<stdio.h>
int main()
{
   int arr1[30], arr2[30], i, num;
   printf("\nEnter no of elements:");
   scanf("%d", &num);
   printf("\nEnter the values:");
   for (i = 0; i < num; i++) {
        scanf("%d", &arr1[i]); }
   for (i = 0; i < num; i++)          // Copying array 'a' to 'b’
   {
        arr2[i] = arr1[i];
   }
   printf("The copied array is: ");
   for (i = 0; i < num; i++)
        printf("\n arr2[%d] = %d", i, arr2[i]);
   return (0);
}
```

Output:

```
    return (0);
}
```

Output:

```
Enter array size: 10
Enter array elements: 10 20 30 40 50 60 70 80 90 99

Enter search key: 50
Search key found at location = 5
------- ---
Enter search key: 100
Search key not found
```

```
Enter no of elements: 5
Enter the values: 11 22 33 44 55

The copied array is:
arr2[0] = 11
arr2[1] = 22
arr2[2] = 33
arr2[3] = 44
arr2[4] = 55
```

7: Write a C Program to **Merge two arrays** into new.

```
#include<stdio.h>
int main()
{
    int arr1[30], arr2[30], mrg[60];
    int i, j, n1, n2, n3;
    printf("\nEnter no of elements in 1st array:");
    scanf("%d", &n1);
    for (i = 0; i < n1; i++) {
        scanf("%d", &arr1[i]); }
    printf("\nEnter no of elements in 2nd array:");
    scanf("%d", &n2);
    for (i = 0; i < n2; i++)
    {
        scanf("%d", &arr2[i]);
    }
    for (i=0; i<n1; i++)                // Merging starts
    {
        mrg[i] = arr1[i];               // copy arr1[ ]  into  mrg[ ]
        j=i;
    }
    for (i=0; i<n2; i++)
    {
        j++;
        mrg[j] = arr2[i];               // copy arr2[ ]  into  mrg[ ]
    }
    n3=n1+n2;
    printf("Merged array is: ");
    for (i = 0; i < n3; i++)
    {
        printf("%d", mrg[i]);
    }
    return(0);
}
```

Output:

```
Enter no of elements in 1st array: 5
11 22 33 44 55
Enter no of elements in 2nd array: 5
66 77 88 99 100

Merged array is: 10 11 22 33 44 55 66 77 88 99 100
```

8: Write a C Program to find **Sum of all array elements**.

```
#include<stdio.h>
int main()
{
     int i, arr[50], sum, num;
     printf("\nEnter no of elements: ");
     scanf("%d", &num);
     printf("\nEnter the values: ");
     for (i = 0; i < num; i++)
          scanf("%d", &arr[i]);
     for (i = 0; i < num; i++)
          sum = sum + arr[i];
     for (i = 0; i < num; i++)
          printf("\na[%d]=%d", i, arr[i]);
     printf("\nSum=%d", sum);
     return (0);
}
```

Output:

```
Enter no of elements: 3
Enter the values: 11 22 33
a[0]=11
a[1]=22
a[2]=33
Sum=66
```

8.1: Write a C Program to find **Sum of all array elements & average** of them.

```
#include<stdio.h>
int main()
{
     int i, n=10, sum;  float avg=0.0;
     int arr[n] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 10 };
     for (i = 0; i < n; i++)
          sum = sum + arr[i];
     avg = (float) sum / 10;
     printf("\nSum = %d", sum);
     printf("\nAverage = %f", avg);
     return(0);
}
```

Output: Sum = 55 Avg = 5.5

9: Write a C program to **find smallest element** in an array

```
#include<stdio.h>
int main()
{
   int a[30], i, num, smallest;
   printf("\nEnter no. of elements: ");
   scanf("%d", &num);
   for (i = 0; i < num; i++)
        scanf("%d", &a[i]);
   smallest = a[0];
   for (i = 0; i < num; i++)
   {
         if (a[i] < smallest)
         {
               smallest = a[i];
         }
   }
   printf("\nSmallest Element is: %d", smallest);
   return (0);
}
```

Output:

```
Enter no of elements: 5
55 44 99 11 22
Smallest Element: 11
```

10: Write a C program to **find biggest element** in an array

```
#include<stdio.h>
int main()
{
   int a[30], i, num, biggest;
   printf("\nEnter no. of elements: ");
   scanf("%d", &num);
   for (i = 0; i < num; i++)
        scanf("%d", &a[i]);
   biggest = a[0];
   for (i = 0; i < num; i++)
   {
         if (a[i] > biggest)
         {
               biggest = a[i];
         }
   }
   printf("\nBiggest Element is: %d", biggest);
   return (0);
}
```

Output:

```
Enter no of elements: 5
55 44 99 11 22
Biggest Element: 99
```

11: Write a C program to **reverse an array element** in an array.

```
#include<stdio.h>
int main()
{
    int arr[30], i, j, num, temp;
    printf("\nEnter no of elements: ");
    scanf("%d", &num);
    for (i = 0; i < num; i++)
    {    scanf("%d", &arr[i]);     }
     j = i - 1;                  //  j will Point to last Element
     i = 0;                      //  i will be pointing to first element
    while (i < j)
    {
         temp  = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
         i++;                    // increment i
         j--;                    // decrement j
    }
    printf("\nResult after reversal : ");
    for (i = 0; i < num; i++)
    {    printf("%d \t", arr[i]);     }
    return (0);
}
```

Output:

```
Enter no of elements: 5
11 22 33 44 55
Result after reversal: 55 44 33 22 11
```

11.1: Write a C program to **reverse an array element** in an array.

```
#include<stdio.h>
int main()
{
    int arr[30], i, j, num, temp;
    printf("\nEnter no of elements: ");
    scanf("%d", &num);
    for (i = 0; i < num; i++)
    {
         scanf("%d", &arr[i]);
    }
    printf("\nResult after reversal : ");
    for (i=num-1; i <= 0; i--)
    {
         printf("%d \t", arr[i]);
    }
    return (0);
}
```

Output:

```
Enter no of elements: 5
11 22 33 44 55
Result after reversal: 55 44 33 22 11
```

TWO DIMENSIONAL ARRAYS .

So far, we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. In such situations this concept is useful.

- An array with two dimensions is called "Two-dimensional array"
- An array with two dimensions is called matrix
- When the data must be stored in the form of a matrix, we use two dimensional arrays

Definition of 2-D array:

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. 2-D arrays are stored in contiguous memory location row wise. ***(or)***

A matrix is a two-dimensional array that has a size of m-by-n, where m and n are nonnegative integers. As like one dimensional array, two dimensional arrays are stored in contiguous memory location row wise. ***(or)***

A two-dimensional array, also known as a 2D array, is a collection of data elements arranged in a grid-like structure with rows and columns. Each element in the array is referred to as a cell and can be accessed by its row and column indices/indexes.

Example:

```
int matrix_a[3][3];     float b[5][10];     double a[10][10];     char name[5][25];
```

2-D Array Declaration:

Two Dimensional Array requires Two Subscript Variables. It stores the values in the form of matrix. One Subscript Variable denotes the "Row" of a matrix. Second Subscript Variable denotes the "Column" of a matrix.

Syntax:

```
<datatype> array_name [row_size] [column_size];
```

For example, int matrix_a[3][3];

where, two dimensional array consisting of 3 rows and 3 columns. So the total number of elements which can be stored in this array are 3 * 3 i.e., 9.

2-D Array Initialization:

1. Initialize all Array elements but initialization is much straight forward / comma separated values. All values are assigned sequentially and row-wise.

Syntax:

```
<data_type> array_name [row_size] [column_size] = { var_1, var_2, ... , var_n };
```

(or)

```
<data_type> array_name [row_size] [column_size] = { {row1 list}, {row2 list}, ... {row n list} };
```

Example:

1. **int a[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };** // initialize all array elements as list / comma separated values

2. int a[3][3] = { 1, , , , 1, , , , 1 }; // initialize some array elements as list / comma separated values

3. **int a[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };** // array elements in row-wise

4.
```
int a[3][3] = {  { 1, 2, 3 },      // like matrix / grid of array elements in row-wise
                 { 4, 5, 6 },
                 { 7, 8, 9 }  };
```

5. int a[3][3] = { 0 }; // to initialize 0 to all coordinates

6.
```
char grid[3][4] = { { 'a', 'b', 'c', 'd' },  // 3 rows & 4 columns as single char comma separated value in row-wise
                    { 'e', 'f', 'g', 'h' },
                    { 'i',  'j', 'k', 'l' }
                  };
```

7.
```
char name[5][10] = {  "tree",        // 5 rows & 10 columns as string in row-wise
                      "bowl",
                      "hat",
                      "mice",
                      "toon"
                   };
```

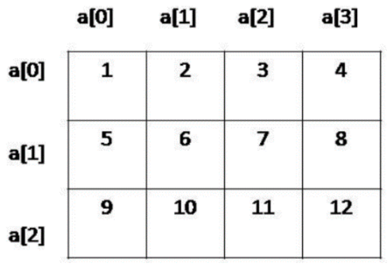
Ex: 2-D Array elements initialize by its subscripts / index positions	**Ex: 2-D Array elements initialize by list / comma separated values**

```
#include<stdio.h>      // Simple Sequential program method
int main()
{
    int mat_a[3][3];

    mat_a[0][0]=1;
    mat_a[0][1]=2;
    mat_a[0][2]=3;
    mat_a[1][0]=4;
    mat_a[1][1]=5;
    mat_a[1][2]=6;
    mat_a[2][0]=7;
    mat_a[2][1]=8;
    mat_a[2][2]=9;

    printf(" %d ", mat_a[0][0] );
    printf(" %d ", mat_a[0][1] );
    printf(" %d ", mat_a[0][2] );
    printf("\n");

    printf(" %d ", mat_a[1][0] );
    printf(" %d ", mat_a[1][1] );
    printf(" %d ", mat_a[1][2] );
    printf("\n");

    printf(" %d ", mat_a[2][0] );
    printf(" %d ", mat_a[2][1] );
    printf(" %d ", mat_a[2][2] );
    printf("\n");

    return 0;
}

O/p:     1  2  3
         4  5  6
         7  8  9
```
	```
	#include<stdio.h>      // Simple Sequential program method
	int main()
	{
	    int mat_a[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
	                        /* int mat_a[ ][ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }; */

	    printf(" %d ", mat_a[0][0] );
	    printf(" %d ", mat_a[0][1] );
	    printf(" %d ", mat_a[0][2] );
	    printf("\n");

	    printf(" %d ", mat_a[1][0] );
	    printf(" %d ", mat_a[1][1] );
	    printf(" %d ", mat_a[1][2] );
	    printf("\n");

	    printf(" %d ", mat_a[2][0] );
	    printf(" %d ", mat_a[2][1] );
	    printf(" %d ", mat_a[2][2] );
	    printf("\n");

	    return 0;
	}

	O/p:     1  2  3
	         4  5  6
	         7  8  9
	```

Storage Representation of Two-Dimensional array:

When speaking of two-dimensional arrays, we are logically saying that, it consists of two rows and columns but when it is stored in memory, the memory is linear. Hence, the actual storage differs from our matrix / grid representation.

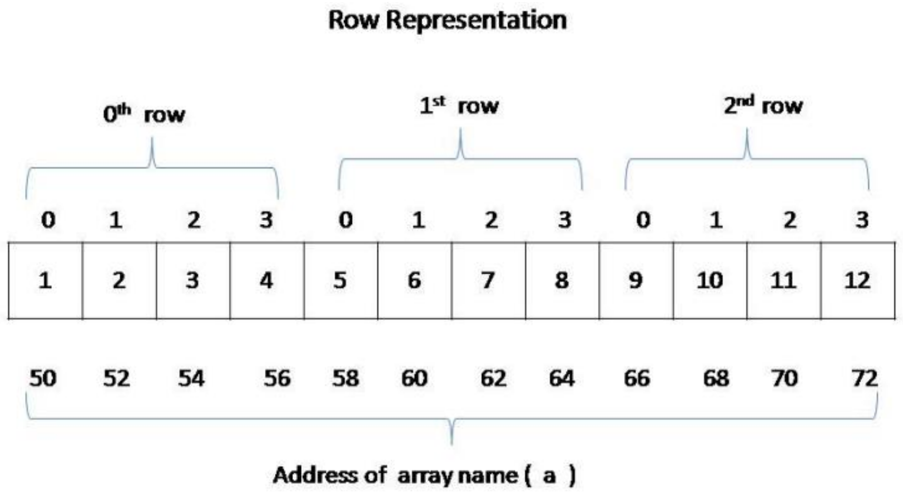
e.g. Logical view of **int a[3][4];** as,

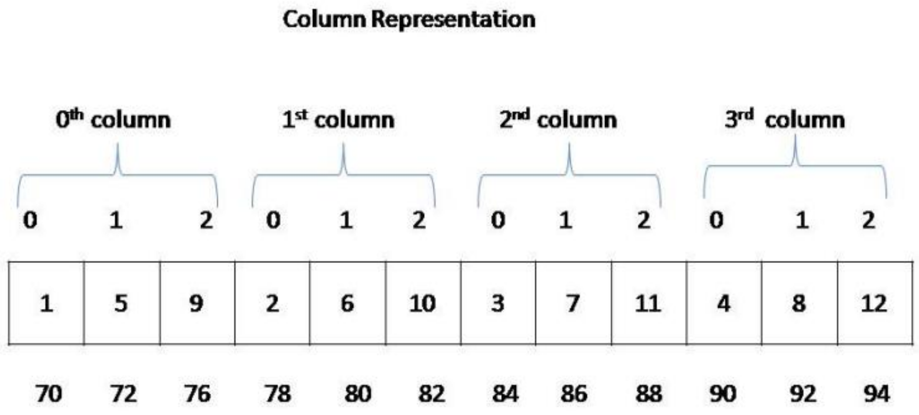


	a[0]	a[1]	a[2]	a[3]
a[0]	1	2	3	4
a[1]	5	6	7	8
a[2]	9	10	11	12

Two major types of representation can be used for 2-D array,

1. Row representation (or) Row major order
2. Column representation (or) Column major order





1. Subscript view of **int a[3][3];** by
Row major order as,

= a[**0**][0]=1;
a[**0**][1]=2;
a[**0**][2]=3;
a[**1**][0]=4;
a[**1**][1]=5;
a[**1**][2]=6;
a[**2**][0]=7;
a[**2**][1]=8;
a[**2**][2]=9;

$$= \begin{array}{cccc} r_0 & \cancel{1} & \cancel{2} & \cancel{3} \\ r_1 & 4 & 5 & 6 \\ r_2 & 7 & 8 & 9 \end{array}$$

2. Subscript view of **int a[3][3];** by
Column major order as,

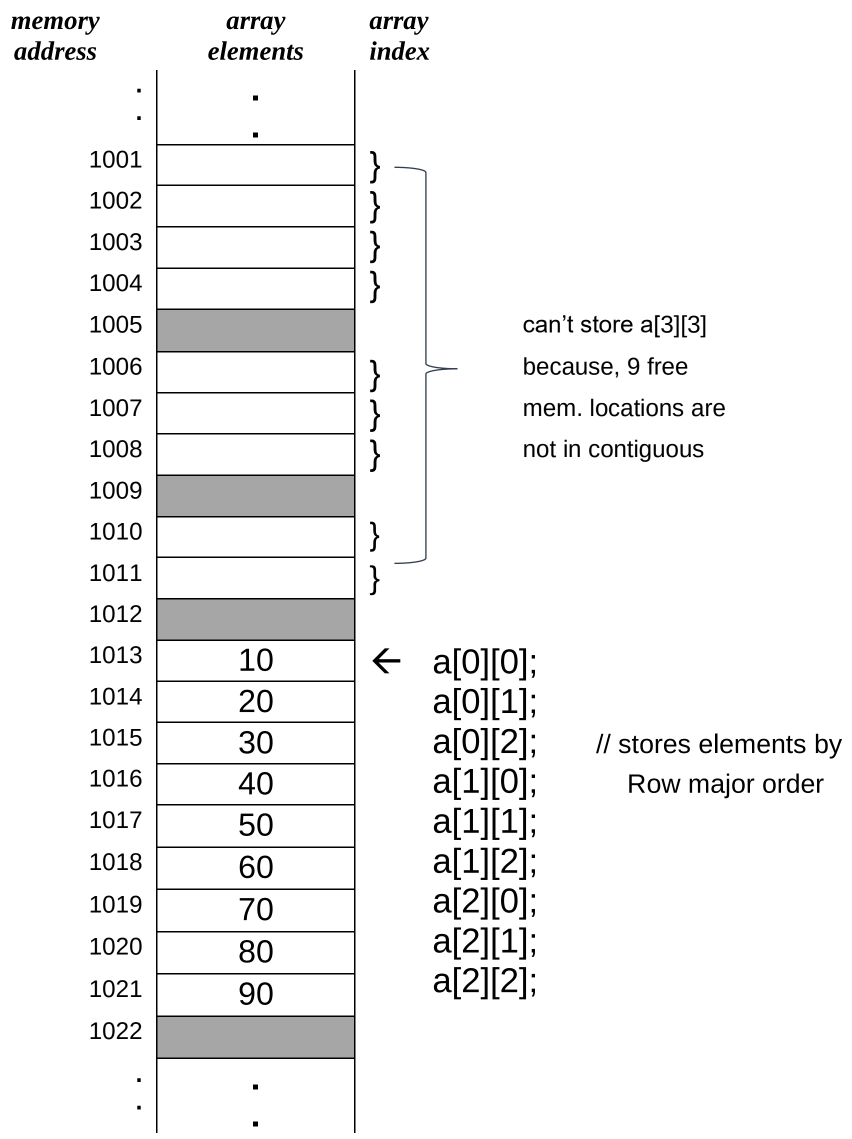
= a[0][**0**]=1;
a[1][**0**]=4;
a[2][**0**]=7;
a[0][**1**]=2;
a[1][**1**]=5;
a[2][**1**]=8;
a[0][**2**]=3;
a[1][**2**]=6;
a[2][**2**]=9;

$$= \begin{array}{ccc} c_0 & c_1 & c_2 \\ \underline{1} & 2 & 3 \\ \underline{4} & 5 & 6 \\ \underline{7} & 8 & 9 \end{array}$$

MEMORY MODEL of 2-D ARRAY:

The subscript of a row and column index will be used to access this type of array. Two subscripts, in this case, represents each element. The items are saved in memory in sequential order. Array elements are stored at contiguous memory locations only. Ex: A[1], A[2], …, A[n]

Representing 2-D array, **int a[3][3] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };** into memory as,



Memory stack model

Note 1: 'C' compiler **represents** 2-D array elements in memory **default by Row representation** or Row major order. For example, **int a[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };** *// row1, row2, row 3*

Note 2: We can tell the 'C' compiler **to represent** 2-D array elements in memory by **Column representation** or Column major order **using** the method of initializing **individual subscripts only**.

For example,

```
int a[3][3];
a[0][0]=1;     // column_0 & all row elements
a[1][0]=4;
a[2][0]=7;
a[0][1]=2;     // column_1 & all row elements
a[1][1]=5;
a[2][1]=8;
a[0][2]=3;     // column_2 & all row elements
a[1][2]=6;
a[2][2]=9;
```

PROGRAMS WITH 2-D ARRAY OF INTEGERS: .

List of Programs using 2-D array

1. **Initialize** and **print** 2-D array elements.
2. **Scan** and **Print** 2-D array.
3. **Transpose of matrix** array.
4. **Addition of two matrix** array.
5. **Multiplication of two matrix** array.

1: Write a C program to **Initialize and Print** 2-D array.

```
#include <stdio.h>
int main()
{
  int a[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  //  int a[3][3] = { { 1, 2, 3 },  { 4, 5, 6 }, { 7, 8, 9 } };
  int i, j;
  for (i=0; i<3; i++)
  {
     for (j=0; j<3; j++)
     {
        printf("\nArray Element a[%d][%d] = %d", i, j,
                                                    a[i][j] );
     }
  return 0;
}
```

Output:

```
          Array Element a[0][0] = 1
          Array Element a[0][1] = 2
          Array Element a[0][2] = 3
          Array Element a[1][0] = 4
          Array Element a[1][1] = 5
          Array Element a[1][2] = 6
          Array Element a[2][0] = 7
          Array Element a[2][1] = 8
          Array Element a[2][2] = 9
```

1.1: Write a C program to **Initialize and Print** 2-D array.

```
#include <stdio.h>
int main()
{
  int a[3][3] = { { 1, 2, 3 },
                  { 4, 5, 6 },
                  { 7, 8, 9 }
                };
  int i, j;
  for (i=0; i<3; i++)
  {
     for (j=0; j<3; j++)
     {
        printf("\nArray Element a[%d][%d] = %d", i, j,
                                                    a[i][j] );
     }
  return 0;
}
```

Output:

```
          Array Element a[0][0] = 1
          Array Element a[0][1] = 2
          Array Element a[0][2] = 3
          Array Element a[1][0] = 4
          Array Element a[1][1] = 5
          Array Element a[1][2] = 6
          Array Element a[2][0] = 7
          Array Element a[2][1] = 8
          Array Element a[2][2] = 9
```

2: Write a C program to **Scan and Print** 2-D array.

```
#include <stdio.h>
int main()
{
  int a[3][3];
  int i, j;
  for (i=0; i<3; i++)      // scan i/p
  {
     for (j=0; j<3; j++)
     {
        printf("Enter Element at a[%d][%d] : ", i, j);
        scanf("%d", &a[i][j] );
     }
  }
```

3: Write a C program to find **Transpose of matrix** array.

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

```
  for (i=0; i<3; i++)       // print o/p as matrix / grid form
  {
     for (j=0; j<3; j++)
     {
       printf(" %3d ", a[i][j] );
     }
     printf("\n");
  }
  return 0;
}
```

Output:

```
          Enter Element at a[0][0] = 1
          Enter Element at a[0][1] = 2
          Enter Element at a[0][2] = 3
          Enter Element at a[1][0] = 4
          Enter Element at a[1][1] = 5
          Enter Element at a[1][2] = 6
          Enter Element at a[3][0] = 7
          Enter Element at a[3][1] = 8
          Enter Element at a[3][2] = 9

          1   2   3
          4   5   6
          7   8   9
```

```
   {
      for (j = 0; j < 3; j++)
      {
         transpose[j][i] = matrix[i][j];
      }
   }
   printf("Transpose of the matrix: \n");
   for (i = 0; i< 3; i++)
   {
      for (j = 0; j < 3; j++)
      {
          printf("%4d", transpose[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

Outrput:

```
Enter elements of the matrix:     1   2   3
                                  4   5   6
                                  7   8   9

Transpose of the matrix:          1   4   7
                                  2   5   8
                                  3   6   9
```

4: Write a C program to perform addition of two matrix

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

   for (i = 0; i < 3; i++)
   {
      for (j = 0; j < 3; j++)
      {
         c[i][j] = a[i][j] + b[i][j] ;
      }
   }
   printf("Addition of a & b matrix is: \n");
   for (i = 0; i< 3; i++)
```

5: Write a C program for Multiplication of two matrix

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

   for (i = 0; i < 3; i++)
   {
      for (j = 0; j < 3; j++)
      {
         c[i][j]=0;
         for (k = 0; k < 3; k++)
         {
             c[i][j] = c[i][j] + a[i][k] * b[k][j];
         }
```

```
    {
        for (j = 0; j < 3; j++)
        {
            printf("%5d", c[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Outrput:

```
Enter elements of the matrix a:   1   2   3
                                  4   5   6
                                  7   8   9

Enter elements of the matrix b:   1   2   3
                                  4   5   6
                                  7   8   9

Addition of a & b matrix is:      2    4    6
                                  8   10   12
                                 14   16   18
```

```
        }
    }
    printf("Multiplication of a & b matrix is: \n");
    for (i = 0; i< 3; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%5d", c[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Outrput:

```
Enter elements of the matrix a:   1   1   1
                                  2   2   2
                                  3   3   3
Enter elements of the matrix b:   1   1   1
                                  2   2   2
                                  3   3   3
Multiplication of a & b matrix :
(1*1 + 1*2 + 1*3)  (1*1 + 1*2 + 1*3)  (1*1 + 1*2 + 1*3)      6  6  6
(2*1 + 2*2 + 2*3)  (2*1 + 2*2 + 2*3)  (2*1 + 2*2 + 2*3)  =  12 12 12
(3*1 + 3*2 + 3*3)  (3*1 + 3*2 + 3*3)  (3*1 + 3*2 + 3*3)     18 18 18
```

INTRODUCTION TO STRINGS:

Introduction:

Strings are a fundamental concept in computer science and programming, representing sequences of characters. A character is a single unit of text, which can be a letter, digit, punctuation mark, or any other symbol and even spaces. Strings are used to manipulate and store textual data in programming languages.

String in C is merely an array of characters. The length of a string is determined by a terminating null character: '\0'. So, a string with the contents, say, "abc" has four characters: 'a' , 'b' , 'c' , and the terminating null ('\0') character. The terminating null character has the value zero.

The Null Character in C is represented using the escape sequence '\0'. It is essential to distinguish it from the character '0' to prevent potential bugs in string handling.

Note: *'C' does not have a **String** data type to easily create string variables. So, must use the **char type** and create an array of characters to make a string in C: **char str[ ] = "Hello World!";***

Definition of String:

String is a sequence of character enclosed with in double quotes (" ") but ends with '\0'. The compiler puts '\0' at the end of string to specify the end of the string.

Example: "abc123" "Hello World" "Welcome to Learn 'C' programming."

Declaration of String:

Declaring a string in C is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

syntax: **char string_name[size];**

where, string_name - is any name given to the string variable and
size - is used to define the length of the string + '\0'.

Example: **char str[35] = "Welcome to learn 'C' programming.";**

Initialization of String:

String in C can be initialized in 4 different ways. They are as follows,

1. Assigning a String Literal without Size:

It is possible to directly assign a string literal to a character array without any size. The size gets determined automatically by the compiler at compile time. Here, the name of the string "str" acts as a pointer because it is an array.

example: **char str[ ] = "Welcome to learn 'C' programming.";**

2. Assigning a String Literal with Size:

String literals can be assigned with a predefined size. But we should always account for one extra space which will be assigned to the null character. If we want to store a string of size n then we should always declare a string with a size equal to or greater than n+1.

example: **char str[35] = "Welcome to learn ‘C’ programming.";**

3. Assigning Character by Character without size:

Like assigning directly without size, we also assign character by character with the Null Character at the end. The compiler will determine the size of the string automatically.

example: **char str[ ] = { ‘W’, ‘e’, ‘l’, ‘c’, ‘o’, ‘m’, ‘e’, ‘\0’ };**

4. Assigning Character by Character with size:

String can be assigned with a predefined size character by character with the Null Character at the end. Total size / length of the string + one character for ‘\0’.

example: **char str[8] = { ‘W’, ‘e’, ‘l’, ‘c’, ‘o’, ‘m’, ‘e’, ‘\0’ };**

‘C’ programs to Declare, Initialize and Read string from user:

1. String program to **Declare, Initialize** and **print.**

```
#include <stdio.h>
int main()
{
   char str[ ] = "Welcome";  // declare & initialize string
   //  char str[10] = “Welcome”;
   printf("%s", str);            // print string
   return 0;
}
```

Output:

```
        Welcome
```

2. String program to **Declare, Initialize** and **print.**

```
#include <stdio.h>
int main()
{
   char str[ ] = { ‘W’, ’e’, ’l’, ’c’, ’o’, ’m’, ’e’, ‘\0’ };
   // char str[10] = { ‘W’, ‘e’, ‘l’, ‘c’, ‘o’, ‘m’, ‘e’, ‘\0’ };
   printf("%s", str);            // print string
   return 0;
}
```

Output:

```
        Welcome
```

3. String program to **read string** and **print.**

```
#include <stdio.h>
int main()
{
   char str[25];
   printf(“Enter a String: “);
   scanf("%s", str);
   printf("Entered String: %s", str);
   return 0;
}
```

Output:

```
Enter a String: Welcome to learn C.
Entered String: Welcome
```

(Note: Entered string printed up to the first occurrence of whitespace.)

4. String program to **read** str with **whitespace** and **print.**

```
#include <stdio.h>
int main()
{
   char str[25];
   printf(“Enter a String: “);
   scanf("%[^\n]s", str);
   printf("Entered String: %s", str);
   return 0;
}
```

Output:

```
Enter a String: Welcome to learn C.
Entered String: Welcome to learn C.
```

*(Note: Entered string printed completely using **%[^\n]s** in scanf().)*

5. String program to **read** str with **gets()** and **puts().**

```
#include <stdio.h>
int main()
{
   char str[25];
   printf("\n Enter a String: ");
   gets(str);
   printf("\n Entered String: ");
   puts(str);
   return 0;
}
```

Output:

```
Enter a String: Welcome to learn C.
Entered String: Welcome to learn C.
```

6. String program to **read** str with **fgets()** and **fputs().**

```
#include <stdio.h>
int main()
{
   char str[25];
   printf("Enter a String: ");
   fgets(str, 25, stdin);
   printf("\n Entered String: ");
   fputs(str, stdout);
   return 0;
}
```

Output:

```
Enter a String: Welcome to learn C.
Entered String: Welcome to learn C.
```

String Handling Functions:

The standard ‘C’ library provides various functions to manipulate the strings within a program. These functions are also called as string handlers. All these handlers are present inside <string.h> header file.

Important String handling library functions:

Function	Purpose
1. strlen()	Used for finding a length of a string. It returns how many characters are present in a string excluding the NULL character. Returns in integer type.
2. strcpy(str1, str2)	Used to copy one string to another. It copies the contents of str2 to str1.
3. strcat(str1, str2)	Used for combining two strings together to form a single string. It Appends or concatenates str2 to the end of str1 and returns a pointer to str1.
4. strcmp(str1, str2)	Used to compare two strings with each other. It returns 0 if str1 is equal to str2, less than 0 if str1 < str2, and greater than 0 if str1 > str2.
5. strrev(str)	Used to get reverse of the given String str.
6. strlwr(str)	Used to convert the given string into lower case one.
7. strupr(str)	Used to convert the given string into upper case one.
8. strchr(str1, ch)	Used to find the first occurrence of a specified character (ch) in the given string (str1).
9. strstr(str1, str2)	Used to find the first occurrence of a specified string (str2) in the given string (str1).

1. Write a C program to find the **length of the string**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str[25];
int strlength;
printf("Enter a String: ");
scanf("%s", str);
strlength = strlen(str);
printf("Given String Length is: %d", strlength);
return(0);
}
```

Output:

```
        Enter a String:  Welcome
        Given String Length Is: 7
```

1.1. Write a C program to find the **length of the string**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str[25];
int strlength;
printf("Enter a String: ");
gets(str);
strlength = strlen(str);
printf("Given String Length is: %d", strlength);
return(0);
}
```

Output:

```
        Enter a String:  Welcome
        Given String Length Is: 7
```

2. Write a C program to **Copy one string into another**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[25], str2[25];
printf("Enter a First String: ");
scanf("%s", str1);
printf("Enter a Second String: ");
scanf("%s", str2);
printf("First String is: %s", str1);
printf("Second String is: %s", str2);
strcpy(str1, str2);
printf("After strcpy(), First String is: %s", str1);
return(0);
}
```

Output:

```
Enter a First String:
Enter a Second String: to learn C.

First String is: Welcome
Second String is: to
After strcpy(), First String is: to
```

2.1. Write a C program to **Copy one str. into another**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[25], str2[25];
printf("Enter a First String: ");
gets(str1);
printf("Enter a Second String: ");
gets(str2);
printf("First String is: %s", str1);
printf("Second String is: %s", str2);
strcpy(str1, str2);
printf("After strcpy(), First String is: %s", str1);
return(0);
}
```

Output:

```
Enter a First String: Welcome
Enter a Second String: to learn C.

First String is: Welcome
Second String is: to learn C.
After strcpy(), First String is: to learn C.
```

3. Write C program to perform **String Concatenation**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[25], str2[25];
printf("Enter a First String: ");
gets(str1);
printf("Enter a Second String: ");
gets(str2);
printf("First String is: %s", str1);
printf("Second String is: %s", str2);
strcat(str1, str2);
printf("After strcat(), First String is: %s", str1);
```

3.1. Write a C program for **String Concatenation**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[25], str2[25];
printf("Enter a First String: ");
gets(str1);
printf("Enter a Second String: ");
gets(str2);
strcat(str1, str2);
printf("After strcat(), First String is:");
puts(str1);
return(0);
return(0);
}
```

<u>Output:</u>

```
Enter a First String: Welcome
Enter a Second String: to learn C.

First String is: Welcome
Second String is: to learn C.
After strcat(), First String is: Welcome to learn C.
```

```
}
```

<u>Output:</u>

```
Enter a First String: welcome
Enter a Second String: to learn c.

First String is: welcome
Second String is: to learn c.
After strcat(), First String is: welcome to learn c.
```

4. Write C program to perform **String Comparison.**

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[25], str2[25];
int result;
printf("Enter a First String: ");
gets(str1);
printf("Enter a Second String: ");
gets(str2);
result = strcmp(str1, str2);
if(result==0)
   printf("Both the strings are Equal.");
else
   printf("Both the strings are Not Equal.");
return(0);
}
```

<u>Output:</u>

```
Enter a First String: Welcome
Enter a Second String: Welcome
Both the strings are Equal.
```

5. Write a C program to **Reverse the given String.**

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[25];
printf("Enter a String: ");
gets(str1);
puts(strrev(str1);
return(0);
}
```

<u>Output:</u>

```
Enter a String: Welcome
emocleW
```

5.1. Write a C program to find **Palindrome or not.**

```
#include<stdio.h>
#include<string.h>
int main()
{
char actual[25], reverse[25];
int result;
printf("Enter a String: ");
gets(actual);
strcpy(reverse, actual);
strrev(reverse);
result=strcmp(actual, reverse);
if(result == 0)
   printf("Given string is a Palindrome.");
else
   printf("Given strings is Not a Palindrome");

return(0);
}
```

Output:

```
Enter a String: Malayalam

Given string is a Palindrome.
```

[radar, level, civic, madam, refer, noon, rotor, deed, reviver, rotavator.]

6 & 7. C program to convert into **lower & upper case.**

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[25];
printf("Enter a String: ");
gets(str1);
puts("Actual String: ");
puts(str1);

puts("After strlwr(): ");
puts(strlwr(str1));

puts("After strupr(): ");
puts(strupr(str1));
return(0);
}
```

Output:

```
Enter a String: WelCome To Learn c

After strlwr(): welcome to learn c
After strupr(): WELCOME TO LEARN C
```

8. C program to **find search character.**

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

int main()
{
   char str1[ ] = "Welcome to learn C";
   char ch = 'm';
   puts(str1);
   puts(strchr(str1, ch));

   return 0;
}
```

Output:

```
Welcome to learn C

me to learn C
```

9. C program to **find search String.**

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

int main()
{
   char str1[ ] = "Welcome to learn C";
   char str2[ ] = "co";
   puts(str1);
   puts(strchr(str1, str2));

   return 0;
}
```

Output:

```
Welcome to learn C

come to learn C
```

STRINGS AND POINTERS:

An efficient String handling concept in ‘C’ language is achieved by implementing two methods. They are,

1. Using character array. *example: char str[25] = “Welcome to learn C.”;*

2. Using pointer variable. *example: char *ptr; ptr = str;*

Definition:

In C, Pointers play a crucial role in handling strings efficiently. String is an array of characters terminated by a null character ('\0'). Pointers can be used to manipulate strings more efficiently. Pointer that points to the beginning of the string.

Pointer is used to store the String in a Variable. That variable is called as ‘pointer variable’. While using pointer, no need to go for character data type array.

Pointer is a variable that stores the memory address of another variable. Pointers are widely used in languages like C to facilitate dynamic memory allocation, array manipulation, and function parameter passing. a pointer is declared using the * (asterisk) symbol. Dereferencing a pointer means accessing the value stored at the memory address it points to. The * (asterisk) symbol is used for dereferencing.

1. ‘C’ String program using **Pointer variable**.

```
#include <stdio.h>
int main()
{
  char myString[] = "Hello, World!";

  char *ptr = myString;

  while (*ptr != '\0')
  {
     printf("%c", *ptr);
     ptr++;
  }

  return 0;
}
```

output:

```
        Hello, World!
```

1.1. ‘C’ String program using **Pointer variable**.

```
#include <stdio.h>
int main()
{
  char myString[ ];

  char *ptr;

  puts(“Enter a String: “);
  gets(myString);

  ptr = myString;

  puts(ptr);
  return 0;
}
```

output:

```
        Enter a String:  Hello, World!
        Hello, World!
```

2. ‘C’ program to **check palindrome or not** using **Pointer variable**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char actual[ ] = ”refer”, *rev;
int result;
// rev = actual;
```

2.1. ‘C’ program to **check palindrome or not** using **Pointer variable**.

```
#include<stdio.h>
#include<string.h>
int main()
{
char actual[ ], *rev;
int result;
printf(“Enter a String: “);
strcpy(rev, actual);
strrev(rev);
result=strcmp(actual, rev);
if(result == 0)
   printf("Given string is a Palindrome.");
else
   printf("Given strings is Not a Palindrome");

return(0);
}
```

Output:

```
Enter a String: reviver
Given string is a Palindrome.

Enter a String: welcome
Given string is Not a Palindrome.
```

```
gets(actual);
strcpy(rev, actual);
strrev(rev);
result=strcmp(actual, rev);
if(result == 0)
   printf("Given string is a Palindrome.");
else
   printf("Given strings is Not a Palindrome");

return(0);
}
```

Output:

```
Enter a String: refer
Given string is a Palindrome.
```

[radar, level, civic, madam, refer, noon, rotor, deed, reviver, rotavator.]

*[C does not have **Boolean** data types, and normally uses integers for Boolean testing. **Zero is** used to represent **false**, and **One is** used to represent **true**. For interpretation, Zero is interpreted as false and anything non-zero is interpreted as true.]*

[Courtesy: PROBLEM SOLVING and PROGRAMMING (AK19) by Mr. V.SAMBASIVA, Assistant Professor, Dept. of CSE.]

UNIT 4

Pointers & User Defined Data types

Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, User-defined data types-Structures and Unions.

POINTERS: .

Introduction to Pointers:

In programming, a pointer is a variable that stores the memory address of another variable. Pointers are used to work with memory directly and enable dynamic memory allocation and manipulation. To declare a pointer, use the data type followed by an asterisk (*) and the name of the pointer variable.

You can initialize a pointer with the address of another variable. To access the value stored at the memory address pointed to by a pointer, you use the dereference operator (*).

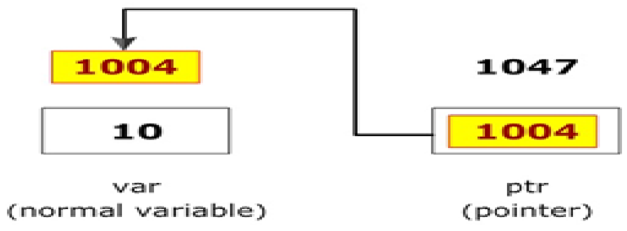
Pointers are commonly used for dynamic memory allocation, passing parameters to functions by reference, and working with arrays and structures. Pointers are powerful but avoid issues like dereferencing null pointers or accessing memory out of bounds.

Definition:

Pointer is defined as a derived data type that can store the address of another variable or a memory location. (or)

Pointer is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. (or)

Pointer is a variable that stores the memory address of another variable as its value. A pointer variable points to a data type (like int) of the same type, and is created with the * operator.



Example:

```
int n = 10;
int *p = &n;    // variable p of type pointer is pointing to the address of the variable n of integer data type
```

Features of pointers:

Pointers are used frequently in c, as they offer a number of benefits to the programmers.

1. Pointers save the memory space.
2. Execution time with pointer is faster because data is manipulated with the address i.e., direct access to memory address.
3. Pointers reduce length and complexity of programs.
4. A pointer allows C to support dynamic memory management.
5. Pointers are useful for representing two-dimensional and multi-dimensional arrays.
6. Pointers are used with data structures.

Dereferencing and address operators: .

The Dereference / Indirection operator:

Pointer is a variable that stores the address of another variable. The dereference operator is also known as an indirection operator, which is represented by (*). When indirection operator (*) is used with the pointer variable, then it is known as dereferencing a pointer. When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Dereferencing is used to access or find out the data which is contained in the memory location which is pointed by the pointer. The * (asterisk) operator which is also known as the C dereference pointer is used with the pointer variable to dereference the pointer.

Example: **int *ptr;**

Address operator:

The Address Operator in C is a special unary operator that returns the address of a variable. It is denoted as the Ampersand Symbol (&). This operator returns an integer value which is the address of its operand in the memory.

Example:
```
int num = 10;
printf("The address of num is %p", &num);      // output:  The address of num is 0x7fff5fbff7ac
```

Note: The memory address is printed using the “%p” format specifier in hexadecimal format.

Example 1: Program to explain the dereference & the address operators.

```
#include <stdio.h>
int main()
{
        int x = 9;              // declare the integer variable x
        int *ptr;               // declare the integer pointer variable ptr
        ptr = &x;               //  store the address of 'x' variable to the pointer variable 'ptr'
        *ptr = 14;              // change the value of 'x' variable by dereferencing a pointer 'ptr'
        printf("value of x is: %d", x);      // value of 'x' from 9 to 14 because 'ptr' points to 'x' location and
        return 0;                            //dereferencing of 'ptr'
}
```

Output:
```
value of x is: 14
```

<u>Example 2: Program to explain the dereference & the address operators.</u>

```
#include<stdio.h>
int main()
{
        int u = 3;
        int v;
        int *pu;
        int *pv;

        pu = &u;
        v = *pu;
        pv = &v;

        printf("\n u=%d &u=%u pu=%u *pu=%d", u, &u, pu, *pu);
        printf("\n\n v=%d &v=%X pv=%X *pv=%d ", v, &v, pv, *pv);

        return(0);
}
```

Output:

```
        u=3     &u=6684356   pu=6684356    *pu=3

        v=3     &v=65FEC0    pv=65FEC0     *pv=3
```

Note:

pu is a pointer to u, and pv is a pointer to v. therefore pu represent the address of u, and pv represents the address of v. The output shown above is, in the first printf we have printed the address of variables output with unsigned integer (%u) , and in the Second printf the address of variables is printed with hexa decimal values (%X).

<u>Example 3: Program to explain the different Arithmetic operation using pointers.</u>

```
#include<stdio.h>
int main()
{
        int a = 25, b = 10, *p, *j;
        p = &a;
        j = &b;
        printf("Addition a + b = %d \n", *p + b);
        printf("Subtraction a – b = %d \n", *p - b);
        printf("Multiplication a * b = %d \n", *p * *j);
        printf("Division a / b = %d \n", *p / *j);
        printf(" Modulo division a % b = %d \n", *p % *j);
        return(0);
}
```

Output:

```
        Addition a + b = 35
        Subtraction a - b = 15
        Multiplication a * b = 250
        Division a  /b = 2
        Modulo division a % b = 5
```

Pointer and Address arithmetic:

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The pointer variables store the memory address of another variable. It doesn’t store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations.

These operations are:

1. Increment / Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers

Example:

Data type	Initial address	Operation		Address after operations		Required bytes
int i = 2	4046	++	--	4048	4044	2 bytes
char c = ’x’	4053	++	--	4054	4052	1 bytes
float f = 4.2	4058	++	--	4062	4054	4 bytes
long int l = 2	4060	++	--	4064	4056	4 bytes

From the above table we can observe that on increase of pointer variable for integers the address is increased by two 4046 is original address and on increase its value will be 4048 because integer requires two bytes.

Similarly characters, float numbers and long integers requires 1, 4 and 4 bytes respectively.

1.1 Increment of a Pointer

```
#include<stdio.h>
int main()
{
   int n;
   int *ptr;       // pointer to an integer
   ptr = &n;

   printf("sizeof(int) is: %d bytes", sizeof(int));

   printf("Pointer ptr before Increment: %d \n", ptr);
   ptr++;          // Increment of ptr++
   printf("Pointer ptr after Increment: %d \n", ptr);


   return 0;
}
```

Output:
```
sizeof(int) is: 2 bytes
Pointer ptr before Increment: 6684360
Pointer ptr after Increment   : 6684362
```

1.2 Decrement of a Pointer

```
#include<stdio.h>
int main()
{
   int n;
   int *ptr;       // pointer to an integer
   ptr = &n;

   printf("sizeof(int) is: %d bytes", sizeof(int));

   printf("Pointer ptr before Increment: %d \n", ptr);
   ptr--;          // Decrement of ptr--
   printf("Pointer ptr after Increment: %d \n", ptr);


   return 0;
}
```

Output:
```
sizeof(int) is: 2 bytes
Pointer ptr before Increment: 6684360
Pointer ptr after Increment   : 6684358
```

2. Addition of integer to a pointer

```
#include<stdio.h>
int main()
{
    int n;
    int *ptr;                   // pointer to an integer
    ptr = &n;

    printf("sizeof(int) is: %d bytes", sizeof(int));

    printf("Pointer ptr before Addition: %d \n", ptr);
    ptr = ptr + 3;              // addition of 3 to ptr
    printf("Pointer ptr after Addition: %d \n", ptr);

    return 0;
}
```

Output:

```
sizeof(int) is: 2 bytes
Pointer ptr before Addition: 6684360
Pointer ptr after Addition   : 6684366
```

3. Subtraction of integer to a pointer

```
#include<stdio.h>
int main()
{
    int n;
    int *ptr;                   // pointer to an integer
    ptr = &n;

    printf("sizeof(int) is: %d bytes", sizeof(int));

    printf("Pointer ptr before Subtraction: %d \n", ptr);
    ptr = ptr - 3;              // Subtraction of 3 to ptr
    printf("Pointer ptr after Subtraction: %d \n", ptr);

    return 0;
}
```

Output:

```
sizeof(int) is: 2 bytes
Pointer ptr before Subtraction: 6684360
Pointer ptr after Subtraction   : 6684354
```

4. Subtracting two pointers of the same type

```
#include<stdio.h>
int main()
{
    int m, n, tot;
    int *ptr1, *ptr2;           // pointers to an integer
    ptr1 = &m;
    ptr2 = &n;

    printf("sizeof(int) is: %d bytes", sizeof(int));

    printf(" ptr1 = %d, ptr2 = %d\n", ptr1, ptr2);
    tot = ptr1 - ptr2;          // Subtraction of ptr2 and ptr1
    printf("Subtraction of ptr1 & ptr2 is: %d\n", tot);

    return 0;
}
```

Output:

```
sizeof(int) is: 2 bytes
ptr1 = 6684352,        ptr2 = 6684350
Subtraction of ptr1 & ptr2 is: 1
```

5. Comparison of pointers

```
#include <stdio.h>
int main()
{
    int arr[5];                 // declaring an array
    int *ptr1, *ptr2;           // pointers to an integer array
    ptr1 = &arr;                // declaring pointer to array name
    ptr2 = &arr[0];             // declaring pointer to first element

    if (ptr1 == ptr2)
    {   printf("Pointer to arr & 1st element are Equal.");        }
    else
    {   printf("Pointer to arr & 1st element are not Equal."); }

    return 0;
}
```

Output:

```
Pointer to Array Name and First Element are Equal.
```

Array manipulation using pointers:

Pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

pointer_type *array_name [array_size];

Here,	pointer_type:	Type of data the pointer is pointing to.
	array_name :	Name of the array of pointers.
	array_size :	Size of the array of pointers.

<u>Note:</u> *It is important to keep in mind the operator precedence and associativity in the array of pointers declarations of different type as a single change will mean the whole different thing.*

*For example, enclosing *array_name in the parenthesis will mean that array_name is a pointer to an array.*

<u>Example: C program to demonstrate the use of array of pointers</u>

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

        int var1 = 10;                                  // declaring some temp variables
        int var2 = 20;
        int var3 = 30;

        int *ptr_arr[3] = { &var1, &var2, &var3 };      // array of pointers to integers

        for (int i = 0; i < 3; i++)                     // traversing using loop
        {
                printf("Value of var%d: %d\tAddress: %d \n", i + 1, *ptr_arr[i], ptr_arr[i]);
        }

        return 0;
}
```

<u>Output:</u>

```
Value of var1: 10      Address: 6684360
Value of var2: 20      Address: 6684356
Value of var3: 30      Address: 6684352
```

USER-DEFINED DATA TYPES - Structures and Unions:

STRUCTURES [BASICS OF STRUCTURES]

Definition of Structure:

A Structure is a collection of one or more variables of different data types, grouped together under a single name. (or)

A Structure is a collection of data items of different data types under a single name.

Features of Structures:

- Structures are used to hold related data belonging to different data types.
- Nesting of Structures is possible.
- It is also possible to pass Structure elements to a function.
- It is also possible to create Structure pointers.

Declaration of Structure:

Structure can be declared as given below,

Syntax:

```
struct  <structure-name>
{
        data_type  member-1;
        data_type  member-2;
        ---        ---        ---
        data_type  member-n;
};
struct <structure-name> variable-1, variable-2,  …  …  , variable-n;
```

- Structure declaration always starts with **struct** keyword. Here, **structure-name** is known as **tag**.
- The struct declaration is enclosed with in a, pair of curly braces.
- Each member of structure may belong to different types of data.
- The individual members can be ordinary variables, arrays, pointers or other structures.

E.g. :

```
struct account                                  struct account
{                                               {
        int ac_no;                                      int ac_no;
        char ac_type;                                   char ac_type;
        char name[25];                  (or)            char name[25];
        float balance;                                  float balance;
};                                              } oldcustomer, newcustomer;
struct account oldcustomer, newcustomer;
```

Here structure name is account. It contains four members.

- Members of structure themselves do not occupy any space. Memory is allocated only after declaring structure variables.

Example for structures:

Student	: regno, student_name, age, address
Book	: bookid, bookname, author, price, edition, publisher, year
Employee	: employeeid, employee_name, age, sex, dateofbirth, basicpay
Customer	: cust_id, cust_name, cust_address, cust_phone

Structure Initialization:

Like variables, structures can also be initialized at the compile time.

Syntax:

STRUCTURE_Variable = { value_1, value_2, ... , value_n };

Note: *Individual structure members cannot be initialized within the template. Initialization is possible only with the declaration of structure members.*

Example:

```
struct student              // student is called as structure "tag"
{
        int rollno;         // rollno & attendance are called as structure "members"
        int attendance;
} s1={ 3001, 98 };          // s1 is called as structure "variable"
```

The above example assigns 3001 to the rollno and 98 to the attendance. Structure variable can be initialized outside the function also.

Example:

```
struct student
{
        int rollno;
        int attendance;
};
struct student s1={ 3001, 98 };
struct student s2={ 3002, 97 };
```

Accessing Structure Members:

There are many ways for storing values into structure variables. The members of a structure can be accessed using a “dot operator” or “period operator”.

Syntax

STRUCTURE_Variable . STRUCTURE_Member

E.g.

b1.author where, b1 - is a structure variable and
author - is a structure member

The different ways for storing values into structure variable is given below:

Method 1:	Using Simple Assignment Statement	**b1.pages** = 786;
		b1.price = 786.50;

Method 2:	Using strcpy() function	strcpy(**b1.title**, “Programming in C”);
		strcpy(**b1.author**, “John”);

Method 3:	Using scanf() function	scanf(“%s”, **b1.title**);
		scanf(“%d”, **&b1.pages**);

Example 1:

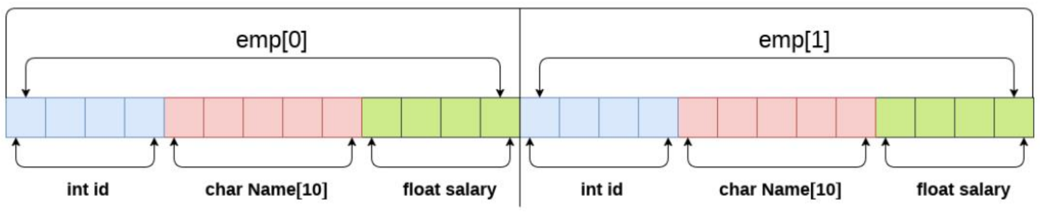
```
#include<stdio.h>
struct book
{
        int bookid;
        char bookname[30];
        char author[25];
        float price;
        int year;
        int pages;
        char publisher[25];
};
int main()
{
        struct book b1;
        printf("Enter the Book Id: ");
        scanf("%d", &b1.bookid);
        printf("Enter the Book Name: ");
        scanf("%s", b1.bookname);
        printf("Enter the Author Name: ");
        scanf("%s", b1.author);
        printf("Enter the Price: ");
        scanf("%f", &b1.price);
        printf("Enter the Year: ");
        scanf("%d", &b1.year);
        printf("Enter the Total No. of Pages: ");
        scanf("%d", &b1.pages);
        printf("Enter the Publisher Name: ");
        scanf("%s", b1.publisher);

        printf(“Entered Book Details are: “);
        printf("%d %s %s %4.2f %d %d %s", b1.bookid, b1.bookname, b1.author, b1.price,
                                                        b1.year, b1.pages, b1.publisher);
        return(0);
}
```

Output:

```
Enter the Book Id: 1002
Enter the Book Name: C_Programming
Enter the Author Name: John_Doe
Enter the Price: 123.50
Enter the Year: 2015
Enter the Total No. of Pages: 649
Enter the Publisher Name: Tata_McGraw

Entered Book Details are:
1002  C_Programming  John_Doe  123.50  2015  649  Tata_McGraw
```

Arrays of Structures:

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.



```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Example 2: C program to define an array of structures that stores information of 5 students and prints it.

```
#include<stdio.h>
struct student
{
        int rollno;
        char name[25];
};
int main()
{
        int i;
        struct student st[5];
        printf("Enter Records of 5 students");
        for(i=0; i<5; i++)
        {
                printf("\nEnter Rollno: ");
                scanf("%d", &st[i].rollno);
                printf("\nEnter Name: ");
                scanf("%s", st[i].name);
        }
        printf("\nStudents Information List:");
        for(i=0; i<5; i++)
        {
                printf("\nRollno: %d, Name: %s", st[i].rollno, st[i].name);
        }
        return 0;
}
```

Output:

```
Enter Records of 5 students     Enter Rollno: 4           Student Information List:
Enter Rollno: 1                 Enter Name: v yamini      Rollno: 1, Name: k nithyasree
Enter Name: k nithyasree        Enter Rollno: 5           Rollno: 2, Name: V Murali
Enter Rollno: 2                 Enter Name: v poojitha    Rollno: 3, Name: c v jagadeesh
Enter Name: V Murali                                      Rollno: 4, Name: v yamini
Enter Rollno: 3                                           Rollno: 5, Name: v poojitha
Enter Name: c v Jagadeesh
```

Structures within structures: (or) [Nested Structures]

Definition:

Structure can be defined inside another structure. i.e., Structures can also be nested. (or)

Nested structure in C is a structure within structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure. (or)

Nested structure in C is a structure that contains one or more members that are themselves structures.

Declaration in 2 ways:

1) **By separate nested** structure:

```
#include<stdio.h>
struct date
{
    int day;
    int month;
    int year;
};
struct account
{
    int acc_no;
    char acc_type;
    char name[25];
    float balance;
    struct date lastpayment;
};
struct account cust1;

int main()
{
struct account cust1 = { 101, ‘S’, "Samba",
                              5310.25, { 31, 03, 2023 } };
printf("Entered account Details are: ");
printf("%d %c %s %6.2f %d-%d-%d", cust1.acc_no,
            cust1.acc_type, cust1.name, cust1.balance,
            cust1.lastpayment);
return(0);
}
```

Output:

```
Entered account Details are:
101   S   Samba   5310.25   31-3-2023
```

Note : printf() can also be written as,

*printf("%d %c %s %6.2f %d-%d-%d", cust1.acc_no, cust1.acc_type, cust1.name, cust1.balance, cust1.lastpayment**.day**, cust1.lastpayment**.month**, cust1.lastpayment**.year**);*

2) **By embedded nested** structure:

```
#include<stdio.h>
struct account
{
    int acc_no;
    char acc_type;
    char name[25];
    float balance;
    struct                  // Here, date tag – may / mayn’t present
    {                       // called as Empty tag.
        int day;
        int month;
        int year;
    } lastpayment;
};

int main()
{
struct account cust1 = { 101, ‘S’, "Samba",
                              5310.25, { 31, 03, 2023 } };
printf("Entered account Details are: ");
printf("%d %c %s %6.2f %d-%d-%d", cust1.acc_no,
            cust1.acc_type, cust1.name, cust1.balance,
            cust1.lastpayment);
return(0);
}
```

Output:

```
Entered account Details are:
101   S   Samba   5310.25   31-3-2023
```

*Note : Empty structure **tag** name is permissible only in **embedded nested structure.***

UNIONS [BASICS OF UNIONS]

The concept of Union is borrowed from structure and the format are same as structure. The difference between them is in terms of total memory space required to access its members.

Definition of Union:

"Union" is a composite data type that allows you to store different types of data in the same memory location. Unlike structures, where each member has its own memory space, members of a union share the same memory location. The size of a union is determined by the size of its largest member.

Declaration of Union:

Union can be declared as given below,

Syntax:

```
union  <union-name>
{
        data_type  member-1;
        data_type  member-2;
        ---      ---      ---
        data_type  member-n;
};
union <union-name> variable-1, variable-2,  …  …  , variable-n;
```

- Union declaration always starts with **uinon** keyword. Here, **union-name** is known as **tag**.
- The union declaration is enclosed with in a, pair of curly braces.
- Each member of union may belong to different types of data.
- The individual members can be ordinary variables, arrays, pointers or other unions.

E.g. :

```
union account
{
        int ac_no;
        char ac_type;
        char name[25];
        float balance;
};
union account oldcustomer, newcustomer;
```

(or)

```
union account
{
        int ac_no;
        char ac_type;
        char name[25];
        float balance;
} oldcustomer, newcustomer;
```

Here union name is account. It contains four members.

- Members of union themselves do not occupy any space. Memory is allocated only after declaring union variables.

Example for union:

Student	: regno, student_name, age, address
Book	: bookid, bookname, author, price, edition, publisher, year
Employee	: employeeid, employee_name, age, sex, dateofbirth, basicpay
Customer	: cust_id, cust_name, cust_address, cust_phone

<u>Union Initialization:</u>

Like variables, union can also be initialized at the compile time.

Syntax:

```
UNION_Variable = { value_1, value_2, ... , value_n };
```

<u>Note:</u> *Individual union members cannot be initialized within the template. Initialization is possible only with the declaration of union members.*

Example:

```
union student             // student is called as union "tag"
{
        int rollno;       // rollno & attendance are called as union "members"
        int attendance;
} s1={ 3001, 98 };        // s1 is called as union "variable"
```

The above example assigns 3001 to the rollno and 98 to the attendance. Union variable can be initialized outside the function also.

Example:

```
union student
{
        int rollno;
        int attendance;
};
union student s1={ 3001, 98 };
union student s2={ 3002, 97 };
```

<u>Accessing Union Members:</u>

There are many ways for storing values into union variables. The members of a union can be accessed using a "dot operator (**.**)" .or. "indirect membership operator (**->**)" / "arrow operator"

Syntax

```
UNION_Variable . UNION_Member

                (or)

union <union-name> <union-variable>;
union <union-name> *ptr = &<union-variable>;
ptr -> union_member = value;
```

E.g.

ptr -> rollno = 14;   where, ptr - is a pointer to specify the union
rollno - is a union member

The different ways for storing values into union variable is given below:

Method 1:	Using Simple Assignment Statement	**b1.pages** = 786;
Method 2:	Using strcpy() function	strcpy(**b1.title**, "Programming in C");

Method 3:	Using scanf() function	scanf("%d", **&b1.rollno**);
		scanf("%d", **&(ptr -> rollno)**);

Example 1: **Union program using dot operator (.)** `

```
#include <stdio.h>
union MyUnion
{
        int integer;
        float floatingPoint;
        char character;
};
int main()
{
        union MyUnion data;
        printf("Enter an integer: ");
        scanf("%d", &data.integer);
        printf("You entered: %d\n", data.integer);

        printf("Enter a floating-point number: ");
        scanf("%f", &data.floatingPoint);
        printf("You entered: %.2f\n", data.floatingPoint);

        printf("Enter a character: ");
        scanf(" %c", &data.character);
        printf("You entered: %c\n", data.character);
        return 0;
}
```

Output:

```
        Enter an integer: 101
        You entered: 101
        Enter a floating-point number: 125.75
        You entered: 125.75
        Enter a character: c
        You entered: c
```

all the members are getting printed very well because one member is being used at a time.

Note : *If we attempt to print all the outputs together, we cannot print the correct outputs. Means, members of a union share the same memory location. The size of a union is determined by the size of its largest member.*

The values of integer and floatingPoint members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of character member is getting printed very well.

```
#include <stdio.h>
union MyUnion
{
        int integer;
        float floatingPoint;
        char character;
};
int main()
{
        union MyUnion data;
        printf("Enter an integer: ");
        scanf("%d", &data.integer);
        printf("Enter a floating-point number: ");
        scanf("%f", &data.floatingPoint);
        printf("Enter a character: ");
        scanf(" %c", &data.character);

        printf("You entered: %.2f\n", data.floatingPoint);
        printf("You entered: %d\n", data.integer);
        printf("You entered: %c\n", data.character);
        return 0;
}
```

Output:

```
Enter an integer: 101
Enter a floating-point number: 125.75
Enter a character: c
You entered: 1123778659
You entered: 125.75
You entered: c
```

Note: *This union program approach produces a wrong output.*

Example 2: **Union program using indirect membership operator (->) / arrow operator** `

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
union employee
{
        int empid;
        char name[25];
        char depart[15];
        float salary;
};
union employee *emp = NULL;
int main()
{
        emp = (union employee *) malloc( sizeof(union employee) );
        printf("Employee Details: \n");
        emp->empid = 101;
        printf("Employee ID: %d\n", emp->empid);
        strcpy(emp->name, "John Doe");
        printf("Name: %s\n", emp->name);
        strcpy(emp->depart, "Engineering");
        printf("Department: %s\n", emp->depart);
        emp->salary = 65000.00;
        printf("Salary: %.2f\n", emp->salary);
        free(emp);                              // Release dynamically allocated memory
        return 0;
}
```

<u>Output:</u>

```
        Employee Details:
        Employee ID: 101
        Name: John Doe
        Department: Engineering
        Salary: 65000.00
```

Difference between Structure and Union in C:

Structure	Union
1. It is declared using the struct keyword.	It is declared using the union keyword.
2. Each variable member occupied a unique memory space.	All members share the same memory space.
3. The size of the structure is determined by the sum of the sizes of its members.	The size of the union is determined by the size of its largest member.
4. Individual members can be accessed at a time using the dot operator.	Only one member can be accessed at a time.
5. Members can be accessed by dot operator (**.**)	Members can be accessed by dot operator (**.**) and also arrow operator / indirect membership operator (->)

6. Example:
struct student
{
int id;
char name[25];
float avg;
} s1, s2, s3;
	Example:
	union student
	{
	int id;
	char name[25];
	float avg;
	} s1, s2, s3;

7. For above structure, memory allocation will be, int id - 02 Bytes char name[25] - 25 Bytes float avg - 04 Bytes Total memory allocation = **31 Bytes** (02 + 25 + 04 = 31 Bytes must require to access)	For above union, the memory allocation will be based on the size of the largest member because all members share the same memory space in a union. the largest member is the char name[25] array, which requires 25 Bytes (1 bytes x 25 = 25). Total memory allocation = **25 Bytes** are enough.

(or)

The main differences between unions and structures can be summarized as follows:

Memory Allocation:	Unions have members that share the same memory space, while structures have separate memory space for each member.
Size:	Unions have a size determined by the largest member, while structures have a size determined by the sum of all members' sizes.
Member Access:	Only one member can be accessed at a time in a union, while structure members can be accessed individually using the dot operator.
Initialization:	Only the first member of a union can be explicitly initialized, whereas all members of a structure can be initialized individually or collectively.

UNIT 5

Functions and File Handling

Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, modifying parameters inside functions using pointers, arrays as parameters. Scope and Lifetime of Variables, Basics of File Handling.

FUNCTIONS:

Introduction to Functions:

A function is a block of code which only runs when it is called. We can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

In computer programming, a function or subroutine is a sequence of program instructions that performs a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed.

In C, functions allow developers to break down complex tasks into smaller, manageable components, promoting code reusability and maintainability.

For example,

main( ) is a function, which is used to execute code, and printf( ) is a function; used to output / print text to the screen:

```
int main()
{
        printf("Hello World!");
        return 0;
}
```

Function Definition:

Functions in C are the basic building blocks of a C program. A function is a set of statements enclosed within curly brackets { }, that take inputs, do the computation, and provide the resultant output. We can call a function multiple times, thereby allowing reusability and modularity in C programming.

The collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

Basics of functions:

- Functions are self-contained program segments that carry out some specific well-defined tasks.
- We include the header files like stdio, conio, string etc. These files contain number of library functions which are as follows: printf(), scanf(), getch(), getchar(), gets(), puts() etc.
- Every “C” program must have a function. One of the functions must be main( ).

Uses of a Function:

a) By using function, it becomes easy to write program and keep track of what they are doing
b) Length of the program can be reduced by using function
c) Debugging is easier
d) It facilitates top – down modular programming
e) Using functions avoids rewriting

There are two types of function in C programming:

i) Standard library functions
ii) User-defined functions

i) Standard Library functions:

They are predefined in the standard library of "C". We need to include the library functions such as sin(x), cos(x), pow(x), sqrt(n) etc. The code of these functions is not available to the user. So, they cannot modify these functions.

ii) User defined functions:

These functions are used need to be developed by the user at the time of program writing. Functions break large computing tasks into smaller ones, which help in the modular development of big program.

There are three steps involved in creating and using of User Defined functions.

1. Declaring of function prototype.
2. Calling a function.
3. Defining a function.

1. Declaring a Function:

A function declaration usually contains the return type, function name and the parameters data type. Usually, we declare the sub function prototype outside / above the main() globally. The following is the syntax for declaring a function prototype in C as:

syntax: **<return_type> func_name (parameters_type);** // *parameters data type*

where,	<return_type>	- is the data type of the value that the function returns
	func_name	- is the name of the function
	parameters_type	- is the data type of the parameters that the function takes as input

For example: suppose, we have a function called add() that takes two integers as input and returns their sum. We can DECLARE the function as follows:

int add(int, int); // *parameters data type*

This tells the compiler that there is a function prototype called add() that takes two integers as input (No need to specify the variables name) and returns an integer as output.

2. Calling a Function:

A function call usually contains a storage variable for the return value, the function name, and the parameters List. A function CALL is made anywhere within the main() and also any no. of times.

The following is the syntax for calling a function in C:

syntax: **func_name (parameters_list);** *// actual / informal parameters*
or
Variable = func_name (parameters_list); *// actual / informal parameters*

where,	Variable	- is the storage variable name that holds the return value of the called function
	function_name	- is the name of the function
	parameters_list	- is the list of parameters that the function takes as input

For example: suppose, we have a function called add() in the calling function main() that takes two integers as input and returns their sum. We can CALL the function as follows:

add(a, b); *// a & b are called as **actual / informal parameters***

(or)

sum = add(a, b); *// a & b are called as **actual / informal parameters***

This tells the compiler that there is a function called add() that takes two integers as input (Need to specify the variables name) and returns an integer output in sum.

3. Defining a Function:

A function defining usually contains the return type, function name and passing parameters. Usually, we define the sub function after the main(). The following is the syntax for defining a function in C as:

syntax:
```
<return_type>  func_name (parameters_list)      // dummy / formal parameter
{
        function body;       // arguments
        return;
}
```

where,	<return_type>	- is the data type of the value that the function returns
	func_name	- is the name of the function
	parameters_type	- is the list of the formal parameters that the function takes as input
	function body	- It contains the actual statements which are to be executed.

For example: suppose, we have a function called add() that takes two integers as input and returns their sum. We can DEFINE the function as follows:

```
int add(int a, int b)         // a & b are dummy / formal parameters
{
        int sum;              // sum is called as argument
        sum = a + b;
        return (sum);
}
```

This tells the compiler that there is a function called add() that takes two integers as input (Need to specify the data_type & variable name) and returns an integer as output.

[Note: The values that are declared within a function when the function is called are known as an argument. The variables that are defined when the function is declared are known as parameters.]

[In C, there are two types of parameters and they are as follows,
i) The actual parameters are the parameters that are specified in calling function (i.e., main()).
ii) The formal parameters are the parameters that are declared at called function (i.e., sub function)]

[The parameters that appear in the method definition are called formal or dummy parameters whereas the parameters that appear in the method call are called actual parameters.]

Methodology to write a Function and Sub Function Program:

There are three aspects of a C function,

1. Function declaration: A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

```
syntax:   <return_type>  func_name (parameters data_type);    // parameters data type
```

2. Function call: Function can be called from anywhere and also any no. of times in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

```
syntax:   func_name (parameter_list);               // actual parameter
          (or)
          var = func_name (parameter_list);         // actual parameter
```

3. Function definition: It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

```
syntax:   <return_type> function_name (parameter list)     // dummy parameter
          {
              function body;      // arguments
              return;
          }
```

Parameters, Arguments and Return Types:

Parameters - The variables or constants, which are passed into a function (at the time of Declaring, while Calling and when Defining).

i) Actual parameters	- Variables / Constants passed through the function call. (also called as informal parameters)
ii) Dummy parameters	- Variables received in the function definition. (also called as formal parameters)

Arguments - The variables, that are locally declare in the sub-function. Arguments created upon entry into the function and destroyed upon exit.

Return Types -

1) Simple Return	Example:	**return;**
2) Return with constant		**return (10);**
3) Return with variable		**return (a);**
4) Return with expression		**return (a+b);**

[Note: In C, a function can only have one return value. The **return** *statement in C is used to return a single value from a function.*
- ✓ *For example:* ***return; return(10); return(a); return(a+b); ----- all are valid in C.***
- ✗ ~~*For example: return(10, 14); return(a, b); return(a+b, c+d); ----- all are NOT valid in C.*~~ *]*

Function: Declare, Call and Defining.

```
        #include<stdio.h>
1.      void printName();   // parameter
        int main()
        {
           printf("Welcome to Learn: ");
2.         printName();     // actual / informal parameter
           return(0);
        }

3.      void printName()  // dummy / formal parameter
        {
           printf(" C Programming.");  // arguments
           .. …
           return;
        }
```

Output:

```
         Welcome to Learn: C Programming.
```

Function: Simple return

```
           #include<stdio.h>
1. ----    void sum();      //  parameters
           int main()
           {
              printf("\n -:- Sum of two numbers -:- \n");
2. ----       sum();        //  actual / informal parameters
              return(0);
           }
3. ---     void sum()       //  dummy / formal parameters
           {
              int a, b, c=0;          // a, b, c are arguments
              printf("\nEnter two numbers: ");
              scanf("%d %d", &a, &b);
              c = a + b;
              printf("\n The sum is: %d", c );
              return;
           }
```

Output:

```
        -:- Sum of two numbers -:-
         Enter two numbers: 10  20
         The sum is:  30
```

Function: Return with constant

```
           #include<stdio.h>
1. ----    int leapyr(int);          // parameters data type
           int main()
           {
              int r;
2. ----       r = leapyr();          // actual parameters
              if (r == 0)
                 printf("It is a Leap Year”);
              else
                 printf(“It is not a Leap Year”);
              return(0);
           }
3. ----    int leapyr()              // dummy parameters
           {
              int y, yr;          // y & yr -function arguments
              printf(“Enter a Year: “);
              scanf(“%d”, &y);
              yr = y % 4;
              if (yr == 0)
                 return(0);
              else
                 return(1);
           }
```

Output:

```
         Enter a Year: 2020
         It is a Leap Year
```

Function: Return with variable

```
           #include<stdio.h>
1. ----    int sum(int, int);        // parameters data type
           int main()
           {
              int a=10, b=20, c=0;
2. ----       c = sum(a, b);         // a & b actual parameters
              printf("Sum of %d + %d is: %d”, a, b, c);
              return(0);
           }
3. ----    int sum(int a, int b) // formal/dummy parameters
           {
              int tot=0;          // tot - function argument
              tot = a + b;
              return (tot);
           }
```

Output:

```
         Sum of 10 + 20 is: 30
```

Function: Return with Expression

```
#include<stdio.h>
1. ----  int sum(int, int);        // parameters data type
         int main()
         {
             int c=0;
2. ----      c = sum(10, 20);    //10 & 20 actual parameters
             printf("The Sum is: %d", c);
             return(0);
         }
3. ----  int sum(int a, int b) // formal/dummy parameters
         {
             return ((a+b)/2)*2;
         }

Output:
            The Sum is: 30
```

FUNCTION TYPES: .

Function can be classified into Four Types based on Parameter passing and Return value taken back to the caller. They are,

1. Function without Parameters and without Return values
2. Function without Parameters and with Return values
3. Function with Parameters and without Return values
4. Function with Parameters and with Return values

[A function may or may not accept any argument. It may or may not return any value. Based on these facts, there are four different aspects of function calls.

1. Function with no arguments and no return values

2. Function with no arguments and return values

3. Function with arguments and no return values

4. Function with arguments and return values]

[Both phrases, "function without argument" and "function without parameter," are commonly used and can be considered correct. They essentially refer to the same concept in programming. In everyday usage, both terms are acceptable, but the choice may depend on the programming language or the context in which you are discussing the function.]

1. Function with no arguments and no return values:

A function with no arguments and no return values is the First type of function in programming that performs a specific task or set of tasks but doesn't take any input parameters and doesn't provide any data back to the caller.

Syntax:

```
void function_name();      //  function declaration or prototype
function_name();           //  function call
void function_name()       //  function definition
{
    statements;
    return;
}
```

Here keyword “void” means, it is used to return no value to the calling function and there are no arguments contained in the pair of parentheses in the caller and also in the called functions.

1. Function with no arguments & no return values

```
#include<stdio.h>
void sum();
void main()
{
   printf("\nDo sum of two nos. using sub-func():");
   sum();
   return(0);
}
void sum()
{
   int a, b, c=0;
   printf("\nEnter two numbers: ");
   scanf("%d %d", &a, &b);
   c = a + b;
   printf("\nThe sum is: %d", c);
   return;
}
```

Output:

```
Do sum of two nos. using sub-func():
Enter two numbers:  10  20
The sum is: 30
```

1.1 Func w/o args & w/o return value – swapping()

```
#include <stdio.h>
void swap_2nos();
int main()
{
   swap_2nos();        // Calling the function swap_2nos()
   return 0;
}
void swap_2nos()
{
   int val_1, val_2, temp;
   printf("Enter Two values: ");
   scanf("%d%d",  &val_1, &val_2);
   int temp = val_1;       // Swapping the values
   val_1 = val_2;
   val_2 = temp;
   printf("After swapping:\n");
   printf("First value: %d\n", val_1);
   printf("Second value: %d\n", val_2);
   return;
}
```

Output:

```
        Enter Two values:  10  20
        After swapping:
        First value: 20
        Second value: 10
```

2. Function with no arguments and return values:

A function with no arguments and with return values is the Second type of function in programming that performs a specific task or set of tasks but doesn't take any input parameters and does provide a data back to the caller.

Syntax:

```
int function_name();          //  function declaration or prototype
var = function_name();        //  function call
int function_name()           //  function definition
{
     statements;
     return (value);
}
```

Here keyword “int” means, it is used to return a value to the calling function. There are no arguments contained in the pair of parentheses in the caller and also in the called functions. ‘var’ is a variable in main(), used to hold the return (value) of the function.

2. Function with no arguments & return values

```
#include<stdio.h>
int sum();
void main()
{
   int result;
   printf("\nDo sum of two nos. using sub-func():");
   result = sum();
   printf("\nAfter return, The sum is: %d", result);
   return(0);
}
int sum()
{
   int a, b, c=0;
   printf("\nEnter two numbers: ");
   scanf("%d %d", &a, &b);
   c = a + b;
   return (c);
}
```

Output:

```
        Do sum of two nos. using sub-func():
        Enter two numbers:  10  20
        After return, The sum is: 30
```

2.1 Func w/o args & return value

```
#include <stdio.h>
int factorial();
int main()
{
   long int result;
   result = factorial();          // Calling function factorial()
   printf("\nThe factorial is: %ld", result);
   return 0;
}
int factorial()                   // Called function factorial()
{
   int num, i;
   long int fact = 1;
   printf("Enter a number to find x! :  ");
   scanf("%d", &num);
   for (i = 1; i <= num; ++i)    // Calculate factorial
   {
      fact = fact * i;
   }
   return (fact);
}
```

Output:

```
        Enter a number to find x! :  5
        The factorial is:  120
```

3. Function with arguments and no return values:

A function with arguments and with no return values is the Third type of function in programming that performs a specific task or set of tasks does take input parameters and doesn't provide a data back to the caller.

Syntax:

```
void function_name(parameters data_type);     // function declaration or prototype
function_name(actual parameters);             // function call
void function_name(dummy parameters)          // function definition
{
     statements;
     return;
}
```

Here keyword "void" means, it is used to return no values to the calling function. There are arguments contained in the pair of parentheses in the caller and also in the called functions.

3. Function with arguments & no return values	**3.1 Func with args & no return values**

#include<stdio.h>
void sum(int, int);
void main()
{
   int a, b; a=10; b=20;
   sum(a, b);   // passing variable as parameter
   return(0);
}
void sum(int a, int b)
{
   int c=0;
   c = a + b;
   printf("\nThe Sum is: %d", c);
   return;
}
 	
<u>Output:</u>
 	
        The Sum is: 30
	#include<stdio.h>
	void sum(int, int);
	void main()
	{
	   sum(10, 20); // passing constant as parameter
	   return(0);
	}
	void sum(int a, int b)
	{
	   int c=0;
	   c = a + b;
	   printf("\nThe Sum is: %d", c);
	   return;
	}
	
	<u>Output:</u>
	
	        The Sum is: 30

4. Function with arguments and return values:

A function with arguments and with return values is the Fourth type of function in programming that performs a specific task or set of tasks does take input parameters and does provide a data back to the caller.

Syntax:

```
int function_name(parameters data_type);      // function declaration or prototype
var = function_name(actual parameters);       // function call
int function_name(dummy parameters)           // function definition
{
    statements;
     return (value);
}
```

Here keyword "int" means, it is used to return a value to the calling function. There are arguments contained in the pair of parentheses in the caller and also in the called functions. 'var' is a variable in main(), used to hold the return (value) of the function.

4. Function with arguments & return values

```
#include<stdio.h>
int sum(int, int);
void main()
{
   int a=10, b=20, result=0;
   printf("\nDo sum of two nos. using sub-func():");
   result = sum(a, b);
   printf("\nAfter return, The sum is: %d", result);
   return(0);
}
int sum(int a, int b)
{
   int c=0;
   c = a + b;
   return (c);
}
```

Output:

```
        Do sum of two nos. using sub-func():
        After return, The sum is: 30
```

4.1 Func with args & return value

```
#include <stdio.h>
int factorial(int);
int main()
{
   int num;
   long int result;
   printf("\nEnter a number to find x! : ");
   scanf("%d", &num);
   result = factorial(num);    // Calling function factorial()
   printf("\nThe factorial is: %ld", result);
   return 0;
}
int factorial(int num)          // Called function factorial()
{
   int i;
   long int fact = 1;
   for (i = 1; i <= num; ++i)   // Calculate factorial
   {
      fact = fact * i;
   }
   return (fact);
}
```

Output:

```
        Enter a number to find x! :  5
        The factorial is:  120
```

Modifying parameters inside functions using pointers: .

(Function with Call by Value .and. Cal by Reference)

(i) Call by Value:

1. The value of the actual parameters is copied into the formal parameters.
2. The value of the variable is used in the function call in the call by value method.
3. We cannot modify the value of the actual parameter by the formal parameter.
4. Different memory is allocated for actual and formal parameters.
5. The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Example program: (i) Call by Value:

```
#include<stdio.h>
void swap_cv(int, int);
int main()
{
        int a, b;
        printf("enter a, b values: ");
        scanf("%d%d", &a, &b);
        printf("\nActual values of a=%d  and b=%d  before swap", a, b);
        swap_cv(a, b);
        printf("\nActual values of  a=%d  and b=%d  after swap", a, b);
        return(0);
}
void swap_cv(int a, int b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
        printf("\nAfter swap  a = %d   and   a = %d ", a, b);
        return;
}
```

Output :

```
        Actual values of  a=10  and  b=20  before swap
        After swap  a = 20   and   a = 10
        Actual values of  a=10  and  b=20  after swap
```

(ii) Call by Reference:

1. The address of the variable is passed into the function call as the actual parameter.
2. The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
3. The memory allocation is similar for both formal parameters and actual parameters.
4. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Example program: (ii) Call by Reference:

```
#include<stdio.h>
void swap_cr( int *, int *);
int main()
{
        int a, b;
        printf("enter a, b values: ");
        scanf("%d%d", &a, &b);
        printf("Actual values of  x=%d  and  y=%d  before swapping \n ", a, b);
        swap_cr(&a, &b);
        printf("Actual values of  x=%d  and  y=%d  after swapping \n ", a, b);
        return(0);
}
void swap_cr(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
        printf("After swap  x = %d   and   y = %d ", *x, *y);
        return;
}
```

Output:

```
        Actual values of  a=10  and  b=20  before swapping
        After swap  x = 20   and   y = 10
        Actual values of  a=20  and  b=10  after swap
```

Pass Arrays as Function Parameters: .

Definition:

Passing array elements to a function is similar to passing variables to a function. When passing an array into a function, the function will have access to the contents of the original array.

We need to pass the array to a function to make it accessible within the function. If we pass an entire array to a function, all the elements of the array can be accessed within the function. Single array elements can also be passed as arguments. This can be done in exactly the same way as we pass variables to a function.

syntax:

```
(1) ……      int sumofarray(int num[ ]);  // declare prototype, passing array as parameter
(2) ……      result = sumofarray(num);  // calling function sumofarray(), passing array as parameter
(3) ……      int sumofarray(int num[ ])   // defining function sumofarray(), passing array as parameter
            {
                    int sum = 0;          // declaring sum as local variable in sub-function as 'argument'
                    for (int i = 0; i < 5; i++)
                    {
                            sum = sum + num[i];
                    }
                    return (sum);
            }
```

Example:

```
        #include <stdio.h>
(1) …   int sumofarray(int num[ ]);         // declare prototype, passing array num[ ] as parameter
        int main()
        {
                int result, num[ ] = { 10, 20, 30, 40, 50 };
(2) …           result = sumofarray(num); // calling function sumofarray(), passing array as actual parameter
                printf("The Sum of the given Array is = %d", result);
                return 0;
        }
(3) …   int sumofarray(int num[ ])  // defining function sumofarray(), passing array as dummy parameter
        {
                int sum = 0;            // declaring sum as local variable in sub-function as 'argument'
                for (int i = 0; i < 5; i++)
                        sum = sum + num[i];
                return (sum);                            // output:
        }                                                // The Sum of the given Array is = 150
```

What are the advantages of passing array to functions?

i) Efficiency: Passing an entire array requires less overhead than passing each element individually, as it involves passing only a single reference to the array rather than multiple values.

ii) Simplicity: It simplifies the function call and makes the code more readable, especially when dealing with large arrays.

Different ways to pass Arrays as Function Parameters:

There are Two ways to pass array as function parameter. They are,

i) can pass / have an array as a parameter.

example: **int sum(int arr[ ]);** ……………………………………….. *see example: i) 4*

ii) can pass / have a pointers in the parameter list, to hold the base address of an array.

example: **int sum (int *ptr);** ……………………………………….. *see example: ii) 1*

i) can pass / have an array as a parameter:

There is a way to pass one-dimensional arrays as arguments in C. We need to pass the array to a function to make it accessible within the function. If we pass an entire array to a function, all the elements of the array can be accessed within the function. Single array elements can also be passed as arguments.

This can be done in exactly the same way as we pass variables to a function. When passing an array into a function, the function will have access to the contents of the original array.

Examples:

1. pass array subscripts as Variable into a function:

```
#include <stdio.h>
void display(int, int);
int main()
{
  int num[ ] = { 10, 20, 30, 40, 50 };
  int a=num[1], b=num[2];
  display(a, b); //pass 2nd , 3rd  subscripts as var to func.
  return 0;
}
void display(int n1, int n2)
{
  printf("Array subscript num[1]=%d\n", n1);
  printf("Array subscript num[2]=%d\n", n2);
  return;
}
```

Output:

```
                Array subscript num[1]=20
                Array subscript num[2]=30
```

2. pass array subscripts as subscripts into a function:

```
#include <stdio.h>
void sum(int, int);
int main()
{
  int num[ ] = { 10, 20, 30, 40, 50 };
  sum(num[1], num[2]); //pass 2nd , 3rd subscripts to func.
  return 0;
}
void sum(int n1, int n2)
{
  int result=0;
  result=n1+n2;
  printf("The Sum is: %d", result);
  return;
}
```

Output:

```
                The Sum is: 50
```

3. pass array and size into a function:

```
#include<stdio.h>
void sum(int num[ ], int);
int main()
{
  int num[5]={ 10, 20, 30, 40, 50 };
  sum(num, 5); //pass entire array 'num[], size' to function
  return(0);
}
void sum(int num[ ], int n)
{
  int i, total=0;
```

4. pass entire array into a function:

```
#include<stdio.h>
void display(float num[ ]);
int main()
{
  float num[ ] = { 23.4, 55.1, 22.6, 40.5, 19.0 };
  display(num); //pass full/entire array 'num[]' to function
  return 0;
}
void display(float num[ ])
{
  int i;
  for (i=0; i<n; i++)
  {
    printf("array of num[%d] is: %d\n", i, num[i]);
    total=total+num[i];
  }
  printf("\nThe sum is: %d", total);
  return;
}
```

Output:

```
        array of num[0] is: 10
        array of num[1] is: 20
        array of num[2] is: 30
        array of num[3] is: 40
        array of num[4] is: 50

        The sum is: 150
```

```
  float total=0.0;
  printf("The given Array is: );
  for (int i = 0; i < 5; i++)
  {
     printf(" %3.1f ", num[i]);
     total=total+num[i];
  }
  printf("\nThe sum is: %5.2f", total);
  return;
}
```

Output:

```
The given Array is:  23.4  55.1  22.6  40.5  19.0
The sum is: 160.60
```

5. pass entire array into a function:

```
#include <stdio.h>
float sumavg(float num[ ]);
int main()
{
  float tot, avg;
  float num[ ] = { 23.4, 55.1, 22.6, 40.5, 19.0 };
  tot=sumavg(num); //pass entire array num[] to function
  avg=tot/5;
  printf("\nafter return, The average is: %3.2f", avg);
  return 0;
}
float sumavg(float num[ ])
{
  int i;
  float total=0.0;
  printf("The given Array is: );
  for (int i = 0; i < 5; i++)
  {
     printf(" %3.1f ", num[i]);
     total=total+num[i];
  }
  printf("\nThe sum is: %5.2f", total);
  return (total);
}
```

Output:

```
The given Array is:  23.4  55.1  22.6  40.5  19.0
The sum is: 160.60
after return, The average is: 32.12
```

6. pass Multidimensional Arrays to a Function:

To pass multidimensional arrays to a function, only the name of the array is passed to the function (similar to one-dimensional arrays).

Example: Pass two-dimensional array as a parameter into a function.

```
#include <stdio.h>
void printMatrix(int num[2][2]);
int main()
{
        int num[2][2];
        printf("Enter 2-D array: ");
        for (int i = 0; i < 2; i++)
        {
                for (int j = 0; j < 2; j++)
                {
                    scanf("%d", &num[i][j]);
                }
        }
        printMatrix(num);
        return 0;
}
void printMatrix(int num[2][2])
{
        printf("The given Matrix: \n");
        for (int i = 0; i < 2; i++)
        {
                for (int j = 0; j < 2; j++)
                {
                        printf(" %4d ", num[i][j]);
                }
                printf("\n");
        }
        return;
}
```

Output:

```
Enter 2-D array:   2   3        The given Matrix:   2   3
                   4   5                            4   5
```

Notice the parameter int num[2][2] in the function prototype and function definition:

void printMatrix(int num[2][2]); *// function prototype*

This signifies that the function takes a two-dimensional array as an argument. We can also pass arrays with more than 2 dimensions as a function argument.

When passing two-dimensional arrays, it is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified.

For example,

```
void printMatrix(int num[ ][2])
{
        // block of sub function code;
        return;
}
```

ii) can pass / have a pointers in the parameter list:

A pointer is a variable that stores a memory address. Pointers are used to store the addresses of other variables or memory items. Pointers are very useful for another type of parameter passing, usually referred to as Pass By Address. Pointers are essential for dynamic memory allocation. With a regular array declaration, you get a pointer for free. The name of the array acts as a pointer to the first element of the array. The following is the example for passing string array as parameter into a function. +++

Example 1:

```
#include<stdio.h>
void printString(char *arr);
int main()
{
        char arr[ ] = "Welcome to Learn C.";
        printString(arr);
        return 0;
}
void printString(char *str)
{
        printf("Array of Characters: ");
        int i = 0;
        while (str[i] != '\0')
        {
                printf(" \' %c \' + ", str[i]);
                i++;
        }
        printf(" \' \0 \' ");
        return;
}
```

Output:

```
Array of Characters:
'W' + 'e' + 'l' + 'c' + 'o' + 'm' + 'e' + ' ' + 't' + 'o' + ' ' + 'L' + 'e' + 'a' + 'r' + 'n' + ' ' + 'C' + '.' + '\0'
```

Scope and Lifetime of Variables:

The scope of a variable is the area in which a variable is accessible. You cannot use a variable beyond its scope. In C programming, the scope of variables is local and global. A lifetime of a variable is the working time of a variable.

Scope is defined as the availability of a variable inside a program, scope is basically the region of code in which a variable is available to use. There are four types of scope:

i) file scope,
ii) block scope,
iii) function scope and
iv) prototype scope.

Lifetime of a variable is the time for which the variable is taking up a valid space in the system's memory, it is of three types:

i) static lifetime,
ii) automatic lifetime and
iii) dynamic lifetime.

Storage classes specify the scope, lifetime and binding of variables. To fully define a variable, one needs to mention not only its 'type' but also its storage class. A variable name identifies some physical location within computer memory, where a collection of bits is allocated for storing values of variable.

Storage class tells us the following factors −

Where the variable is stored (in memory or CPU register)?
What will be the initial value of variable, if nothing is initialized?
What is the scope of variable (where it can be accessed)?
What is the life of a variable?

Storage Classes:

There are four storage classes in C language −

Storage Class	Storage Area	Default initial value	Lifetime	Scope	Keyword
Automatic	Memory	Till control remains in block	Till control remains in block	Local	**Auto**
Register	CPU register	Garbage value	Till control remains in block	Local	**Register**
Static	Memory	Zero	Value in between function calls	Local	**Static**
External	Memory	Garbage value	Throughout program execution	Global	**Extern**

Thus, a storage class is used to represent the information about a variable. There are four types of standard storage classes in C. They are,

auto	- It is a default storage class.
extern	- It is a global variable.
static	- It is a local variable which is capable of returning a value even when control is transferred to the function call.
register	- It is a variable stored inside a Register, not in a memory for faster access.

Example 1: Program for <u>auto</u> and <u>external</u> variables.

```
#include <stdio.h>
extern int x = 32;
int b = 8;
int main()
{
        auto int a = 28;
        extern int b;
        printf("The value of auto variable: %d\n", a);
        printf("The value of extern variables x and b: %d, %d\n", x, b);
        x = 15;
        printf("The value of modified extern variable x: %d\n", x);
        return 0;
}
```

<u>Output :</u>

```
The value of auto variable: 28
The value of extern variables x and b: 32,  8
The value of modified extern variable x: 15
```

Example 2: Program for <u>static</u> and <u>register</u> variables.

```
#include <stdio.h>
void st_func();
int main()
{
        int i;
        for (i = 0; i < 5; i++)
        {
              st_func();
        }
        return(0);
}

void st_func()
{
        register int ra = 10;
        static int sa = 10;
        ra += 5;
        sa += 5;
        printf("ra = %d,  sa = %d\n", ra, sa);
        return;
}
```

<u>Output:</u>

```
ra = 15,  sa = 15
ra = 15,  sa = 20
ra = 15,  sa = 25
ra = 15,  sa = 30
ra = 15,  sa = 35
```

Basics of FILE Handling: .

Intoduction:

So far, the operations using the C program are done on a prompt/terminal which is not stored anywhere. The output is deleted when the program is closed. Storing in a file will preserve your data even if the program terminates.

The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. These achieved by file handling in C.

Some few important features of FILEs:

Reusability: The data stored in the file can be accessed, updated, and deleted anywhere and anytime providing high reusability.

Portability: Without losing any data, files can be transferred to another in the computer system. The risk of flawed coding is minimized with this feature.

Efficient: A large amount of input may be required for some programs. File handling allows you to easily access a part of a file using few instructions which saves a lot of time and reduces the chance of errors.

Storage Capacity: Files allow you to store a large amount of data without having to worry about storing everything simultaneously in a program.

Definition of a FILE:

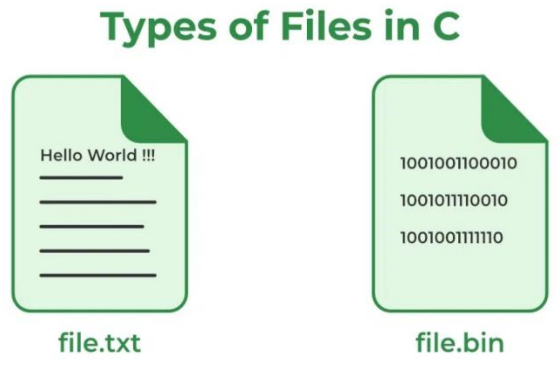
A FILE is a container in computer storage devices used for storing data. FILE is basically a data type, and we need to create a pointer variable to work with it (fptr).

A FILE represents a sequence of bytes on the disk where a group of related data is stored. FILE is created for permanent storage of data. File handing in C is the process in which we create, open, read, write, and close operations on a file.

Types of Files in C:

A file can be classified into two types based on the way the file stores the data. They are as follows,

- Text Files
- Binary Files



1. Text Files:

A text file contains data in the form of ASCII characters and is generally used to store a stream of characters.

- Each line in a text file ends with a new line character ('\n').
- It can be read or written by any text editor.
- They are generally stored with **.txt** file extension.
- Text files can also be used to store the source code.

2. Binary Files:

A binary file contains data in binary form (i.e., 0's and 1's) instead of ASCII characters. They contain data that is stored in a similar manner to how it is stored in the main memory.

- The binary files can be created only from within a program and their contents can only be read by a program.
- More secure as they are not easily readable.
- They are generally stored with **.bin** file extension.

C File Operations:

File operations refer to the different possible operations can perform on a file in 'C' such as:

Creating a new Text file	– fopen() attributes "a" or "a+" or "r" or "r+" or "w" or "w+"
Creating a new Binary file	– fopen() attributes "ab" or "ab+" or "rb" or "rb+" or "wb" or "wb+"
Opening an existing file	– fopen()
Reading from file	– fscanf() or fgets()
Writing to a file	– fprintf() or fputs()
Moving to a specific location in a file	– fseek(), rewind()
Closing a file	– fclose()

Files are classified into the following:

1. Sequential files	− Data is stored and retained in a sequential manner.
2. Random access Files	− Data is stored and retrieved in a random way.

Text File Modes in C

During programming text file can be opened for different purposes i.e., reading, writing or both. This is specified by using file modes. Different text file modes in C programming are tabulated below:

Modes	*Purpose*
r	*Open text file for reading only. The file must already exist.*
w	*Open text file for writing only. If the file specified already exists, its contents will be destroyed. If not exist it will be created.*
a	*Open text file for appending (i.e. adding data to the end of existing file). If it does not exist it will be created.*
r+	*Open text file for both reading and writing. The file must already exist.*
w+	*Open text file for both reading and writing. If the file exists, its content will be overwritten. If not exist it will be created.*
a+	*Open text file for both reading and appending. If a file does not exist it will be created.*

Binary File Modes in C

During programming binary file can be opened for different purposes i.e. reading, writing or both. This is specified by using file modes. Different binary file modes in C programming are tabulated below:

Modes	*Purpose*
rb	*Open binary file for reading only. The file must already exist.*
wb	*Open binary file for writing only. If the file already exists, its contents will be destroyed. If not exist it will be created.*
ab	*Open binary file for appending (i.e. adding data to the end of existing file). If it does not exist it will be created.*
rb+/r+b	*Open binary file for both reading and writing. The file must already exist.*
wb+/w+b	*Open binary file for both reading and writing. If the file exists, its content will be overwritten. If it does not exist it will be created.*
ab+/a+b	*Open binary file for both reading and appending. If a file does not exist it will be created.*

1. fopen()	syntax of File pointer:	FILE *file_pointer;
	example:	**FILE *fptr;**
	syntax for naming & opening a file:	file_pointer = fopen ("file name", "mode");
	example:	**FILE *fptr;** **fptr = fopen ("sample.txt", "w");**
2. fclose()	syntax for file close:	fclose (file_pointer);
	example:	**fclose (fptr);**
3. fscanf()	syntax for read the file contents:	fscanf(file_pointer, “type_specifier”, &variable);
	example:	**fscanf(fptr, "%d", &num);**
4. fprintf()	syntax for read the file contents:	fscanf(file_pointer, “type_specifier”, &variable);
	example:	**fscanf(fptr, "%d", &num);**

Example: File Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        char text[35];

        FILE *fptr;
        fptr = fopen("d:\\test.txt", "w");

        if(fptr == NULL)
        {
                printf("Error!");
                exit(1);
        }
        printf("Write to FILE:  Enter some text: ");
        scanf("%s", &text);
        fprintf(fptr, "%s", text);
        fclose(fptr);

        fptr = fopen("d:\\test.txt", "r");
        fscanf(fptr, "%s", &text);
        printf("Read from FILE:  Entered text =  %s", text);
        fclose(fptr);

        return(0);
}
```

Output:

```
Write to FILE:  Enter some text:  This_is_a_FILE_program.
Read from FILE:  Entered text = This_is_a_FILE_program.
```