



Programming In C

EST-102

As per KTU 2019 syllabus for Semester 2

Jayaraj V S
Sivapriya P M



Programming In C

EST-102

As per KTU 2019 syllabus for Semester 2

Edited By

Jayaraj V S

Sivapriya P M



Free learning content publishers

I would like to dedicate this book

to

*My beloved father late **Shri R Vamadevan***

and

*My beloved mother late **Smt Sheela Vamadevan***

Table of Contents

Preface

About The Editors

Module I

1.1	Basics of Computer Architecture	1
1.1.1	Components of a computer	1
1.1.2	Computing Concepts of a Computer	2
1.1.3	Computer Architecture	3
1.2	System Software and Application Software	6
1.2.1	High-level and Low-level languages	8
1.2.2	Compilers and Interpreters	10
1.3	Introduction to structured approach to programming	12
1.3.1	Algorithm and Flowchart	13
1.3.2	Classification of Algorithms	18
1.3.3	Flowchart definition	19
1.4	Pseudo code	23
1.4.1	Common keywords used in pseudo code	24
1.4.2	Linear search in C	27
1.4.3	Bubble sort in C	30

Module II

Brief History of C Programming Language		35
2.1	Basic structure of C program	38
2.1.1	Character set in C	40
2.1.2	Tokens in C	44
2.1.2.1	Keywords	44
2.1.2.2	Identifiers	45
2.1.2.3	Strings in C	46
2.1.2.4	Operators in C	46
2.1.3	C Variable, Data types, Constants	47
2.1.3.1	Variable	47
2.1.3.2	Data types	48
2.1.3.2.1	Integer data type	49
2.1.3.2.2	Floating point data type	49
2.1.3.2.3	Character data type	49
2.1.3.2.4	Void data type	50
2.1.3.3	Constants	50
2.1.3.3.1	Integer constants	50
2.1.3.3.2	Character constants	50

2.1.3.3.3	String constants	51
2.1.3.3.4	Real Constants	51
2.1.4	Console Input/output Operations: printf() and scanf()	51
2.2	Operators and Expressions	57
2.2.1	Arithmetic Operators	57
2.2.2	Relational Operators	58
2.2.3	Logical Operators	60
2.2.4	Bitwise Operators	61
2.2.5	Assignment Operator	62
2.2.6	Misc Operator	63
2.2.6.1	Sizeof Operator in C	64
2.2.6.2	Conditional Expression (Ternary operator)	65
2.2.7	Precedence of operators	66
2.3	Control Flow Statements	68
2.3.1	Branching	69
2.3.1.1	If Statement	69
2.3.1.2	If-else statement	72
2.3.1.3	If else-if ladder	73
2.3.1.4	Nested if	75
2.3.2	C Switch Statement	77
2.3.3	Unconditional Branching using goto statement	80
2.3.4	C Loops	82
2.3.4.1	do-while loop in C	83
2.3.4.2	While loop in C	84
2.3.4.2.1	Infinitive while loop in C	86
2.3.4.3	For loop in C	87
2.3.4.3.1	Loop expressions For loop	88
2.3.4.3.2	Infinitive for loop in C	92
2.3.5	Break and Continue statements	92
2.3.5.1	C break	92
2.3.5.2	C Continue	93
2.3.6	Simple C programs	95

Module III

3.1	Arrays Declaration and Initialization	102
3.1.1	Declaration of Array in C	103
3.1.2	Initialization of C Array	103
3.1.3	C Array: Declaration with Initialization	104
3.1.4	Two-Dimensional Array in C	106
3.1.4.1	Declaration of two-dimensional Array in C	106
3.1.4.2	Initialization of 2D Array in C	107

3.2	String processing	109
3.2.1	Difference between char array and string literal	109
3.2.2	How to initialize strings?	109
3.2.3	Traversing String	110
3.2.4	Accepting string as the input	112
3.3	String handling functions	113
3.3.1	C String Length: strlen() function	113
3.3.2	C String Copy: strcpy () function	114
3.3.3	C String Concatenation: strcat () function	115
3.3.4	C String Comparison: strcmp () function	116
3.2.5	C gets() and puts() functions	117
3.2.5.1	C gets() function	117
3.2.5.2	C puts() function	119
3.4	Linear search and Bubble sort program, simple programs covering arrays and strings	119
3.4.1	Linear Search Program to search an element in an array	119
3.4.2	Bubble sort Program to sort elements in an array	121
3.4.3	Simple programs covering arrays and strings	123

Module IV

4.1	Working with functions	145
4.1.1	Functions in C	145
4.1.2	Types of function	146
4.1.2.1	Standard library functions	146
4.1.2.2	User-defined function	147
4.1.3	Working of a function in C	147
4.1.4	Declaration / Defining a function	148
4.1.5	Calling a function	149
4.1.6	Example of a User-defined function	149
4.1.7	Different aspects of function calling in C	150
4.1.7.1	Function without arguments and without return value	150
4.1.7.2	Function without arguments and with return value	151
4.1.7.3	Function with arguments and without return value	152
4.1.7.4	Function with arguments and with return value	153
4.1.8	Actual and Formal arguments in C	154
4.1.8.1	Actual arguments	154
4.1.8.2	Formal Arguments	154
4.2	Working with functions continued	156
4.2.1	Call by value in C	156
4.2.1.1	Call by Value Example: Swapping the values of the two variables	156
4.2.2	Recursion in C	157

4.2.2.1	Recursion function	157
4.2.2.2	How recursion works?	158
4.2.3	Arrays as Function Parameters / Passing arrays as parameter to function	161
4.2.3.1	Passing a single array element to a function	161
4.2.3.2	Passing a complete One-dimensional array to a function	162
4.2.3.3	Passing a Multi-dimensional array to a function	163
4.2.3.4	More Examples	164
4.3	Working with functions continued	166
4.3.1	C Structure	166
4.3.1.1	Defining a structure in C	166
4.3.1.2	Declaring Structure Variables	167
4.3.1.3	Accessing Structure Members	167
4.3.1.4	Structure Initialization	169
4.3.1.5	Array of Structure	170
4.3.1.6	Nested Structures	173
4.3.1.7	typedef in C	176
4.3.1.7.1	Structure definition using typedef	177
4.3.2	C Union	179
4.3.2.1	Defining union	179
4.3.2.2	Create union variables	180
4.3.2.3	Difference between structures and unions	182
4.3.2.4	Accessing Union Members	184
4.3.2.5	Comparing Structures and unions	185
4.3.3	Storage Classes in C	186
4.3.3.1	The auto Storage Class	187
4.3.3.2	The extern Storage Class	188
4.3.3.3	The static Storage Class	189
4.3.3.4	The register Storage Class	190
4.3.4	Scope and life time of variables	190
4.3.5	Simple programs using functions	191

Module V

5.1	Basics of Pointer	205
5.1.1	What is Pointer in C Programming?	205
5.1.2	Declaring a pointer	205
5.1.2.1	Pointer to integer, double, character, float	205
5.1.2.2	Pointer to array	206
5.1.2.3	Pointer to a function	206
5.1.3	Advantages of pointer	206
5.1.4	Usage of pointer	206
5.1.5	Address Of (&) Operator	206

5.1.6	NULL Pointer	207
5.1.7	Working of Pointers	207
5.1.7.1	Pointer Example: Assigning addresses to Pointers	207
5.1.7.2	Pointer Example: Get Value of Thing Pointed by Pointers	208
5.1.7.3	Pointer Example: Changing Value Pointed by Pointers	208
5.1.7.4	Pointer Example: Using pointers to print the address and value	209
5.1.7.5	Pointer Example: Working of Pointers	210
5.1.8	Common mistakes when working with pointers	210
5.1.9	Arrays and Pointers	211
5.1.9.1	Relationship Between Arrays and Pointers	211
5.1.9.2	Pointer to an Array in C	214
5.1.9.3	Arrays of Pointers in C	215
5.1.9.4	Sample programs demonstrating Working of Pointers	215
5.1.10	Call by reference in C	220
5.1.11	Dynamic memory allocation in C	221
5.1.11.1	malloc()	222
5.1.11.2	calloc()	222
5.1.11.3	realloc()	222
5.1.11.4	free()	222
5.2	File Operations	226
5.2.1	Introduction to File Handling	226
5.2.2	C File Operations	227
5.2.2.1	Opening a file	227
5.2.2.2	Reading a File	228
5.2.2.3	Writing to a file	230
5.2.2.4	Closing a file: fclose()	232
5.2.2.5	Appending a file	232
5.2.3	Reading and writing to a binary file	233
5.2.3.1	Writing to a binary file	233
5.2.3.2	Reading from a binary file	234
5.2.3.3	Appending to a binary file	235
5.3	Sequential access and random access to files	237
5.3.1	What is Sequential Access?	237
5.3.2	What is Random Access?	237
5.3.3	Advantages of Sequential access and random access to files	237
5.3.4	Random access/Direct access files	237
5.3.5	C fseek() function	237
5.3.6	C rewind() function	240
5.3.7	C ftell() function	240
5.3.8	feof() function	241

5.3.9 Simple programs covering pointers and files	242
C PROGRAMMING LAB	251
LIST OF LAB EXPERIMENTS	252
1 Familiarization of Hardware Components of a Computer	254
2 Familiarization of Linux environment – How to do Programming in C with Linux	256
3 Familiarization of console I/O and operators in C	261
i) Display “Hello World”	
ii) Read two numbers, add them and display their sum	
iii) Read the radius of a circle, calculate its area and display it	
iv) Evaluate the arithmetic expression $((a - b / c * d + e) * (f + g))$ and display its solution. Read the values of the variables from the user through console.	
4 Read 3 integer values and find the largest among them.	263
5 Read a Natural Number and check whether the number is prime or not.	263
6 Read a Natural Number and check whether the number is Armstrong or not.	264
7 Read n integers, store them in an array and find their sum and average.	265
8 Read n integers, store them in an array and search for an element in the array using an algorithm for Linear Search.	267
9 Read n integers, store them in an array and sort the elements in the array using Bubble Sort algorithm.	268
10 Read a string (word), store it in an array and check whether it is a palindrome word or not.	270
11 Read two strings (each one ending with a \$ symbol), store them in arrays and concatenate them without using library functions.	271
12 Read a string (ending with a \$ symbol), store it in an array and count the number of vowels, consonants and spaces in it.	272
13 Read two input each representing the distances between two points in the Euclidean space, store these in structure variables and add the two distance values.	273
14 Using structure, read and print data of n employees (Name, Employee Id and Salary).	274
15 Declare a union containing 5 string variables (Name, House Name, City Name, State and Pin code) each with a length of C_SIZE (user defined constant). Then, read and display the address of a person using a variable of the union.	276
16 Find the factorial of a given Natural Number n using recursive and non-recursive functions.	277
17 Read a string (word), store it in an array and obtain its reverse by using a user defined function.	279

18	Write a menu driven program for performing matrix addition, multiplication and finding the transpose.	280
	Use functions to	
	(i) Read a matrix.	
	(ii) Find the sum of two matrices.	
	(iii) Find the product of two matrices.	
	(iv) Find the transpose of a matrix	
	(v) Display a matrix.	
19	Do the following using pointers	284
	i) add two numbers	
	ii) swap two numbers using a user defined function	
20	Input and print the elements of an array using pointers.	286
21	Compute sum of the elements stored in an array using pointers and user defined function.	287
22	Create a file and perform the following	288
	i) Write data to the file	
	ii) Read the data in a given file & display the file content on console	
	iii) Append new data and display on console	
23	Open a text input file and count number of characters, words and lines in it; and store the results in an output file.	289

Bibliography

Preface

This book is intended for all aspirants who would like to begin exploring their programming skills with C. C is currently the premier language for software developers as it's widely distributed and standardized. Newer languages are available, such as Python, Java, R etc. but still C is the language of choice for robust and portable programming. This is because System level programming and Embedded system programming still dependent on C.

This book emphasizes the basic skills one requires to solve real-world programming problems. It teaches you not only the mechanics of the C language, but helps you to understand the concepts clearly with usage of different color shades in code presentation and explanation.

In order to bridge the gap between theory and practical, each concept is explained at length in an easy-to-understand manner supported with numerous worked-out examples and programs.

The Codes presented in this book are compiled and run-on **Visual Studio Code**. Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with a rich ecosystem of extensions for languages (such as C++, C#, Java, Python, PHP, Go). Using Visual Studio Code would help a beginner to visually understand and rectify the errors while coding.

Anybody can write a code, but real skill lies in the way a code is written and presented. So, to create a good program, one must do more than just type in a code. It is always expected from a programmer to blend both writing and programming skills together to form a simple, readable and easy to understand Code.

Book Organization

As this book is edited as per the KTU syllabus, it is organized into two parts:

i) Module by Module

Module-I: Gives a basic understanding of computer components and its architecture along with best method of writing algorithms and drawing of flowchart. It also describes the writing of Pseudo code.

Module-II: Gives a basic understanding of the structure of a C program, Operators and Expressions and also in detail the programming with Control Flow Statements.

Module-III: Gives a detail understanding of Arrays and Strings along with programs covering arrays and strings.

Module-IV: Introduces to modular programming, Functions, Recursion, Arrays as Function Parameters, Structure, Union, Storage Classes, Scope and life time of variables.

Module-V: Introduces to Basics of Pointers, File Operations in C, Sequential access and Random access to files.

ii) Lab Programs

A list of 23 programs as per the syllabus is provided to give the students a better hand on session with topics learned from modules in part one.

As editors of this book, we are very delighted to present the *First Edition of Programming in C*, which is free and can be downloaded from [jayarajvamadevan \(Jayaraj Vamadevan\) · GitHub](https://github.com/jayarajvamadevan). We have tried to make this book Student friendly by using simple and lucid language. We hope both student community and teaching fraternity are benefitted by our book.

Jayaraj V S

Sivapriya P M

About The Editors

Jayaraj V S did his BE in Electronics & Communication Engineering in 2005 from Mohammed Sathak Engineering College, Kilakarai and ME in Communication Systems in 2007 from Mepco Schlenk Engineering College, Sivakasi. Currently working as a faculty of Sree Buddha College of Engineering, Pattoor.

Sivapriya P M did her BTech in Computer Science in 2009 from Government Engineering College Palakkad, Sreekrishnapuram. Currently working with the Department of Post Government of India.

MODULE BY MODULE

(Theory part of EST 102, Programming in C)

Module-I

Basics of Computer Hardware and Software	
1.1	Basics of Computer Architecture: Processor, Memory, Input& Output devices
1.2	Application Software & System software: Compilers, interpreters, High level and low-level languages
1.3	Introduction to structured approach to programming, Flow chart
1.4	Algorithms, Pseudo code (bubble sort, linear search - algorithms and pseudo code)

1.1 Basics of Computer Architecture

What is a computer?

“A computer is an electronic machine that takes input from the user, processes the given input and generates output in the form of useful information”

A computer accepts input in different forms such as **data, programs** and **user reply**.

- **Data** refer to the raw details that need to be processed to generate some useful information.
- **Programs** refer to the set of instructions that can be executed by the computer in sequential or non-sequential manner.
- **User reply** is the input provided by the user in response to a question asked by the computer.

The main task of a computer system is to process the given input of any type in an efficient manner. Therefore, computer is also known by various other names such as data processing unit, data processor and data processing system.

1.1.1 Components of a computer

Computer includes various devices that function as an integrated system to perform several tasks described above.

These devices are:

1) Central Processing Unit (CPU)

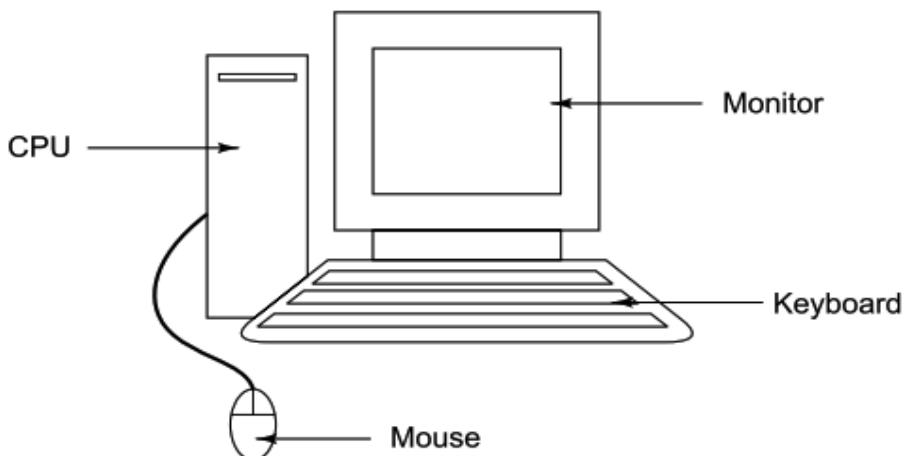
It is the processor of the computer that is responsible for controlling and executing instructions in the computer. It is considered as the most significant component of the computer. It is the “brain” of the computer.

2) Monitor

It is a screen, which displays information in visual form, after receiving the video signals from the computer.

3) Keyboard and Mouse

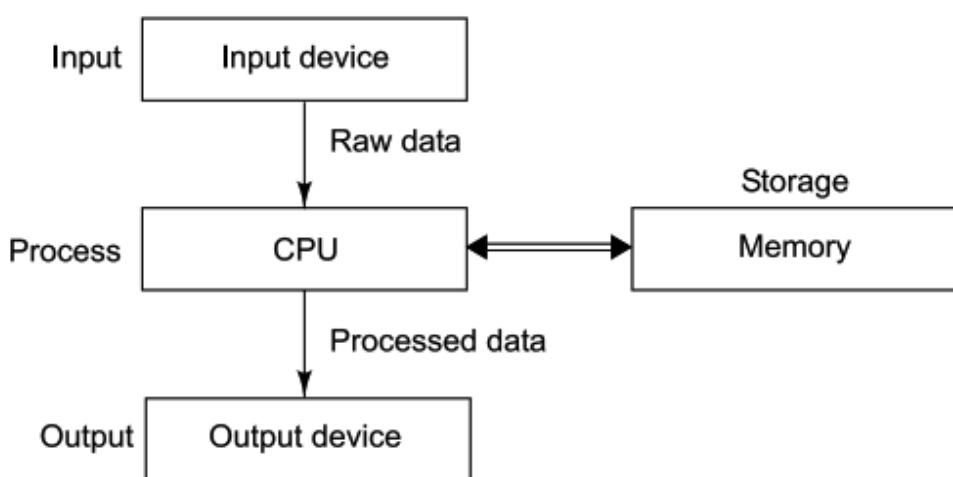
These are the devices, which are used by the computer, for receiving input from the user.



The components of a computer

1.1.2 Computing Concepts of a Computer

We can understand how a computer functions by analysing the fundamental computing concepts. The most elementary computing concepts include receiving input known as data from the user, manipulating the input according to the given set of instructions and delivering the output known as information to the user. Figure below shows the functioning of a computer based on these concepts.



The Input-Process-Output Cycle of a computer

The various functions performed by the computer are briefly described below:

a) Accepting the raw data

The first task to be performed by a computer is to accept the data from the user, with the help of an input device, such as mouse and keyboard. Mouse is used to enter the data through point-and-click operation while keyboard is used to enter the character data by typing the various keys.

b) Processing the data

The data is processed with the help of specific instructions known as programs after taking the input from the user. The manipulation of data is handled by the CPU of the computer. CPU is considered as the brain of the computer because it controls the execution of various instructions. The raw data entered by the user through input devices is processed by the CPU to generate meaningful information.

c) Storing the data

The data is stored in the main memory of a computer in its processed form. The various external storage devices—such as hard disk and magnetic disk—can also be used for storing the processed data so that it can again be fetched later.

d) Delivering the output

The processed data is delivered as useful information to the user with the help of output devices, such as printer and monitor.

1.1.3 Computer Architecture

Computer architecture is a science or a set of rules stating how computer software and hardware join and interact to make a computer work. The architecture basically defines the logical structure of a computer system.

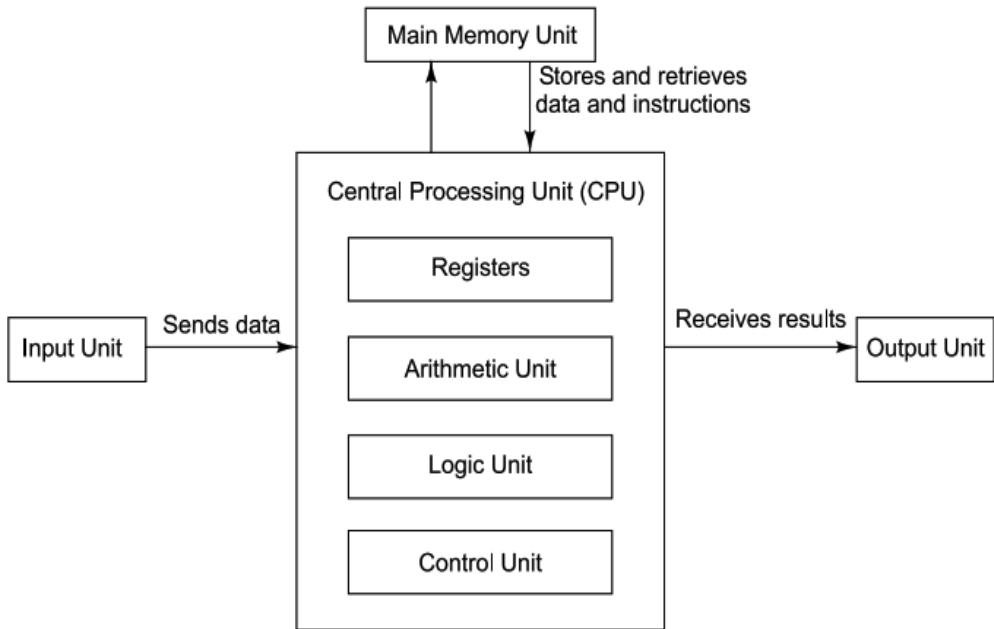
Historically there have been 2 types of Computers:

a) Fixed Program Computers – Their function is very specific, and they couldn't be programmed, e.g., Calculators.

b) Stored Program Computers – These can be programmed to carry out many different tasks, applications are stored on them, hence the name.

The modern computer is based on a stored-program concept introduced by John Von Neumann and has three basic units.

- The Central Processing Unit (CPU)
- Memory Unit
- The Input / Output Unit



Von Neumann Architecture

1) The Central Processing Unit (CPU)

A Central Processing Unit is also called a processor, central processor, or microprocessor. It carries out all the important functions of a computer. It receives instructions from both the hardware and active software and produces output accordingly. It stores all important programs like operating systems and application software.

CPU also helps Input and output devices to communicate with each other. Owing to these features of CPU, it is often referred to as the brain of the computer.

CPU is installed or inserted into a CPU socket located on the motherboard. Furthermore, it is provided with a heat sink to absorb and dissipate heat to keep the CPU cool and functioning smoothly.

It basically has three main units:

- a) Arithmetic and Logic Unit (ALU)
- b) Control Unit
- c) Main Memory Unit (Registers)

a) Arithmetic and Logic Unit (ALU)

It is the arithmetic logic unit, which performs arithmetic and logical functions. Arithmetic functions include addition, subtraction, multiplication, division, and comparisons. Logical functions mainly include selecting, comparing, and merging the data. A CPU may contain more than one ALU. Furthermore, ALUs can be used for maintaining timers that help run the computer.

b) Control Unit

It is the circuitry in the control unit, which makes use of electrical signals to instruct the computer system for executing already stored instructions. It takes instructions from memory and then decodes and executes these instructions. So, it controls and coordinates the functioning of all parts of the computer. The Control Unit's main task is to maintain and regulate the flow of information across the processor. It does not take part in processing and storing data.

c) Main Memory Unit (Registers)

- **Accumulator:** Stores the results of calculations made by ALU.
- **Program Counter (PC):** Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR).
- **Memory Address Register (MAR):** It stores the memory locations of instructions that need to be fetched from memory or stored into memory.
- **Memory Data Register (MDR):** It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.
- **Current Instruction Register (CIR):** It stores the most recently fetched instructions while it is waiting to be coded and executed.
- **Instruction Buffer Register (IBR):** The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

2) Memory Unit

Computer **memory** is any physical device capable of storing information temporarily, like RAM (random access memory), or permanently, like ROM (read-only memory).

Memories can be classified into two categories:

- Primary Memory
- Secondary Memory

a) **Primary memory** is computer **memory** that is accessed directly by the CPU. There are two types of primary memory.

- Read Only Memory (ROM)
- Random Access Memory (RAM)

ROM: The content of it cannot be changed and can be used only by CPU. It is needed to store Basic Input Output System (BIOS), which is responsible for booting. This memory is permanent in storage (non-volatile) and is very small in size.

RAM: It is a volatile memory i.e.; its contents get destroyed as soon as the computer is switched off. All kinds of processing of CPU are done in this memory.

b) Secondary Memory

Primary memory has limited storage capacity and is volatile. Secondary memory overcomes this limitation by providing permanent storage of data and in bulk quantity. Secondary memory is also termed as external memory and refers to the various storage media on which a computer can store data and programs. The Secondary storage media can be fixed or removable. Fixed Storage media is an internal storage medium like hard disk that is fixed inside the computer. Storage medium that are portable and can be taken outside the computer are termed as removable storage media.

Example: Hard disk, Magnetic Tapes, Pen drive.

3) Input /Output Unit

The input/output unit consists of devices used to transmit information between the external world and computer memory. The information fed through the input unit is stored in computer memory for processing and the result stored in memory can be recorded or display on the output medium.

Example: Mouse, Keyboard, Printer, Monitor, etc.

1.2 System Software and Application Software

Software is a set of programs, which is designed to perform a well-defined function. A program is a sequence of instructions written to solve a particular problem.

There are two types of software –

- System Software
- Application Software

a) System Software

The system software is a collection of programs designed to operate, control, and extend the processing capabilities of the computer itself. System software is generally prepared by the computer manufacturers. These software products comprise of programs written in low-level languages, which interact with the hardware at a very basic level. System software serves as the interface between the hardware and the end users.

Examples - Operating System, Compilers, Interpreter, Assemblers, etc.

Some of the most prominent features of system software –

- Close to the system
- Fast in speed

- Difficult to design
- Difficult to understand
- Less interactive
- Smaller in size
- Difficult to manipulate
- Generally written in low-level language

b) Application Software

Application software products are designed to satisfy a particular need of a particular environment. All software applications prepared in the computer lab can come under the category of Application software.

Application software may consist of a single program, such as Microsoft's notepad for writing and editing a simple text. It may also consist of a collection of programs, often called a software package, which work together to accomplish a task, such as MS Office.

Examples – Railways Reservation Software, Microsoft Office Suite Software, Microsoft Word, Microsoft Excel, Microsoft PowerPoint.

Some of the most prominent features of application software are as follows –

- Close to the user
- Easy to design
- More interactive
- Slow in speed
- Generally written in high-level language
- Easy to understand
- Easy to manipulate and use
- Bigger in size and requires large storage space

S.no	System software	Application software
1	System Software maintains the system resources and gives the path for application software to run.	Application software is built for specific tasks.
2	Low level languages are used to write the system software.	High level languages are used to write the application software.
3	Machine Dependent	Machine independent
4	It is general-purpose software.	It is specific purpose software.

5	Without system software, system can't run.	Without application software system always runs.
6	System software runs when system is turned on and when system is turned off.	While application software runs as per the user's request.
7	Compiler, Operating System, Interpreter	Photoshop, Microsoft Office, VLC

1.2.1 High-level and Low-level languages

What is a programming language?

A programming language defines a set of instructions that are compiled together to perform a specific task by the CPU (Central Processing Unit). The programming language mainly refers to high-level languages such as C, C++, Pascal, COBOL, etc.

Basically, there are two main categories of computer languages, namely Low-Level Language and High-Level Language.

1) Low Level Languages

Low level languages are the basic computer instructions or better known as machine codes. A computer cannot understand any instruction given to it by the user in English or any other high-level language. These low-level languages are very easily understandable by the machine.

The main function of low-level languages is to interact with the hardware of the computer. They help in operating, syncing, and managing all the hardware and system components of the computer. They handle all the instructions which form the architecture of the hardware systems.

a) Machine Language

This is one of the most basic low-level languages. The language was first developed to interact with the first-generation computers. It is written in binary code or machine code, which means it basically comprises of only two digits – 1 and 0.

b) Assembly Language

This is the second-generation programming language. It is a development on the machine language, where instead of using only numbers, we use English words, names, and symbols. It is the most basic computer language necessary for any processor.

2) High Level Language

When we talk about high level languages, these are programming languages. Some prominent examples are PASCAL, FORTRAN, C++ etc.

The important feature about such high-level languages is that they allow the programmer to write programs for all types of computers and systems. Every instruction in high level language is converted to machine language for the computer to comprehend.

a) Scripting Languages

Scripting languages or scripts are essentially programming languages. These languages employ a high-level construct which allows it to interpret and execute one command at a time.

Scripting languages are easier to learn and execute than compiled languages. Some examples are AppleScript, JavaScript, Pearl etc.

b) Object-Oriented Languages

These are high level languages that focus on the ‘objects’ rather than the ‘actions’. To accomplish this, the focus will be on data than logic.

The reasoning behind is that the programmers really care about the object they wish to manipulate rather than the logic needed to manipulate them. Some examples include Java, C+, C++, Python, Swift etc.

c) Procedural Programming Language

This is a type of programming language that has well-structured steps and complex procedures within its programming to compose a complete program.

It has a systematic order functions and commands to complete a task or a program. C, FORTRAN, ALGOL, BASIC, COBOL is some examples.

High level Vs Low level language

Low-level language	High-level language
It is a machine-friendly language, i.e., the computer understands the machine language, which is represented in 0 or 1.	It is a user-friendly language as this language is written in simple English words, which can be easily understood by humans.
The low-level language takes more time to execute.	It executes at a faster pace.
It requires the assembler to convert the assembly code into machine code.	It requires the compiler to convert the high-level language instructions into machine code.

The machine code cannot run on all machines, so it is not a portable language.	The high-level code can run all the platforms, so it is a portable language.
It is memory efficient.	It is less memory efficient.
Debugging and maintenance are not easier in a low-level language.	Debugging and maintenance are easier in a high-level language.

1.2.2 Compilers and Interpreters

a) Interpreter

In computer terms, the programming interpreter reads the first line of the code, interprets it into machine code and sends it to the CPU to be processed. It then interprets the next line and sends that to the CPU, and so on. By interpreting the program line by line, the program can immediately start running. However, since every line needs to be interpreted before it can run, the program will often run a bit more slowly than a compiled program. The source code is interpreted as the program is run; the interpreter (and interpreted language) must be installed on the same computer that the program is running. The examples of an interpreted language are JavaScript, Python 2, and Python 3.

b) Compiler

In contrast to an interpreter, the compiler takes the entire source code and translates it into machine code. This translated code is saved as an executable file on the computer that can be opened on any computer without requiring the computer to have a compiler. Examples of compiled languages include BASIC, C, C++, Delphi, and Java.

Compilation process of C programming passes through following stages before being transformed into an executable form:

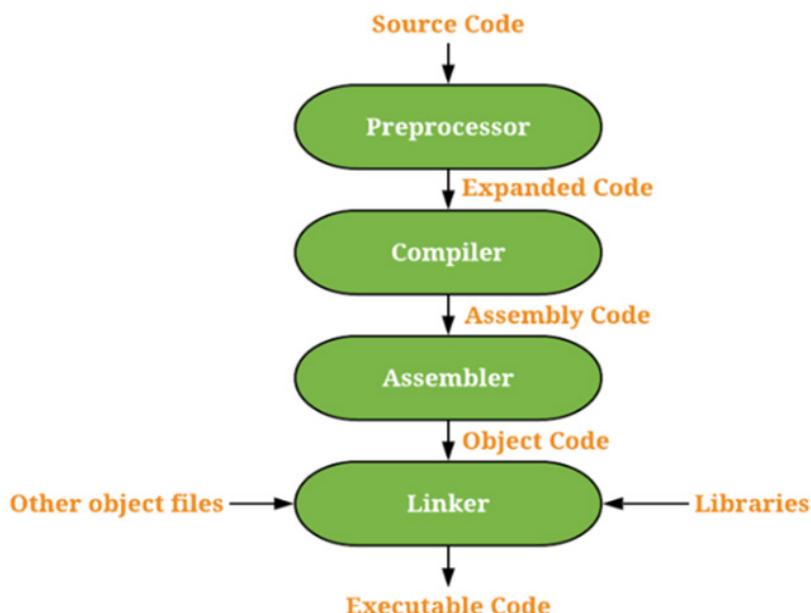
- **Pre-processor**
- **Compiler**
- **Assembler**
- **Linker**

Pre-processor: The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the pre-processor, and then the pre-processor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler: The code which is expanded by the pre-processor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler: The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj'. If the name of the source file is 'hello.c', then the name of the object file would be 'hello.obj'.

Linker: Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe'. For example, if we are using printf () function in a program, then the linker adds its associated code in an output file.



Compiler Vs Interpreter

Basis of comparison	Compiler	Interpreter
Function	A compiler converts high-level language program code into machine language and then executes it.	Interpreter converts source code into the intermediate form and then converts that intermediate code into machine language

Scanning	Compiler scans the entire program first before translating into machine code.	Interpreter scans and translates the program line by line to equivalent machine code.
Working	Compiler takes entire program as input.	Interpreter takes single instruction as input.
Code Generation	Intermediate object code is generated in case of compiler.	In case of interpreter, no intermediate object code is generated.
Execution Time	Compiler takes less execution time when compared to interpreter.	Interpreter takes more execution time when compared to compiler.
Memory Requirement	Compiler requires more memory than interpreter.	Interpreter needs less memory when compared to compiler.
Modification	If you happen to make any modification in program you have to recompile entire program i.e., scan the whole program every time after modification.	If you make any modification and if that line has not been scanned, then no need to recompile entire program.
Speed	Compiler is faster when compared to interpreter.	Interpreter is slower when compared to compiler.
At Execution	There is usually no need to compile program every time (if not modified) at execution time.	Every time program is scanned and translated at execution time.
Error Detection	Compiler gives you the list of all errors after compilation of whole program.	Interpreter stops the translation at the error generation and will continue when error get solved.
Debugging	Compiler is slow for debugging because errors are displayed after entire program has been checked.	Interpreter is good for fast debugging.
Examples	C, COBOL, C#, C++, etc	Python, VB, Java Script etc.

1.3 Introduction to structured approach to programming

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like

GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner.

The structured program mainly consists of three types of elements:

- Selection Statements
- Sequence Statements
- Iteration Statements

The structured program consists of well-structured and separated modules. But the entry and exit in a structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore, a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming Approach is well accepted in the programming world.

Advantages of Structured Programming Approach:

- Easier to read and understand
- User Friendly
- Easier to Maintain
- Mainly problem based instead of being machine based
- Development is easier as it requires less effort and time
- Easier to Debug
- Machine-Independent, mostly.

Disadvantages of Structured Programming Approach:

1. Since it is Machine-Independent, so it takes time to convert into machine code.
2. The converted machine code is not the same as for assembly language.
3. The program depends upon changeable factors like datatypes. Therefore, it needs to be updated with the need on the go.
4. Usually, the development in this approach takes longer time as it is language dependent. Whereas in the case of assembly language, the development takes lesser time as it is fixed for the machine.

1.3.1 Algorithm and Flowchart

Algorithm definition-In programming, algorithm is a set of well-defined instructions in sequence to solve the problem.

Qualities of a good algorithm

- Input and output should be defined precisely.
- Each step-in algorithm should be clear and understandable.

- Algorithm should be most effective among many ways to solve a problem.
- An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

Examples of Algorithms in Programming

1) Write an algorithm to add two numbers entered by user.

Step 1: Start
 Step 2: Declare variables num1, num2 and sum.
 Step 3: Read values num1 and num2.
 Step 4: Add num1 and num2 and assign the result to sum.

$$\text{sum} \leftarrow \text{num1} + \text{num2}$$

 Step 5: Display sum
 Step 6: Stop

2) Write an algorithm to find the largest among three different numbers entered by user.

Step 1: Start
 Start 2: Input a, b, c
 Start 3: If $a > b$ goto step 4, otherwise goto step 5
 Start 4: If $a > c$ goto step 6, otherwise goto step 8
 Start 5: if $b > c$ goto step 7, otherwise goto step 8
 Start 6: Output "a is the largest", goto step 9
 Start 7: Output "b is the largest", goto step 9
 Start 8: Output "c is the largest", goto step 9
 Start 9: Stop

3) Finding Area of the square

Step 1: Start
 Step 2: Declare the variables length and area.
 Step 3: Read the value of length.
 Step 4: Multiply length and length and assign the result to area.

$$\text{area} \leftarrow \text{length} * \text{length}$$

 Step 5: Display area
 Step 6: Stop

4) Finding Area of a square with 3 different sides

Step 1: Start

Step 2: Declare the variables a, b, c, s and Area.

Step 3: Read the value of a, b and c.

Step 4: Compute semi-perimeter and assign the value to s

$$s \leftarrow (a + b + c)/2$$

Step 4: Compute the area and assign the result to Area

$$\text{Area} \leftarrow \sqrt{s(s-a)(s-b)(s-c)}$$

Step 5: Display Area

Step 6: Stop

5) Calculating the average for 3 numbers

Step 1: Start

Step 2: Declare variables num1, num2, num3 and Average.

Step 3: Read values num1, num2, and num3.

Step 4: Add num1, num2 and num3 and divide by 3 then assign the result to Average.

$$\text{Average} \leftarrow (num1 + num2 + num3) / 2$$

Step 5: Display Average.

Step 6: Stop

6) Greatest of two numbers

Step 1: Start

Step 2: Declare variables A and B

Step 3: Read values of A and B

Step 4: If $A \geq B$

 Display A is greater than B.

 Else

 Display B is greater than A.

Step 5: Stop

7) Find Root of the quadratic equation $ax^2 + bx + c = 0$

Step 1: Start

Step 2: Declare variables a, b, c, D, r1, r2, rp and ip

Step 3: Read values of a, b and c.

Step 4: Calculate discriminant and assign the value to D

$$D \leftarrow b^2 - 4ac$$

Step 4: If $D \geq 0$

Calculate r1 and r2
 $r1 \leftarrow (-b + \sqrt{D})/2a$
 $r2 \leftarrow (-b - \sqrt{D})/2a$
 Display r1 and r2 as roots.
 Else
 Calculate real part and imaginary part
 $rp \leftarrow -b/2a$
 $ip \leftarrow \sqrt{(-D)/2a}$
 Display $rp + j(ip)$ and $rp - j(ip)$ as roots

Step 5: Stop

8) Interchange the value of two numbers

Step 1: Start
 Step 2: Declare variables a, b and c
 Step 3: Read the value of a and b
 Step 3: Perform Swapping of value as

$c = a$

$a = b$

$b = c$

Step 4: Display the value of a and b

Step 5: Stop

9) Find the factorial of a number

Step 1: Start
 Step 2: Declare variables F, N and i
 Step 3: Initialize the value of F=1 and i=1
 Step 4: Read the value of N
 Step 5: Multiply F with i and assign the value to F

$F = F * i$

Step 6: Increment i by 1

Step 7: Repeat step 5 & 6 until $i = N$

Step 8: Display the value of F

Step 9: Stop

10) Find the Sum of Natural Numbers till N

Step 1: Start
 Step 2: Declare variables count, N and sum

Step 3: Read the value of N
Step 4: Initialize count = 1, sum = 0
Step 5: count = count + 1
Step 6: sum = sum + count
Step 7: Repeat step 5 & 6 until count \leq N
Step 8: Display the value of sum
Step 9: Stop

11) Find the Sum of all Even Natural Numbers till N

Step 1: Start
Step 2: Declare variables count, N and sum
Step 3: Read the value of N
Step 4: Initialize count = 0, sum = 0
Step 5: count = count + 2
Step 6: sum = sum + count
Step 7: Repeat step 5 & 6 until count \leq N
Step 8: Display the value of sum
Step 9: Stop

12) Find the Fibonacci series till the term less than 1000

Step 1: Start
Step 2: Declare variables first_term, second_term and temp.
Step 3: Initialize variables first_term=0, second_term = 1
Step 4: Display first_term and second_term
Step 5: temp = second_term
Step 6: second_term = second_term + first_term
Step 7: first_term = temp
Step 8: Display second_term
Step 9: Repeat the steps 5 to 8 until second_term \leq 1000
Step 10: Stop

Advantages of algorithm

- It is a stepwise representation of a solution to a given problem, which makes it easy to understand.
- An algorithm uses a definite procedure.

- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.
- By using algorithm, the problem is broken down into smaller pieces or steps hence; it is easier for programmer to convert it into an actual program.

Disadvantages of algorithm.

- Writing algorithm takes a long time.
- An Algorithm is not a computer program; it is rather a concept of how a program should be.

1.3.2 Classification of Algorithms

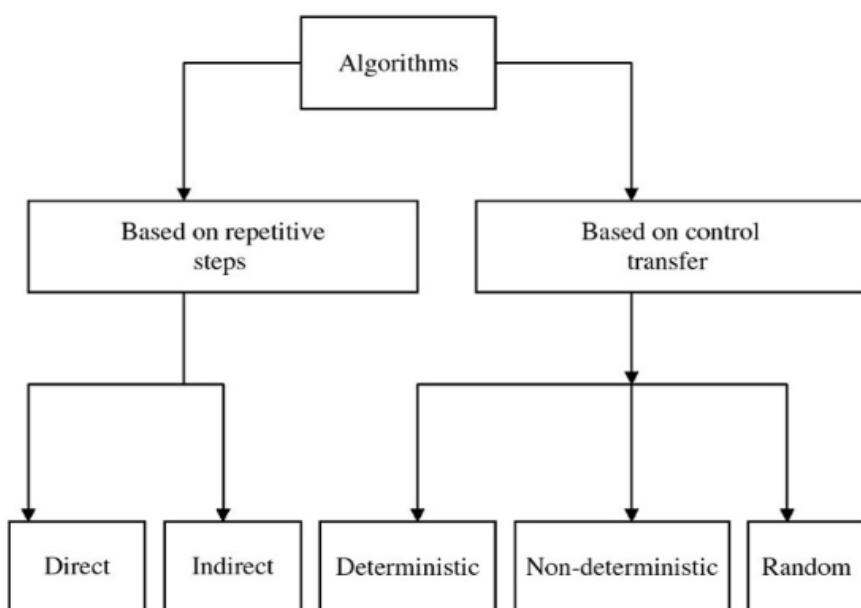
The classification of the algorithm is based on repetitive steps and on control transfer from one statement to another.

On the basis of repetitive steps, an algorithm can further be classified into two types.

a) Direct Algorithm: - In this type of algorithm, the number of iterations is known in advance. For example, for displaying numerical numbers from 1 to 10, the loop variable should be initialized from 1 to 10. The statement would be as follows:

`for (j=1; j<=10; j++)`

In the above statement, it is predicted that the loop will iterate 10 times.



Classification of Algorithms

b) Indirect Algorithm: - In this type of algorithm repetitively steps are executed. Exactly how many repetitions are to be made is unknown.

For example, the repetitive steps are as follow:

i) To find the first Armstrong numbers from 1 to n, where n is the fifth Armstrong number.

ii) To find the first three palindrome numbers.

Based on the control transfer, the algorithms are categorized in the following three types.

a) Deterministic: - Deterministic algorithm is based on either to follow a 'yes' path or 'no' path based on the condition. In this type of algorithm when control comes across a decision logic, two paths 'yes' or 'no' are shown. Program control follows one of the routes depending upon the condition.

Example:

Testing whether a number is even or odd. Testing whether a number is positive or negative.

b) Non-deterministic: - In this type of algorithm to reach the solution, we have one of the multiple paths.

Example:

To find a day of the week.

c) Random algorithm: - After executing a few steps, the control of the program transfer to another step randomly, which is known as a random algorithm.

Example:

A random search

1.3.3 Flowchart definition-is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart Symbols

Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.

	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.

Guidelines for Developing Flowcharts

These are some points to keep in mind while developing a flowchart –

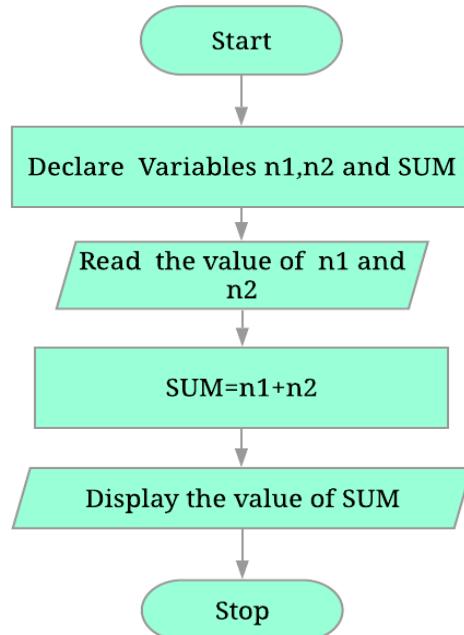
- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers
- Off-page connectors are referenced using alphabets
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

Advantages of flowchart:

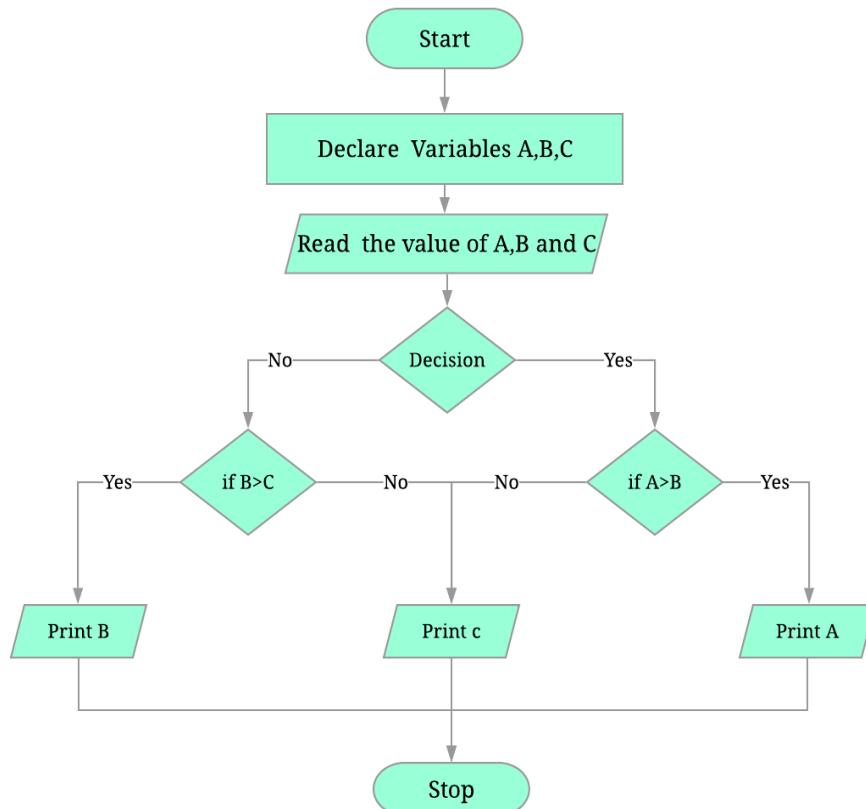
1. The Flowchart is an excellent way of communicating the logic of a program.
2. It is easy and efficient to analyse problem using flowchart.
3. During program development cycle, the flowchart plays the role of a guide or a blueprint. Which makes program development process easier?
4. After successful development of a program, it needs continuous timely maintenance during its operation. The flowchart makes program or system maintenance easier.
5. It helps the programmer to write the program code.

Examples of flowcharts in programming

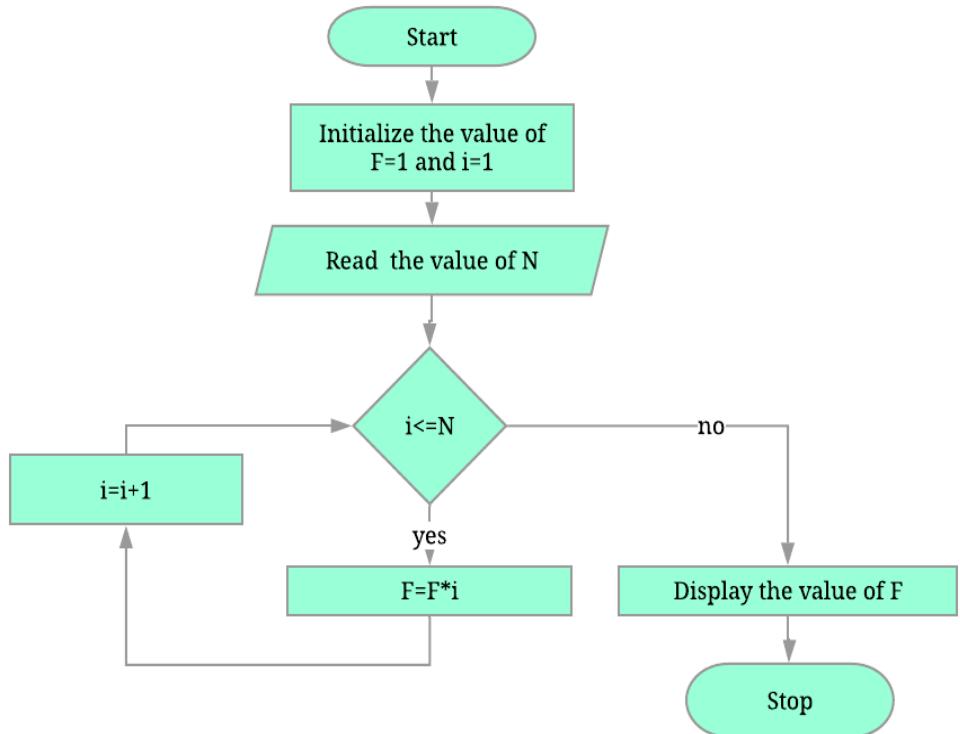
1. Add two numbers entered by the user.



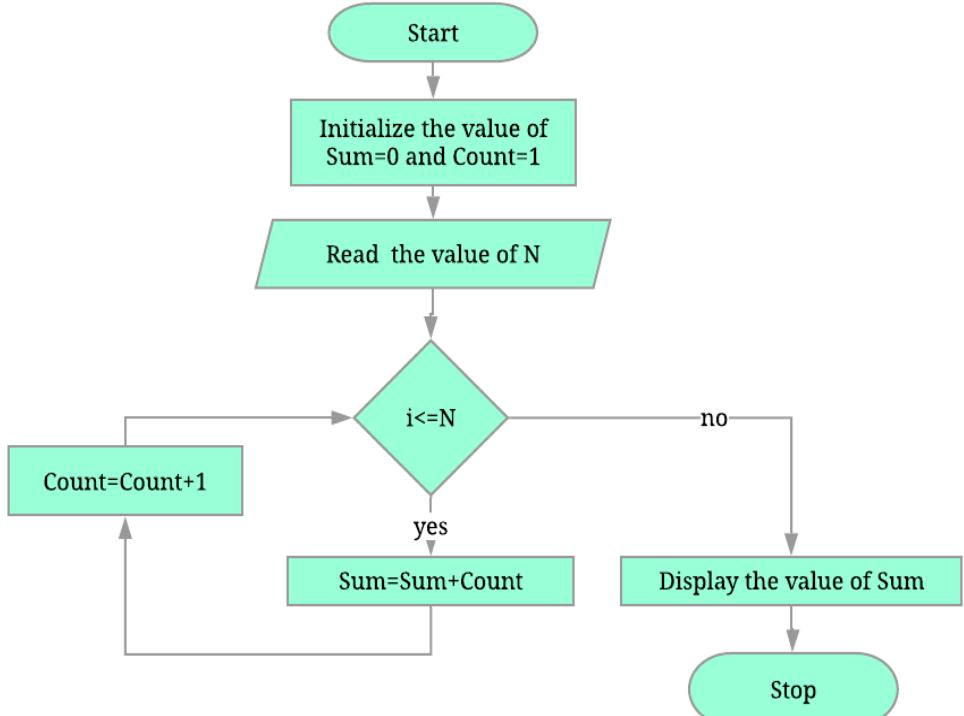
2. Find the largest among three different numbers entered by user.



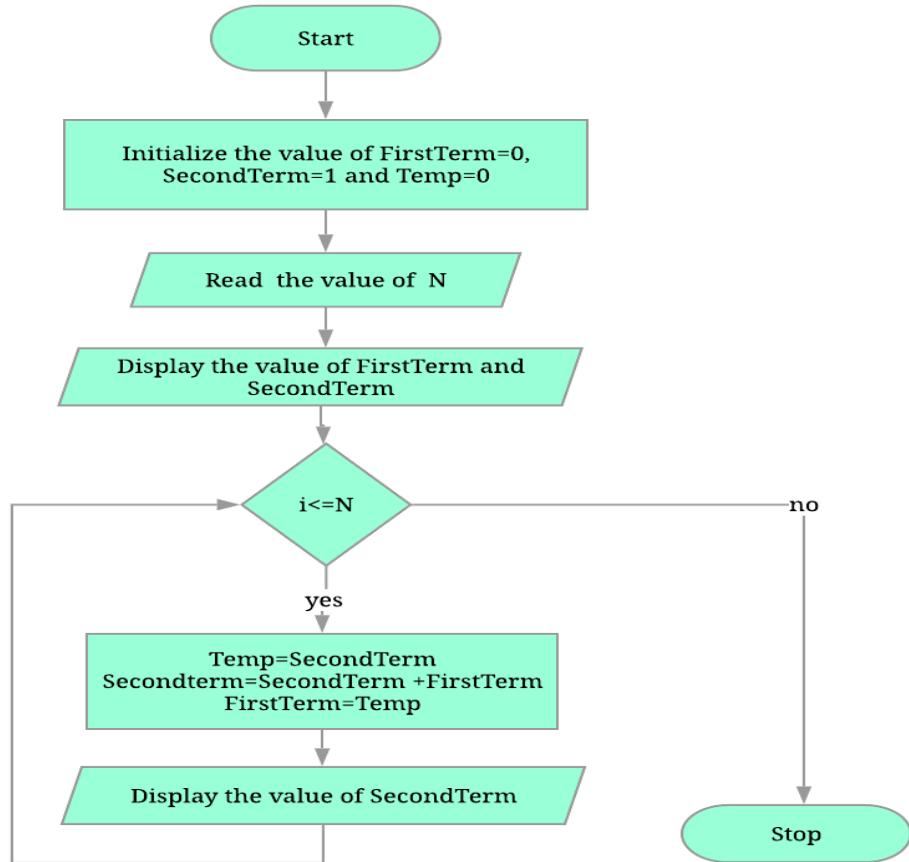
3. Find the factorial of a number entered by user.



4. Find the sum of N natural numbers entered by user.



5. Find the Fibonacci series till the term N.



1.4 Pseudo code

Pseudo code in C is a simple way to write programming code in English. Pseudo-code is informal writing style for program algorithm independent from programming languages to show the basic concept behind the code.

Pseudocode is not an actual programming language. So, it cannot be compiled and not be converted into an executable program. It uses short or simple English language syntax to write code for programs before it is converted into a specific programming language.

Guidelines for writing pseudo code:

- Write one statement per line
- Capitalize initial keyword
- Indent to hierarchy
- End multiline structure
- Keep statements language independent

1.4.1 Common keywords used in pseudo code

The following gives common keywords used in pseudo codes.

- a) //: This keyword used to represent a comment.
- b) BEGIN, END: Begin is the first statement and end are the last statement.
- c) INPUT, GET, READ: The keyword is used to inputting data.
- d) COMPUTE, CALCULATE: used for calculation of the result of the given expression.
- e) ADD, SUBTRACT, INITIALIZE used for addition, subtraction and initialization.
- f) OUTPUT, PRINT, DISPLAY: It is used to display the output of the program.
- g) IF, ELSE, ENDIF: used to make decision.
- h) WHILE ENDWHILE: used for iterative statements.
- i) FOR, ENDFOR: Another iterative incremented/decremented tested automatically.

Syntax for if else:

```
IF (condition)THEN
```

```
Statement
```

```
...
```

```
ELSE
```

```
Statement
```

```
...
```

```
ENDIF
```

Example: Greatest of two numbers

```
BEGIN
```

```
READ a,b
```

```
IF (a>b) THEN
```

```
DISPLAY a is greater
```

```
ELSE
```

```
DISPLAY b is greater
```

```
END IF
```

```
END
```

Syntax for for:

```
FOR ( start-value to end-value) DO
```

```
Statement
```



```
...  
ENDFOR
```

Example: Print n natural numbers

```
BEGIN  
GET n  
INITIALIZE i=1  
FOR (i<=n) DO  
PRINT i  
i=i+1  
ENDFOR  
END
```

Syntax for While:

```
WHILE (condition) DO  
Statement  
...  
ENDWHILE
```

Example: Print n natural numbers

```
BEGIN  
GET n  
INITIALIZE i=1  
WHILE (i<=n) DO  
PRINT i  
i=i+1  
ENDWHILE  
END
```

Advantages:

- Pseudo is independent of any language; it can be used by most programmers.
- It is easy to translate pseudo code into a programming language.
- It can be easily modified as compared to flowchart.
- Converting a pseudo code to programming language is very easy as compared with converting a flowchart to programming language.

Disadvantages:

- It does not provide visual representation of the program's logic.
- There are no accepted standards for writing pseudo codes.
- It cannot be compiled nor executed.
- For a beginner, it is more difficult to follow the logic or write pseudo code as compared to flowchart.

Examples of pseudocode

Example1:

Write a pseudocode to find the largest among three different numbers entered by user.

```
BEGIN
  READ the value of a, b, c.
  IF (a>b && a>c) THEN
    DISPLAY a is largest
  ELSEIF (b>a && b>c) THEN
    DISPLAY b is greater
  ELSE
    DISPLAY c is greater
  END IF
END
```

Example 2:

Write a pseudocode to find the sum of n natural numbers.

```
BEGIN
  INITIALIZE i=1, sum=0.
  READ the value of n
  FOR (i<=n) DO
    CALCULATE sum=sum+i.
    INCREMENT i=i+1
  ENDFOR
  DISPLAY the value of sum.
END
```

Example 3:

Write a pseudocode to find roots of a quadratic equation.

```
BEGIN
  READ values of a, b and c.
```

```

CALCULATE D=b*b - 4*a*c
IF (D ≥ 0) THEN
  CALCULATE r1 and r2
    r1 = (-b+√D)/2a
    r2 = (-b-√D)/2a
  DISPLAY r1 and r2
ELSE
  CALCULATE real part and imaginary part
    rp = -b/2a
    ip = √(-D)/2a
  DISPLAY rp + j(ip) and rp - j(ip) as roots
ENDIF
END

```

Example 4:

Write a pseudocode to find factorial for a given number.

```

BEGIN
  INITIALIZE F=1, i=1.
  READ the value of N
  FOR (i=N) DO
    CALCULATE F=F*i
    INCREMENT i=i+1
  ENDFOR
  DISPLAY the value of F.
END

```

1.4.2 Linear search in C

Linear search in C to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

Linear search is implemented using following steps...

- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with the first element in the list.
- **Step 3** - If both are matched, then display "Given element is found!!!" and terminate the function

- **Step 4** - If both are not matched, then compare search element with the next element in the list.
- **Step 5** - Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6** - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Linear search algorithm

Linear Search (Array A, Value x)

Step 1: Start

Step 2: Declare an array A of size n

Step 3: Declare variables flag, i, x

Step 4: Initialize flag=0 and i=0

Step 5: input the number to be searched x

Step 6: If A[i] equal to x

 set flag=1

 break out of loop

 Else

 set flag=0

Step 7: Increment i=i+1

Step 8: Repeat the step 6 to 7 till i<n

Step 9: If flag equal to 0

 Display The number is not found in the array

Step 10: Else

 Display The number is found in the array

Step 11: Stop

Linear search pseudocode

```

BEGIN
INITIALIZE array A of size n, i=1, flag=0.
READ the value of x.
FOR (i<n) DO
IF (A[i] = x) THEN
SET flag= 1
BREAK

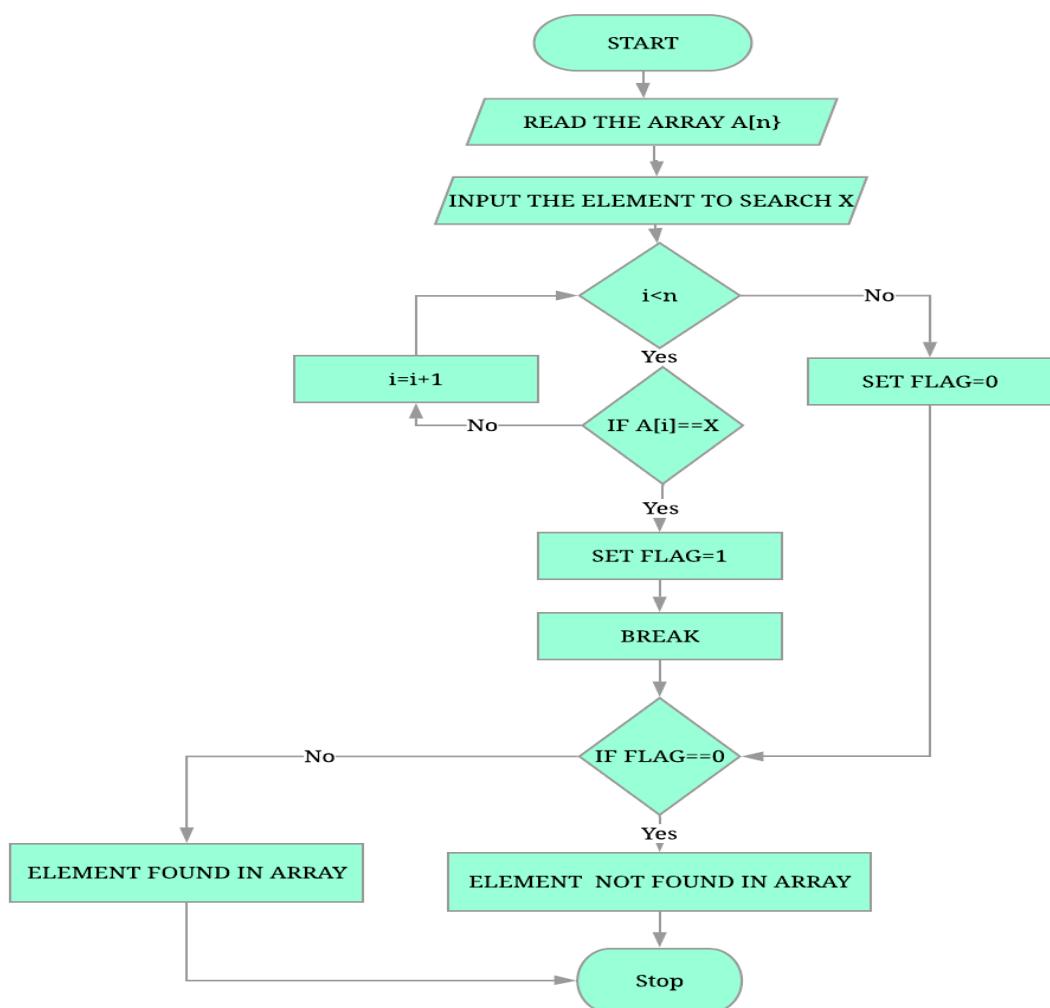
```

```

ELSE
SET flag= 0
END IF
INCREMENT i=i+1
END FOR
IF (flag== 0) THEN
DISPLAY The number is not found in the array
ELSE
DISPLAY The number is found in the array
END IF
END

```

Linear search Flowchart



1.4.3 Bubble sort in C

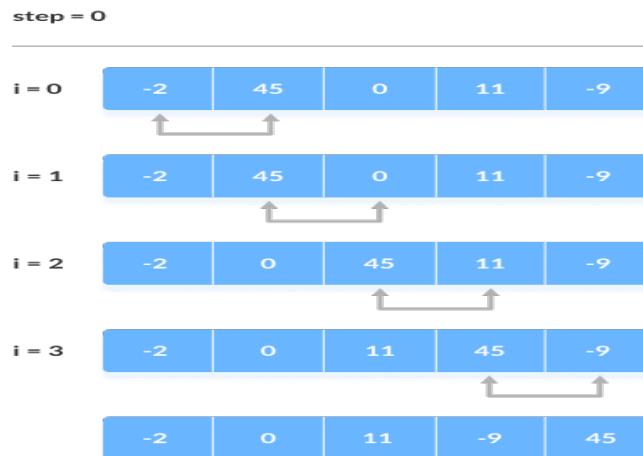
It is a simple sorting **algorithm** that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Working of Bubble Sort

Suppose we are trying to sort the elements in **ascending order**.

1. First Iteration (Compare and Swap)

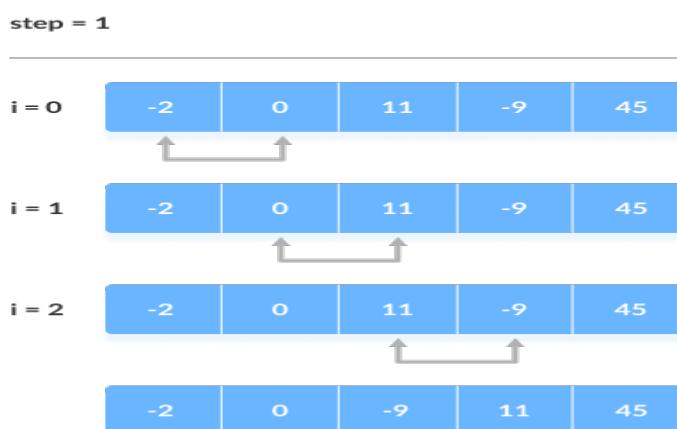
1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.



2. Remaining Iteration

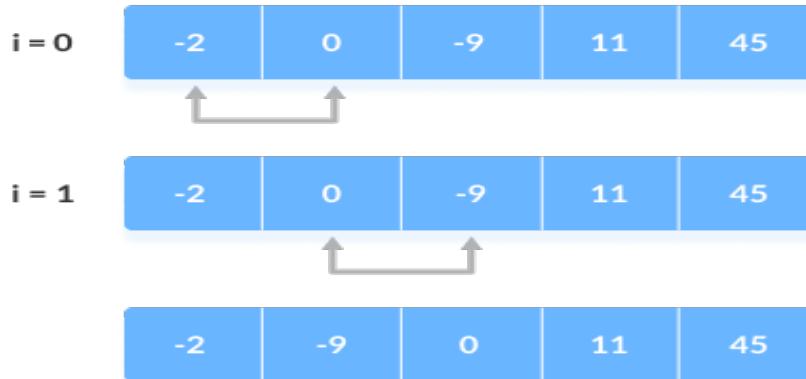
The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.



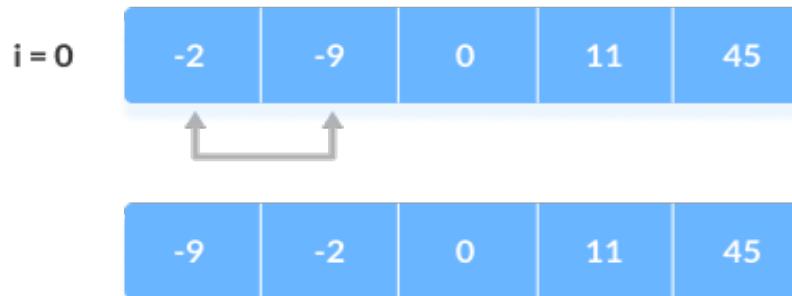
In each iteration, the comparison takes place up to the last unsorted element.

step = 2



The array is sorted when all the unsorted elements are placed at their correct positions.

step = 3



Bubble sort Algorithm

Bubble Sort (Array A, Value n)

Step 1: Start

Step 2: Declare an array A of size n

Step 3: Declare variables temp, i, j

Step 4: Initialize i=0 and j=0

Step 5: Input the value of array

Step 6: if(A[j] > A[j+1])

 temp=A[j]

 A[j]=A[j+1]

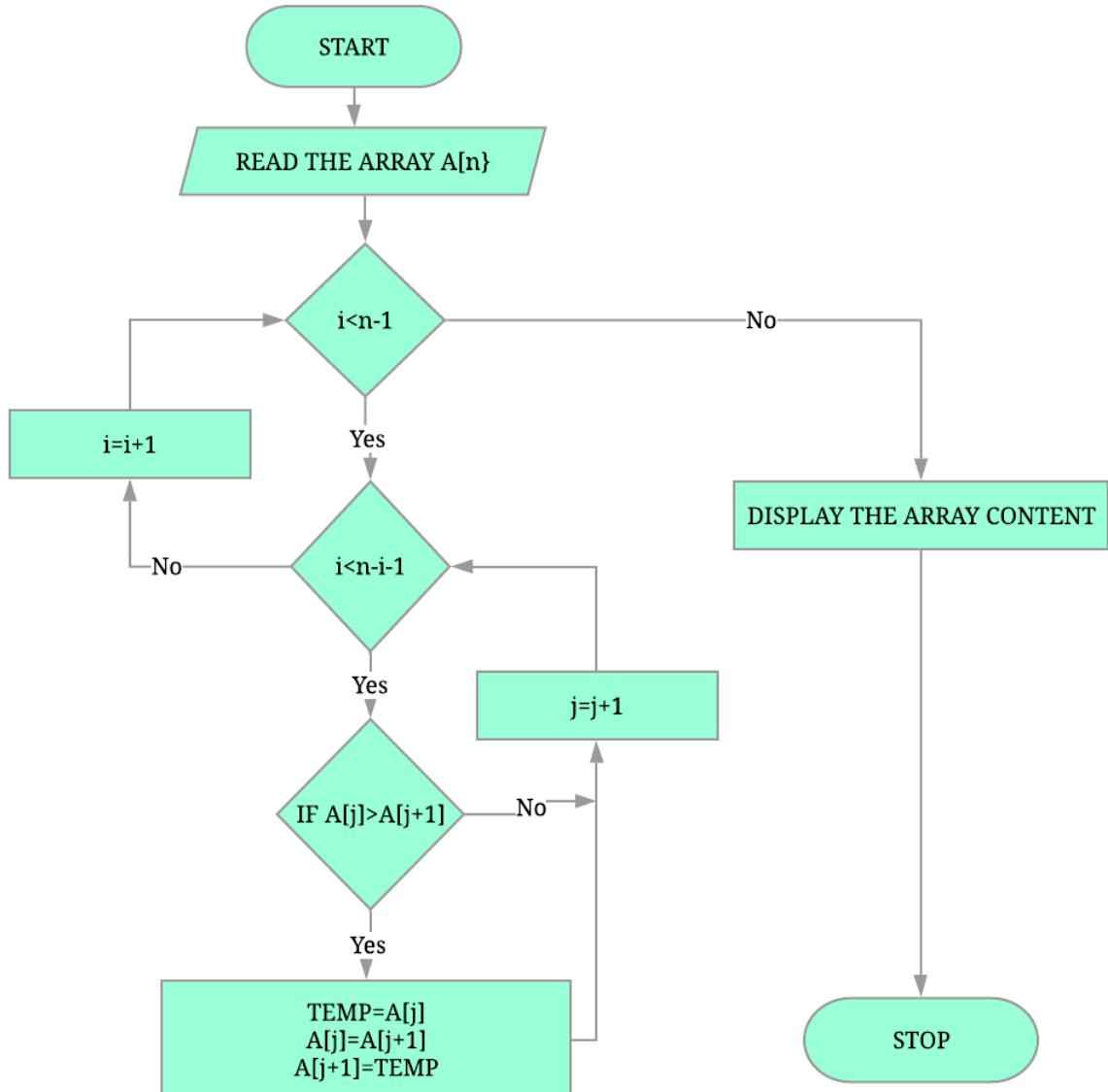
 A[j+1]=temp

Step 7: Increment $j=j+1$
Step 8: Repeat the step 6 to 7 till $j < n-i-1$
Step 9: Increment $i=i+1$
Step 10: Repeat the step 6 to 9 till $i < n-1$
Step 11: Display the sorted value of array.
Step 12: Stop

Bubble Sort pseudocode

```
BEGIN
    INITIALIZE array A of size n, i=0, j=0.
    READ the value of array.
    FOR (i<n-1) DO
        FOR (j<n-i-1) DO
            IF (A[j] > A[j+1]) THEN
                temp=A[j]
                A[j]=A[j+1]
                A[j+1] =temp
            ENDIF
            INCREMENT j=j+1
        ENDFOR
        INCREMENT i=i+1
    ENDFOR
    DISPLAY The sorted value of array
END
```

Bubble Sort Flowchart



THIS PAGE IS INTENTIONALLY LEFT BLANK

Module II

C Programming Basics	
2.1	Basic structure of C program: Character set, Tokens, Identifiers in C, Variables and Data Types, Constants, Console IO Operations, printf and scanf
2.2	Operators and Expressions: Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, size of operator, Assignment operators and Bitwise Operators. Operators Precedence
2.3	Control Flow Statements: If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements. (Simple programs covering control flow)

Brief History of C Programming Language



What is C?

C is a general-purpose structured programming language that is machine-independent and extensively used to write various applications, Operating Systems like Windows, and many other complex programs like Oracle database, Git, Python interpreter, and more.

It was initially developed by Dennis Ritchie in the year 1972 at AT&T Bell Laboratories (USA). Later on in 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard.

It is said that ‘C’ is **The God’s programming language** as knowledge of this language can help one to easily grasp the knowledge of other programming languages that uses the concept of ‘C’.

Features of C Programming Language

a) High-Level Language

C provides strong abstraction in case of its libraries and built-in features that make it machine-independent. It is capable enough to develop system applications such as the kernel, driver, etc.

b) Structured Language

C Language supports structured programming which includes the use of functions. Functions reduce the code complexity and are totally reusable.

c) Rich Library

Unlike its predecessors, C language incorporates multiple built-in arithmetic and logical functions along with many built-in libraries which make development faster and convenient.

d) Extensible

C Language is a High-level language and is also open for upgrades. Hence, the programming language is considered to be extensible like any other high-level languages.

e) Recursion

C Language supports function recursion where a function is called within another function for multiple numbers of times.

f) Pointers

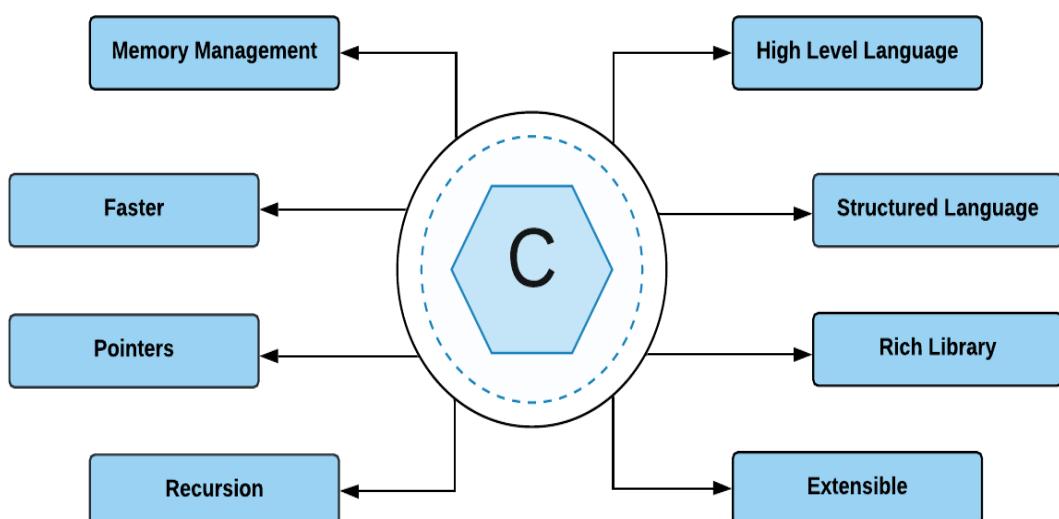
C enables users to directly interact with memory using the pointers. We use pointers in memory, structure, functions, arrays, stack and many more.

g) Faster

C Language comes with a minimal number of libraries and built-in functions which makes the compile and execution-times to be less and the system faces low overhead.

h) Memory Management

C provides the best-in-class memory management. It can both allocate and deallocate memory dynamically. The malloc(), calloc(), realloc() functions are used to allocate memory dynamically and free() function is used to deallocate the used memory at any instance of time.



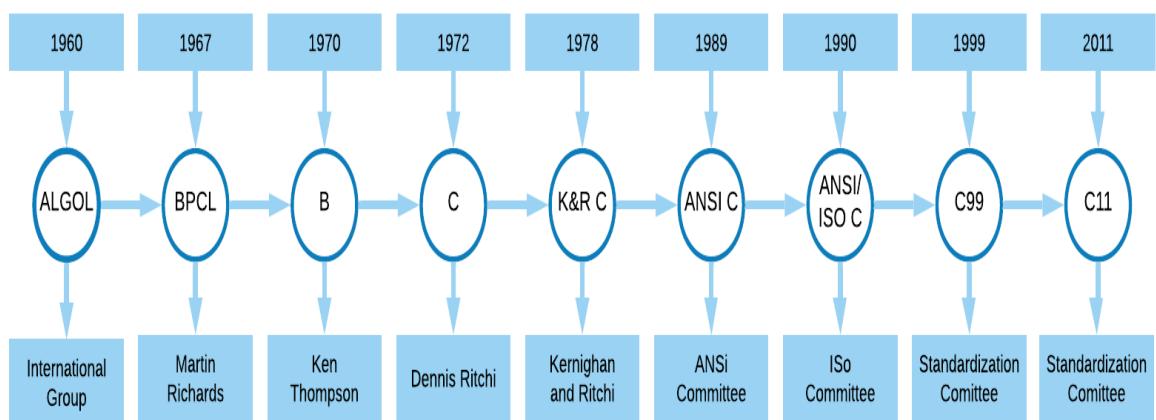
Advantage of C:

- C is the building block for many other programming languages.
- Programs written in C are highly portable.
- Several standard functions are there (like in-built) that can be used to develop programs.
- C programs are collections of C library functions, and it's also easy to add functions to the C library.
- The modular structure makes code debugging, maintenance, and testing easier.

Disadvantage of C:

- C does not provide Object Oriented Programming (OOP) concepts.
- There are no concepts of Namespace in C.
- C does not provide binding or wrapping up of data in a single unit.
- C does not provide Constructor and Destructor.

Given below is an image of the detailed history of how C evolved.



Why should an Engineer learn C programming?

We live in an era of Embedded Systems which by definition, is an integrated system, due to its combination of hardware and software (also known as Firmware). These devices may be a smartphone, smart watches, smart home devices, medical equipment, security alarms, IoT products, etc.

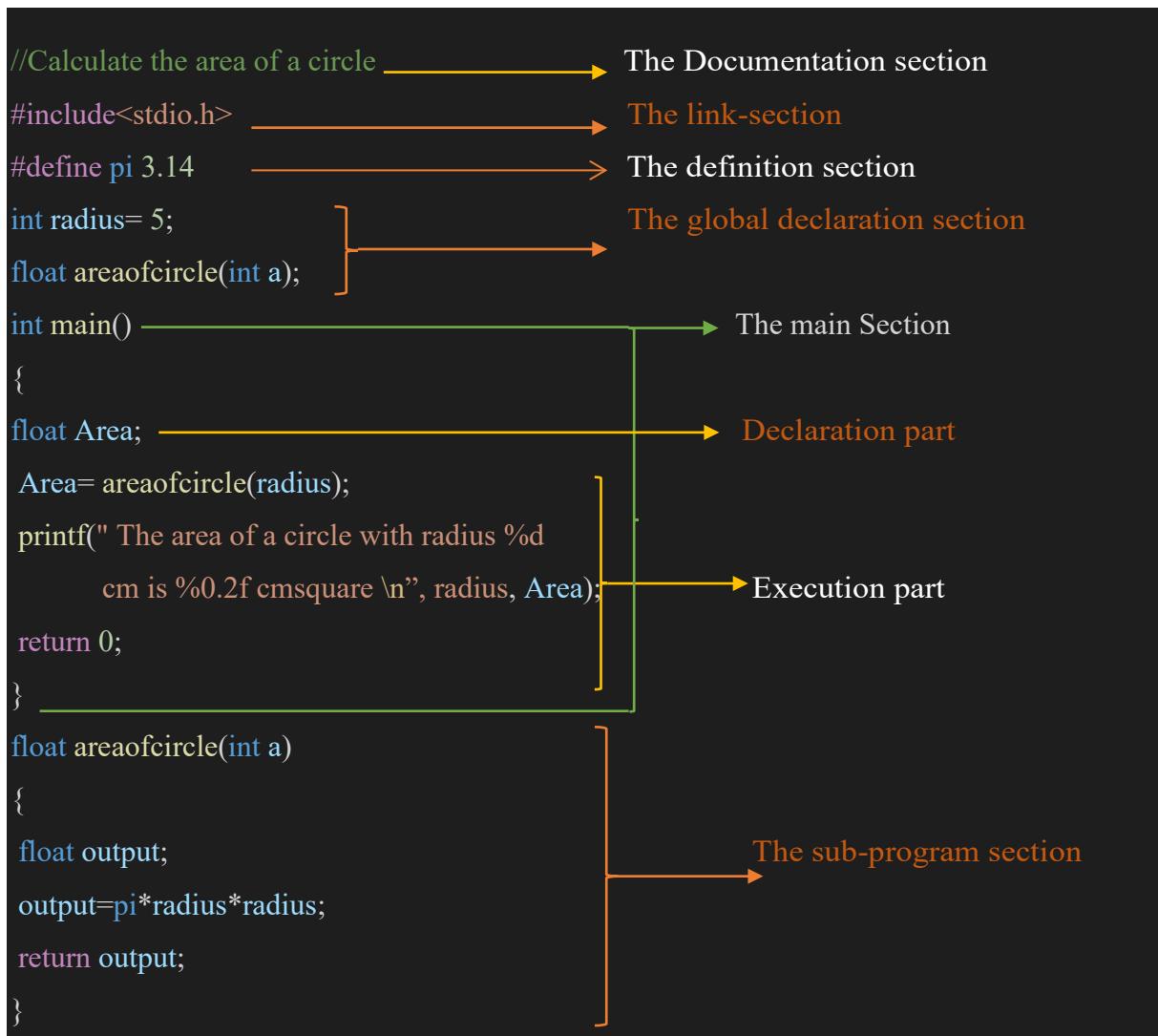
The heart of these devices is referred to as a Micro-controller, which is basically programmed using C language. So, learning this language will help an Engineering graduate to procure a job in Embedded industries which widely constitute Auto-motives, Robotics, Intelligent system design, Space exploration, IoT product design etc.

2.1 Basic structure of a C program

A C program structure involves the following sections:

- **Documentations (Documentation Section)**
- **Preprocessor Statements (Link Section)**
- **Definition Section**
- **Global Declarations**
- **The main() function**
 - **Declarations**
 - **Execution part**
- **User Defined Functions (Sub-program section)**

The Structure of a C program



- **The Documentation section** usually contains the collection of comment lines giving the name of the program, author's or programmer's name etc.

- The second part is **the link-section** which instructs the compiler to connect to the various functions from the system library.
- **The definition section** defines all symbolic constants such by using the #define directive.
- **The global declaration section** is used to define those variables that are used globally within the entire program and is used in more than one function. This section also declares all the user-defined functions.
- All C programs must have a **main()** which contains two parts:
 - **Declaration part:** The declaration part is the part where all the variables are declared.
 - **Execution part:** The execution part begins with the curly brackets and ends with the curly close bracket. Both the declaration and execution part are inside the curly braces.
- **The sub-program section** deals with all user-defined functions that are called from the main(). These user-defined functions are declared and usually defined after the main() function.

Let us look at a simple code that would print the words "Hello World" –

```
// MY first program in C
#include <stdio.h>
int main() {
    printf("Hello World! \n");
    return 0;
}
```

OUTPUT

Hello World!

If we take a look at the various parts of the above program –

- The first line `/*...*/` will be ignored by the compiler and it has been put to add additional comments in the program. So, such lines are called comments in the program.
- The next line of the program `#include <stdio.h>` is a Preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
- The next line `int main()` is the main function where the program execution begins.

- The next line printf(...) is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
- The next line return 0; terminates the main () function and returns the value 0.

2.1.1 Character set in C

What is a Character set?

A *character set* defines the valid characters that can be used in the source programs or interpreted when a program is running.

As every language contains a set of characters used to construct words, statements, etc., C language also has a set of 256 characters.

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set.

C character set is divided into four categories,

- Letters
- Numbers
- Special characters
- White spaces (blank spaces)

A compiler always ignores the use of characters, but it is widely used for formatting the data.

Following is the character set in 'C' programming:

1. Letters
 - Uppercase characters (A-Z)
 - Lowercase characters (a-z)
2. Numbers
 - All the digits from 0 to 9
3. White spaces
 - Blank space
 - New line
 - Carriage return
 - Horizontal tab
4. Special characters
 - Special characters in 'C' are shown in the given table,

1	, (comma)	16	{(opening curly bracket)
2	. (period)	17	} (closing curly bracket)

3	;(semi-colon)	18	[(left bracket)
4	: (colon)	19] (right bracket)
5	? (question mark)	20	((opening left parenthesis)
6	' (apostrophe)	21) (closing right parenthesis)
7	" (double quotation mark)	22	& (ampersand)
8	! (exclamation mark)	23	^ (caret)
9	(vertical bar)	24	+ (addition)
10	/ (forward slash)	25	- (subtraction)
11	\ (backward slash)	26	* (multiplication)
12	~ (tilde)	27	/ (division)
13	_ (underscore)	28	> (greater than or closing angle bracket)
14	\$ (dollar sign)	29	< (less than or opening angle bracket)
15	% (percentage sign)	30	# (hash sign)

In C programming, a character variable holds ASCII (**American Standard Code for Information Interchange**) value (an integer number between 0 and 127) rather than the character itself. This integer value is the ASCII code of the character. When a key is pressed on the keyboard, the computer stores the ASCII representation of the character typed into main memory.

The following table shows the range of ASCII values for various characters:

Character A – Z: ASCII Value 65 – 90

Character a – z: ASCII Value 97 – 122

Character 0 – 9: ASCII Value 48 – 57

Special Symbol: ASCII Value 0 – 47, 58 – 64, 91 – 96, 123 – 127

ASCII Code	ASCII Character	ASCII Character Name	Description
32		Space	ASCII Value of 'Space' is 32
33	!	Exclamation mark	ASCII Value of 'Exclamation mark' is 33
34	“	Double quotes (or speech marks)	ASCII Value of 'Double quotes (or speech marks)' is 34
35	#	Number	ASCII Value of 'Number' is 35
36	\$	Dollar	ASCII Value of 'Dollar' is 36

37	%	Per cent sign	ASCII Value of 'Per cent sign' is 37
38	&	Ampersand	ASCII Value of 'Ampersand' is 38
39	'	Single quote	ASCII Value of 'Single quote' is 39
40	(Open parenthesis (or open bracket)	ASCII Value of 'Open parenthesis (or open bracket)' is 40
41)	Close parenthesis (or close bracket)	ASCII Value of 'Close parenthesis (or close bracket)' is 41
42	*	Asterisk	ASCII Value of 'Asterisk' is 42
43	+	Plus	ASCII Value of 'Plus' is 43
44	,	Comma	ASCII Value of 'Comma' is 44
45	-	Hyphen	ASCII Value of 'Hyphen' is 45
46	.	Period, dot or full stop	ASCII Value of 'Period, dot or full stop' is 46
47	/	Slash or divide	ASCII Value of 'Slash or divide' is 47
48	0	Zero	ASCII Value of 'Zero' is 48
49	1	One	ASCII Value of 'One' is 49
50	2	Two	ASCII Value of 'Two' is 50
51	3	Three	ASCII Value of 'Three' is 51
52	4	Four	ASCII Value of 'Four' is 52
53	5	Five	ASCII Value of 'Five' is 53
54	6	Six	ASCII Value of 'Six' is 54
55	7	Seven	ASCII Value of 'Seven' is 55
56	8	Eight	ASCII Value of 'Eight' is 56
57	9	Nine	ASCII Value of 'Nine' is 57
58	:	Colon	ASCII Value of 'Colon' is 58
59	;	Semicolon	ASCII Value of 'Semicolon' is 59
60	<	Less than (or open angled bracket)	ASCII Value of 'Less than (or open angled bracket)' is 60
61	=	Equals	ASCII Value of 'Equals' is 61
62	>	Greater than (or close angled bracket)	ASCII Value of 'Greater than (or close angled bracket)' is 62
63	?	Question mark	ASCII Value of 'Question mark' is 63
64	@	At symbol	ASCII Value of 'At symbol' is 64
65	A	Uppercase A	ASCII Value of 'Uppercase A' is 65
66	B	Uppercase B	ASCII Value of 'Uppercase B' is 66
67	C	Uppercase C	ASCII Value of 'Uppercase C' is 67
68	D	Uppercase D	ASCII Value of 'Uppercase D' is 68
69	E	Uppercase E	ASCII Value of 'Uppercase E' is 69
70	F	Uppercase F	ASCII Value of 'Uppercase F' is 70
71	G	Uppercase G	ASCII Value of 'Uppercase G' is 71
72	H	Uppercase H	ASCII Value of 'Uppercase H' is 72
73	I	Uppercase I	ASCII Value of 'Uppercase I' is 73
74	J	Uppercase J	ASCII Value of 'Uppercase J' is 74

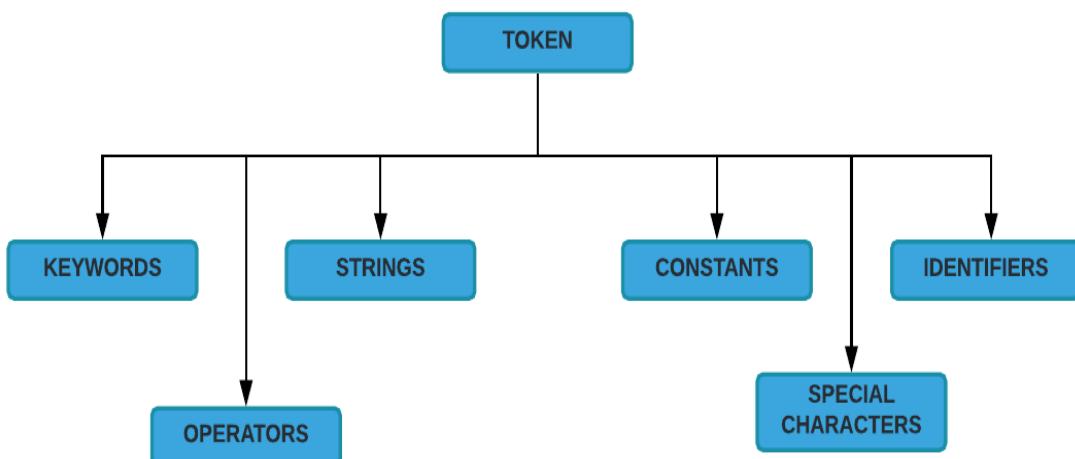
75	K	Uppercase K	ASCII Value of 'Uppercase K' is 75
76	L	Uppercase L	ASCII Value of 'Uppercase L' is 76
77	M	Uppercase M	ASCII Value of 'Uppercase M' is 77
78	N	Uppercase N	ASCII Value of 'Uppercase N' is 78
79	O	Uppercase O	ASCII Value of 'Uppercase O' is 79
80	P	Uppercase P	ASCII Value of 'Uppercase P' is 80
81	Q	Uppercase Q	ASCII Value of 'Uppercase Q' is 81
82	R	Uppercase R	ASCII Value of 'Uppercase R' is 82
83	S	Uppercase S	ASCII Value of 'Uppercase S' is 83
84	T	Uppercase T	ASCII Value of 'Uppercase T' is 84
85	U	Uppercase U	ASCII Value of 'Uppercase U' is 85
86	V	Uppercase V	ASCII Value of 'Uppercase V' is 86
87	W	Uppercase W	ASCII Value of 'Uppercase W' is 87
88	X	Uppercase X	ASCII Value of 'Uppercase X' is 88
89	Y	Uppercase Y	ASCII Value of 'Uppercase Y' is 89
90	Z	Uppercase Z	ASCII Value of 'Uppercase Z' is 90
91	[Opening bracket	ASCII Value of 'Opening bracket' is 91
92	\	Backslash	ASCII Value of 'Backslash' is 92
93]	Closing bracket	ASCII Value of 'Closing bracket' is 93
94	^	Caret – circumflex	ASCII Value of 'Caret – circumflex' is 94
95	_	Underscore	ASCII Value of 'Underscore' is 95
96	`	Grave accent	ASCII Value of 'Grave accent' is 96
97	a	Lowercase a	ASCII Value of 'Lowercase a' is 97
98	b	Lowercase b	ASCII Value of 'Lowercase b' is 98
99	c	Lowercase c	ASCII Value of 'Lowercase c' is 99
100	d	Lowercase d	ASCII Value of 'Lowercase d' is 100
101	e	Lowercase e	ASCII Value of 'Lowercase e' is 101
102	f	Lowercase f	ASCII Value of 'Lowercase f' is 102
103	g	Lowercase g	ASCII Value of 'Lowercase g' is 103
104	h	Lowercase h	ASCII Value of 'Lowercase h' is 104
105	i	Lowercase i	ASCII Value of 'Lowercase i' is 105
106	j	Lowercase j	ASCII Value of 'Lowercase j' is 106
107	k	Lowercase k	ASCII Value of 'Lowercase k' is 107
108	l	Lowercase l	ASCII Value of 'Lowercase l' is 108
109	m	Lowercase m	ASCII Value of 'Lowercase m' is 109
110	n	Lowercase n	ASCII Value of 'Lowercase n' is 110
111	o	Lowercase o	ASCII Value of 'Lowercase o' is 111
112	p	Lowercase p	ASCII Value of 'Lowercase p' is 112
113	q	Lowercase q	ASCII Value of 'Lowercase q' is 113
114	r	Lowercase r	ASCII Value of 'Lowercase r' is 114
115	s	Lowercase s	ASCII Value of 'Lowercase s' is 115
116	t	Lowercase t	ASCII Value of 'Lowercase t' is 116

117	u	Lowercase u	ASCII Value of 'Lowercase u' is 117
118	v	Lowercase v	ASCII Value of 'Lowercase v' is 118
119	w	Lowercase w	ASCII Value of 'Lowercase w' is 119
120	x	Lowercase x	ASCII Value of 'Lowercase x' is 120
121	y	Lowercase y	ASCII Value of 'Lowercase y' is 121
122	z	Lowercase z	ASCII Value of 'Lowercase z' is 122
123	{	Opening brace	ASCII Value of 'Opening brace' is 123
124		Vertical bar	ASCII Value of 'Vertical bar' is 124
125	}	Closing brace	ASCII Value of 'Closing brace' is 125
126	~	Equivalency sign – tilde	ASCII Value of 'Equivalency sign – tilde' is 126
127		Delete	ASCII Value of 'Delete' is 127

2.1.2 Tokens in C

What is Token in C?

TOKEN is the smallest unit in a 'C' program. It is each and every word and punctuation that you come across in your C program. The compiler breaks a program into the smallest possible units (tokens) and proceeds to the various stages of the compilation. A token is divided into six different types, viz, Keywords, Operators, Strings, Constants, Special Characters, and Identifiers.



Tokens in C

2.1.2.1 Keywords

Keyword can be defined as the pre-defined or the reserved words having its own importance, and each keyword has its own functionality. Since keywords are the pre-defined words used by the compiler, so they cannot be used as the variable names. If the keywords are used as the

variable names, it means that we are assigning a different meaning to the keyword, which is not allowed. C language supports 32 keywords.

Following table represents the keywords in 'C'-

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

2.1.2.2 Identifiers

An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc. Example total, sum, average, _m _, sum_1, etc. Identifiers are the user-defined names consisting of 'C' standard character set. As the name says, identifiers are used to identify a particular element in a program. Each identifier must have a unique name.

Following rules must be followed for identifiers:

The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.

- ✓ It should not begin with any numerical digit.
- ✓ In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- ✓ Commas or blank spaces cannot be specified within an identifier.
- ✓ Keywords cannot be represented as an identifier.
- ✓ The length of the identifiers should not be more than 31 characters.
- ✓ Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

2.1.2.3 Strings in C

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is the number of characters that the string contains.

Now, we describe the strings in different ways:

```
char a[10] = "justice"; // The compiler allocates the 10 bytes to the 'a' array.  
char a[] = "justice"; // The compiler allocates the memory at the run time.  
char a[10] = {'j','u','s','t','i','c','e','\0'}; // String is represented in the form of characters.
```

2.1.2.4 Operators in C

Operators in C are special symbols used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands.

Depending on the number of operands, operators are classified as follows:

Unary Operator

A unary operator is an operator applied to the single operand. For example: increment operator (++) , decrement operator (--) , sizeof, (type)*.

Binary Operator

The binary operator is an operator applied between two operands. The following is the list of the binary operators:

- Arithmetic Operators
- Relational Operators

- Shift Operators
- Logical Operators
- Bitwise Operators
- Conditional Operators
- Assignment Operator
- Misc (miscellaneous) Operator

2.1.3 C Variable, Data types, Constants

2.1.3.1 Variable

What is a Variable?

A variable is an identifier which is used to store some value. Variables can change during the execution of a program and update the value stored inside it. While constants never change at the time of execution.

A single variable can be used at multiple locations in a program. A variable name must be meaningful. It should represent the purpose of the variable.

A variable must be declared first before it is used somewhere inside the program. A variable name is formed using characters, digits and an underscore.

Following are the rules that must be followed while creating a variable:

1. A variable name should consist of only characters, digits and an underscore.
2. A variable name should not begin with a number.
3. A variable name should not consist of whitespace.
4. A variable name should not consist of a keyword.
5. 'C' is a case sensitive language that means a variable named 'age' and 'AGE' are different.

Following are the examples of valid variable names in a 'C' program:

```
height or HEIGHT
_height
_height1
My_name
```

Following are the examples of invalid variable names in a 'C' program:

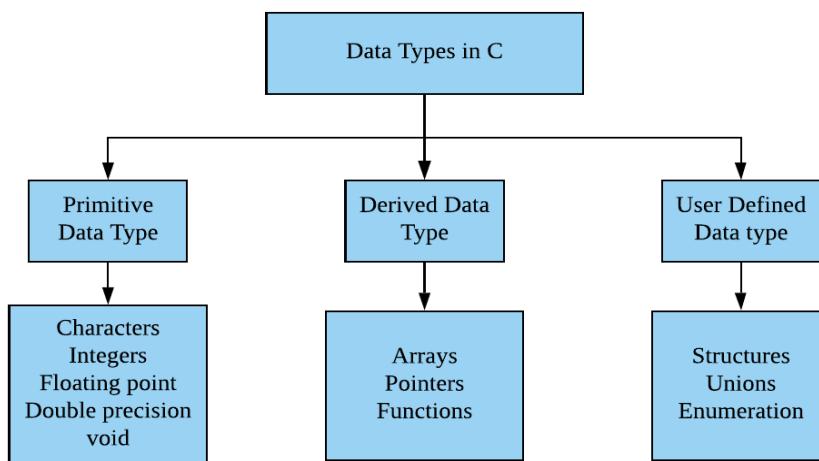
```
1height
Hei$ght
My name
```

2.1.3.2 Data types

'C' provides various data types to make it easy for a programmer to select a suitable data type as per the requirements of an application.

Following are the three data types:

- Primitive data types
- Derived data types
- User-defined data types



Array, functions, pointers are derived data types. Structure, Union and Enumeration are user defined data types. 'C' language provides more extended versions of the above-mentioned primary data types. Each data type differs from one another in size and range.

Following table displays the size and range of each data type.

Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d, %i
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%ld, %li
long long int	at least 8	%lld, %lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf

Note: In C, there is no Boolean data type.

2.1.3.2.1 Integer data type

Integer is nothing but a whole number. The range for an integer data type varies from machine to machine. The standard range for an integer data type is -32768 to 32767.

An integer typically is of 2 bytes which means it consumes a total of 16 bits in memory. A single integer value takes 2 bytes of memory. An integer data type is further divided into other data types such as **short int**, **int**, and **long int**.

Each data type differs in range even though it belongs to the integer data type family. The size may not change for each data type of integer family.

The short int is mostly used for storing small numbers, int is used for storing averagely sized integer values, and long int is used for storing large integer values.

Whenever we want to use an integer data type, we have place int before the identifier such as,

```
int age;
```

Here, age is a variable of an integer data type which can be used to store integer values.

2.1.3.2.2 Floating point data type

Like integers, in 'C' program we can also make use of floating-point data types. The 'float' keyword is used to represent the floating-point data type. It can hold a floating-point value which means a number is having a fraction and a decimal part. A floating-point value is a real number that contains a decimal point. Integer data type doesn't store the decimal part hence we can use floats to store decimal part of a value.

Generally, a float can hold up to 6 precision values. If the float is not sufficient, then we can make use of other data types that can hold large floating-point values. The data type double and long double are used to store real numbers with precision up to 14 and 80 bits respectively. While using a floating-point number a keyword float/double/long double must be placed before an identifier.

The valid examples are,

```
float division;
```

```
double BankBalance;
```

2.1.3.2.3 Character data type

Character data types are used to store a single character value enclosed in single quotes.

A character data type takes up-to 1 byte of memory space.

Example,

```
char letter;
```

2.1.3.2.4 Void data type

A void data type doesn't contain or return any value. It is mostly used for defining functions in 'C'.

Example,

```
void displayData()
```

Type declaration of a variable

```
int x = 10;  
float salary = 13.48;  
char letter = 'K';
```

2.1.3.3 Constants

Constants are the fixed values that never change during the execution of a program.

Following are the various types of constants:

2.1.3.3.1 Integer constants

An integer constant is nothing but a value consisting of digits or numbers. These values never change during the execution of a program. Integer constants can be octal, decimal and hexadecimal.

1. Decimal constant contains digits from 0-9 such as,

Example: - 111, 1234

Above are the valid decimal constants.

2. Octal constant contains digits from 0-7, and these types of constants are always preceded by 0.

Example: - 012, 065

Above are the valid decimal constants.

3. Hexadecimal constant contains a digit from 0-9 as well as characters from A-F. Hexadecimal constants are always preceded by 0X.

Example: - 0X2, 0Xbcd

Above are the valid hexadecimal constants.

The octal and hexadecimal integer constants are very rarely used in programming with 'C'.

2.1.3.3.2 Character constants

A character constant contains only a single character enclosed within a single quote (""). We can also represent character constant by providing ASCII value of it.

Example: - 'A', '9'

Above are the examples of valid character constants.

2.1.3.3.3 String constants

A string constant contains a sequence of characters enclosed within double quotes ("").

Example: - "Hello", "Programming"

These are the examples of valid string constants.

2.1.3.3.4 Real Constants

Like integer constants that always contains an integer value. 'C' also provides real constants that contain a decimal point or a fraction value. The real constants are also called as floating-point constants. The real constant contains a decimal point and a fractional value.

Example: - 202.15, 300.00

These are the valid real constants in 'C'.

A real constant can also be written as,

Mantissa e Exponent

For example, to declare a value that does not change like the classic circle constant PI, there are two ways to declare this constant

1. By using the **const** keyword in a variable declaration which will reserve a storage memory

```
const double PI = 3.14
```

2. By using the **#define** pre-processor directive which doesn't use memory for storage and without putting a semicolon character at the end of that statement

```
#define PI 3.14
```

Summary

A constant is a value that doesn't change throughout the execution of a program.

A variable is an identifier which is used to store a value.

There are four commonly used data types such as int, float, char and a void.

Each data type differs in size and range from one another.

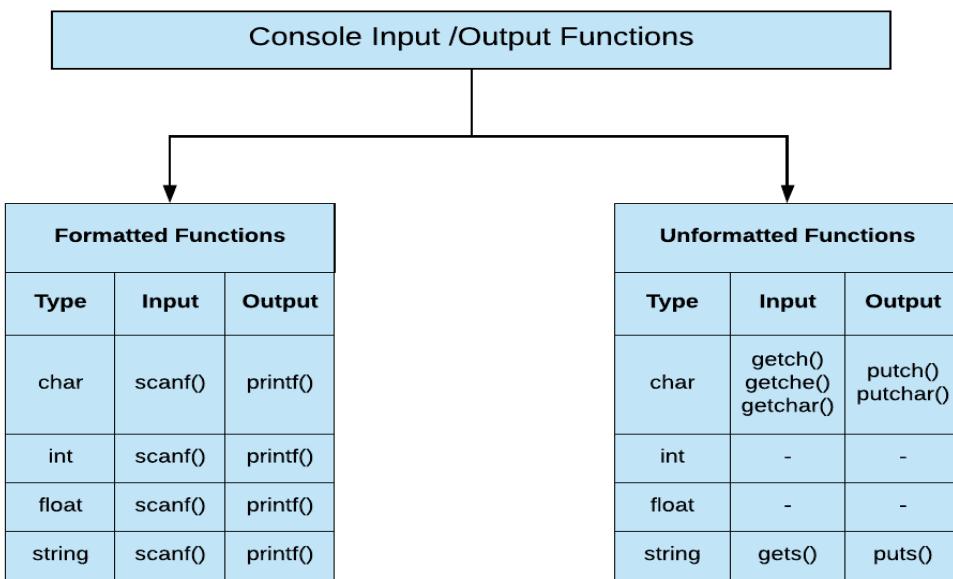
2.1.4 Console Input/output Operations: printf() and scanf()

C language provides us console input/output functions. As the name says, the console input/output functions allow us to -

- Read the input from the keyboard by the user accessing the console.
- Display the output to the user at the console.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

Note: These input and output values could be of *any primitive data type*.



There are two kinds of console input/output functions -

- Formatted input/output functions.
- Unformatted input/output functions.

a) Formatted input/output functions:

The functions which comes under this category are printf() and scanf(). They provide the flexibility to receive the input in some fixed format and to give the output in desired format.

The printf() function sends formatted output to the screen. For example,

Output Functions:

Example 1: C Output

```
#include <stdio.h>
int main()
{
    // Displays the string inside quotations
    printf("C Programming");
    return 0;
}
```

OUTPUT

C Programming

Example 2: Integer Output

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

```
{  
    int testInteger = 5;  
    printf("Number = %d", testInteger);  
    return 0;  
}
```

OUTPUT

Number = 5

We use %d format specifier to print int types. Here, the %d inside the quotations will be replaced by the value of test Integer.

Example 3: float and double Output

```
#include <stdio.h>  
int main()  
{  
    float number1 = 13.5;  
    double number2 = 12.4;  
    printf("number1 = %f\n", number1);  
    printf("number2 = %lf", number2);  
    return 0;  
}
```

OUTPUT

number1 = 13.500000

number2 = 12.400000

To print float, we use %f format specifier. Similarly, we use %lf to print double values.

Example 4: Print Characters

```
#include <stdio.h>  
int main ()  
{  
    char chr = 'a';  
    printf("character = %c", chr);  
    return 0;  
}
```

OUTPUT

character = a

To print char, we use %c format specifier.

Input Functions:

In C programming, scanf() is one of the commonly used function to take input from the user. The scanf() function reads formatted input from the standard input such as keyboards.

Example 5: Integer Input/output

```
#include <stdio.h>
int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d", testInteger);
    return 0;
}
```

OUTPUT

Enter an integer: 4

Number = 4

Here, we have used %d format specifier inside the scanf() function to take int input from the user. When the user enters an integer, it is stored in the **testInteger** variable.

Example 6: Float and Double Input/output

```
#include <stdio.h>
int main()
{
    float num1;
    double num2;
    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);
    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);
    return 0;
}
```

OUTPUT

Enter a number: 12.523

Enter another number: 10.2

num1 = 12.523000

num2 = 10.200000

We use %f and %lf format specifier for float and double respectively.

Example 7: C Character I/O

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c",&chr);
    printf("You entered %c.", chr);
    return 0;
}
```

OUTPUT

```
Enter a character: g
```

```
You entered g
```

When a character is entered by the user in the above program, the character itself is not stored.

Instead, an integer value (ASCII value) is stored.

And when we display that value using %c text format, the entered character is displayed. If we use %d to display the character, its ASCII value is printed.

Example 8: ASCII Value

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c", &chr);
    // When %c is used, a character is displayed
    printf("You entered %c.\n",chr);
    // When %d is used, ASCII value is displayed
    printf("ASCII value is %d.", chr);
    return 0;
}
```

OUTPUT

```
Enter a character: g
```

```
You entered g.
```

ASCII value is 103.

Example 9: I/O Multiple Values

Here's how you can take multiple inputs from the user and display them.

```
#include <stdio.h>

int main()
{
    int a;
    float b;

    printf("Enter integer and then a float: ");
    // Taking multiple inputs
    scanf("%d%f", &a, &b);
    printf("You entered %d and %f", a, b);
    return 0;
}
```

OUTPUT

```
Enter integer and then a float: -3
```

```
3.4
```

```
You entered -3 and 3.400000
```

Format Specifiers for I/O

As you can see from the above examples, we use

```
%d for int
%f for float
%lf for double
%c for char
```

Here's a list of commonly used C data types and their format specifiers.

Data Type	Format Specifier
int	%d
char	%c
float	%f
double	%lf
short int	%hd
unsigned int	%u

long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf

2.2 Operators and Expressions

2.2.1 Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then –

Operator	Description	Example
+	Adds two operands.	$A + B = 30$
-	Subtracts second operand from the first.	$A - B = -10$
*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$
%	Modulus Operator and remainder of after an integer division.	$B \% A = 0$
++	Increment operator increases the integer value by one.	$A++ = 11$
--	Decrement operator decreases the integer value by one.	$A-- = 9$

Example 10: Arithmetic Operators

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9, b = 4, c;
    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
```

```

c = a*b;
printf("a*b = %d \n",c);
c = a/b;
printf("a/b = %d \n",c);
c = a%b;
printf("Remainder when a divided by b = %d \n",c);
return 0;
}

```

OUTPUT

```

a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1

```

Example 11: Increment and Decrement Operators

```

// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;
    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);
    return 0;
}

```

OUTPUT

```

++a = 11
--b = 99
++c = 11.500000
--d = 99.500000

```

2.2.2 Relational Operators

The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then –

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Example 12: Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;
    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
```

```

printf("%d >= %d is %d \n", a, b, a >= b);
printf("%d >= %d is %d \n", a, c, a >= c);
printf("%d <= %d is %d \n", a, b, a <= b);
printf("%d <= %d is %d \n", a, c, a <= c);
return 0;
}

```

OUTPUT

```

5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5! = 5 is 0
5! = 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1

```

2.2.3 Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A&&B) is false.
 	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	! (A&&B) is true.

Example 13: Logical Operators

```
// Working of logical operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;
    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);
    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);
    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);
    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);
    result = !(a != b);
    printf("!(a != b) is %d \n", result);
    result = !(a == b);
    printf("!(a == b) is %d \n", result);
    return 0;
}
```

OUTPUT

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

2.2.4 Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. The truth tables for &, |, and ^ is as follows –

A	B	A & B	A B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume $A = 60$ and $B = 13$ in binary format, they will be as follows –

$A = 0011\ 1100$

$B = 0000\ 1101$

$A \& B = 0000\ 1100$

$A | B = 0011\ 1101$

$A ^ B = 0011\ 0001$

Operator	Description	Example
$\&$	Binary AND Operator copies a bit to the result if it exists in both operands.	$(A \& B) = 12$, i.e., 0000 1100
$ $	Binary OR Operator copies a bit if it exists in either operand.	$(A B) = 61$, i.e., 0011 1101
$^$	Binary XOR Operator copies the bit if it is set in one operand but not both.	$(A ^ B) = 49$, i.e., 0011 0001
\sim	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	$(\sim A) = \sim (60)$, i.e., 0111101
$<<$	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.	$A << 2 = 240$ i.e., 1111 0000
$>>$	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.	$A >> 2 = 15$ i.e., 0000 1111

2.2.5 Assignment Operator

The following table lists the assignment operators supported by the C language

Operator	Description	Example
$+$	Simple assignment operator. Assigns values from right side operands to left side operand.	$C = A + B$ will assign the value of $A + B$ to C .

<code>+=</code>	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	<code>C+=A</code> is equivalent to <code>C = C + A</code> .
<code>-=</code>	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	<code>C-= A</code> is equivalent to <code>C = C - A</code> .
<code>*=</code>	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	<code>C*=A</code> is equivalent to <code>C = C * A</code>
<code>/=</code>	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	<code>C/= A</code> is equivalent to <code>C = C / A</code>
<code>%=</code>	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	<code>C% = A</code> is equivalent to <code>C = C % A</code>
<code><<=</code>	Left shift AND assignment operator.	<code>C <<= 2</code> is same as <code>C = C << 2</code>
<code>>>=</code>	Right shift AND assignment operator.	<code>C >>= 2</code> is same as <code>C = C >> 2</code>
<code>&=</code>	Bitwise AND assignment operator.	<code>C &= 2</code> is same as <code>C = C & 2</code>
<code>^=</code>	Bitwise exclusive OR and assignment operator.	<code>C ^= 2</code> is same as <code>C = C ^ 2</code>
<code> =</code>	Bitwise inclusive OR and assignment operator.	<code>C = 2</code> is same as <code>C = C 2</code>

2.2.6 Misc Operator

Besides the operators discussed above, there are a few other important operators including `sizeof` and `? supported by the C Language.`

Operator	Description	Example
<code>sizeof()</code>	Returns the size of a variable.	<code>sizeof(a)</code> , where <code>a</code> is integer, will return 4.
<code>&</code>	Returns the address of a variable.	<code>&a</code> ; returns the actual address of the variable.
<code>*</code>	Pointer to a variable.	<code>*a</code> ; designate <code>a</code> variable as a pointer
<code>? :</code>	Conditional Expression.	If Condition is true? then value X: otherwise, value Y

2.2.6.1 Sizeof Operator in C

The sizeof operator is the most common operator in C. It is a compile-time unary operator and used to compute the size of its operand. It returns the size of a variable. It can be applied to any data type, float type, pointer type variables.

When sizeof() is used with the data types, it simply returns the amount of memory allocated to that data type. The output can be different on different machines like a 32-bit system can show different output while a 64-bit system can show different of same data types.

Here is an example in C language,

Example 14:

```
#include <stdio.h>

int main() {
    int a = 16;

    printf("Size of variable a : %d\n", sizeof(a));
    printf("Size of int data type : %d\n", sizeof(int));
    printf("Size of char data type : %d\n", sizeof(char));
    printf("Size of float data type : %d\n", sizeof(float));
    printf("Size of double data type : %d\n", sizeof(double));
    return 0;
}
```

OUTPUT

```
Size of variable a: 4
Size of int data type: 4
Size of char data type: 1
Size of float data type: 4
Size of double data type: 8
```

When the sizeof() is used with an expression, it returns the size of the expression. Here is an example.

Example 15:

```
#include <stdio.h>

int main()
{
    char a = 'S';
    double b = 4.65;
```

```

printf("Size of variable a : %d\n",sizeof(a));
printf("Size of an expression : %d\n",sizeof(a+b));
int s = (int)(a+b);
printf("Size of explicitly converted expression : %d\n",sizeof(s));
return 0;
}

```

OUTPUT

```

Size of variable a: 1
Size of an expression: 8
Size of explicitly converted expression: 4

```

2.2.6.2 Conditional Expression (Ternary operator)

If any operator is used on three operands or variable is known as Ternary Operator. It can be represented with `? :`. It is also called as conditional operator/Expression.

The ternary operator is an operator that takes three arguments. The first argument is a comparison argument, the second is the result upon a true comparison, and the third is the result upon a false comparison.

Syntax for Ternary operator:

expression-1 ? expression-2 : expression-3

Assume a ternary operator is given as `a? b: c`, this means if the condition `a` is true `b` will be executed else `c` will be executed.

Example 16: Write a program to determine to eligibility of a person to vote.

```

#include<stdio.h>
int main()
{
int age;
printf("Enter the age in numbers: ");
scanf("%d",&age);
age>=18?(printf("Eligible for Voting as Age is %d \n",age))
:(printf("Not eligible for Voting as Age is %d \n",age));
return 0;
}

```

OUTPUT

```
Enter the age in numbers: 18
Eligible for Voting as Age is 18
Enter the age in numbers: 17
Not eligible for Voting as Age is 17
```

Example 17: Find the largest number among the 3 numbers given.

```
#include<stdio.h>
int main()
{
    int a, b, c, large;
    printf("Enter any three number: ");
    scanf("%d%d%d", &a, &b, &c);
    large= a>b?(a>c?a:c):(b>c?b:c);
    printf("Largest Number is: %d \n",large);
    return 0;
}
```

OUTPUT

```
Enter any three number: 10 20 30
Largest Number is: 30
```

2.2.7 Precedence of operators

The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

Let us consider an example:

```
int x = 5 - 17* 6;
```

In C, the precedence of * is higher than - and =. Hence, $17 * 6$ is evaluated first. Then the expression involving - is evaluated as the precedence of - is higher than that of =.

Here's a table of operator's precedence from higher to lower. This is determined by **associativity** of operator.

Associativity of Operators

The Associativity of operators determines the direction in which an expression is evaluated.

For Example

```
b = a;
```

Here, the value of a is assigned to b, and not the other way around. It's because the associativity of the = operator is from right to left.

Also, if two operators of the same precedence (priority) are present, associativity determines the direction in which they execute.

Let us consider an example:

```
1 == 2 != 3
```

Here, operators == and != have the same precedence. And, their associativity is from left to right. Hence, 1 == 2 is executed first.

The expression above is equivalent to:

```
(1 == 2) != 3
```

Note: If a statement has multiple operators, you can use parentheses () to make the code more readable.

Operators Precedence & Associativity Table

Operator	Meaning of operator	Associativity
()	Functional call	Left to right
[]	Array element reference	
->	Indirect member selection	
.	Direct member selection	
!	Logical negation	Right to left
~	Bitwise (1 's) complement	Right to left
+	Unary plus	Right to left
-	Unary minus	Right to left
++	Increment	Right to left
--	Decrement	Right to left
&	Dereference (Address)	Right to left
*	Pointer reference	Right to left
sizeof	Returns the size of an object	Right to left
(type)	Typecast (conversion)	Right to left
*	Multiply	Left to right
/	Divide	
%	Remainder	
+	Binary plus (Addition)	Left to right
-	Binary minus(subtraction)	

<<	Left shift	Left to right
>>	Right shift	
<	Less than	
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal to	Left to right
!=	Not equal to	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
	Bitwise OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
?:	Conditional Operator	Right to left
=	Simple assignment	Right to left
*=	Assign product	Right to left
/=	Assign quotient	Right to left
%=	Assign remainder	Right to left
+=	Assign sum	Right to left
-=	Assign difference	Right to left
&=	Assign bitwise AND	Right to left
^=	Assign bitwise XOR	Right to left
=	Assign bitwise OR	Right to left
<<=	Assign left shift	Right to left
>>=	Assign right shift	Right to left
,	Separator of expressions	Left to right

2.3 Control Flow Statements

C provides two styles of flow control:

1. Branching
2. Looping

Branching is deciding what actions to take and **Looping** is deciding how many times to take a certain action.

2.3.1 Branching

The branching in C is used to perform the operations based on some specific condition. The operations specified in branching block are executed if and only if the given condition is true. There are four branching statements in C language.

- a) If statement
- b) If-else statement
- c) If else-if ladder
- d) Nested if

2.3.1.1 If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions.

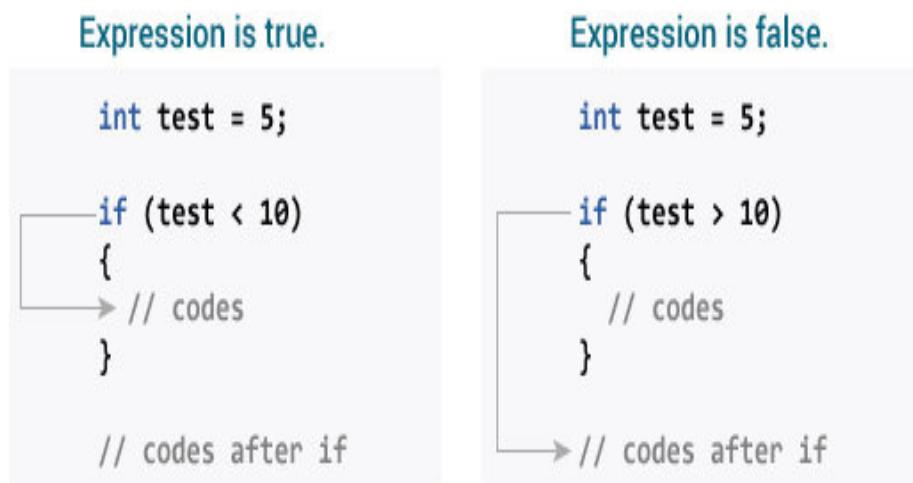
The syntax of the if statement is given below.

```
if(expression)
{
//code to be executed
}
```

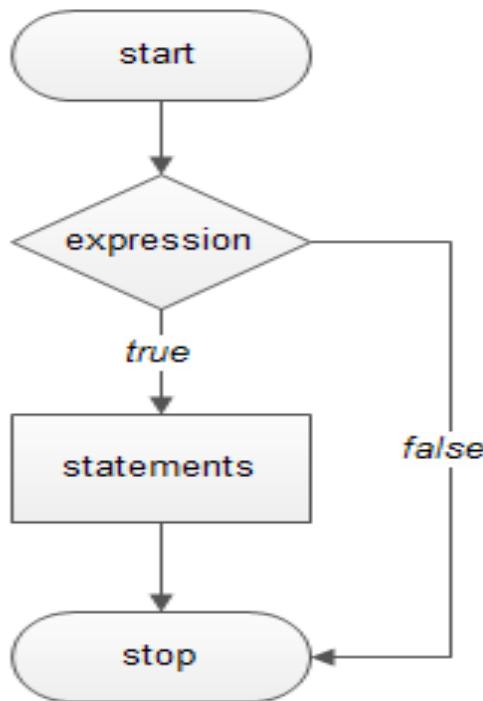
How if statement works?

The if statement evaluates the test expression inside the parenthesis () .

- ✓ If the test expression is evaluated to true, statements inside the body of if are executed.
- ✓ If the test expression is evaluated to false, statements inside the body of if are not executed.



Flowchart of if statement in C



Example 18: if statement

```
// Program to display a number if it is negative
#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    // true if number is less than 0
    if (number < 0) {
        printf("You entered %d.\n", number);
    }
    printf("The if statement is easy.");
    return 0;
}
```

OUTPUT 1

```
Enter an integer: -2
You entered -2.
The if statement is easy.
```

When the user enters -2, the test expression number<0 is evaluated to true. Hence, the entered -2 is displayed on the screen.

OUTPUT 2

```
Enter an integer: 5
```

```
The if statement is easy.
```

When the user enters 5, the test expression number<0 is evaluated to false and the statement inside the body of **if** is not executed.

2.3.1.2 If-else statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simultaneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition.

The syntax of the if-else statement is given below.

```
if(expression)
{
    //code to be executed if condition is true
}
else
{
    //code to be executed if condition is false
}
```

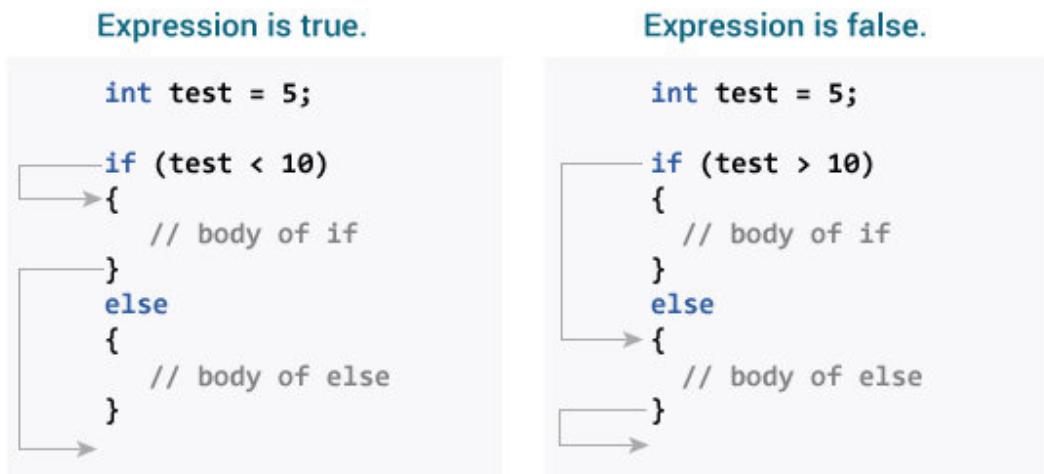
How if...else statement works?

If the test expression is evaluated to true,

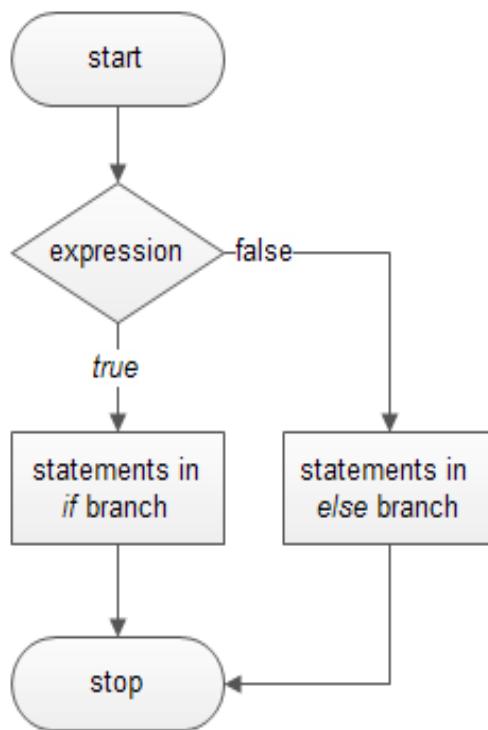
- ✓ statements inside the body of **if** are executed.
- ✓ statements inside the body of **else** are skipped from execution.

If the test expression is evaluated to false,

- ✓ statements inside the body of **else** are executed
- ✓ statements inside the body of **if** are skipped from execution.



Flowchart of if-else statement in C



Example 19: if...else statement

```
// Check whether an integer is odd or even

#include <stdio.h>

int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    // True if the remainder is 0
```

```

if (number%2 == 0) {
    printf("%d is an even integer.",number);
}
else {
    printf("%d is an odd integer.",number);
}
return 0;
}

```

OUTPUT

Enter an integer: 7

7 is an odd integer.

When the user enters 7, the test expression `number%2==0` is evaluated to false. Hence, the statement inside the body of `else` is executed.

2.3.1.3 If else-if ladder

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There is multiple else-if blocks possible.

The syntax of the if else-if ladder is given below.

```

if(condition1)
{
    //code to be executed if condition1 is true
}
else if(condition2)
{
    //code to be executed if condition2 is true
}
else if(condition3)
{
    //code to be executed if condition3 is true
}
...

```

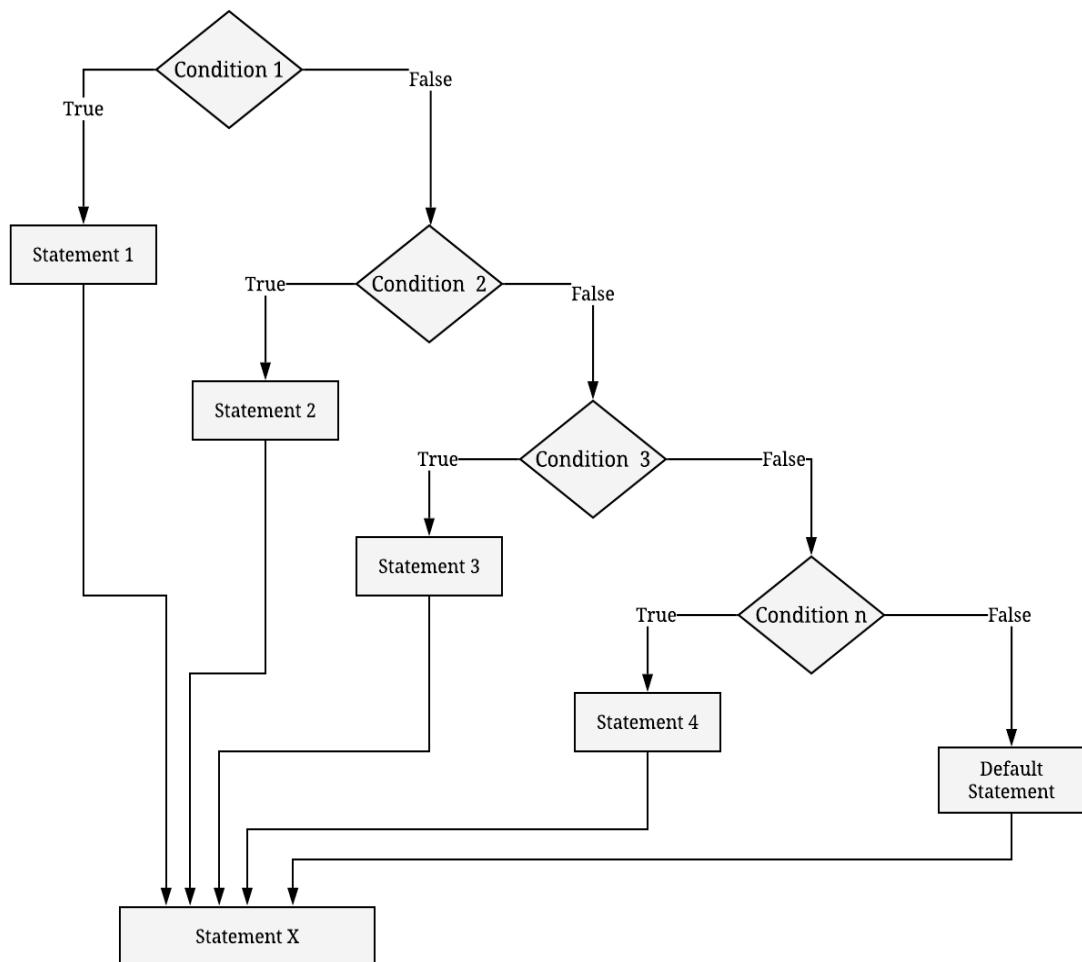


```

else
{
//code to be executed if all the conditions are false
}

```

Flowchart of if else-if ladder statement in C



Example 20: if...else Ladder

```

// Program to relate two integers using =, > or < symbol
#include <stdio.h>

int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
    //checks if the two integers are equal.
}

```

```

if(number1 == number2) {
    printf("Result: %d = %d",number1,number2);
}

//checks if number1 is greater than number2.

else if (number1 > number2) {
    printf("Result: %d > %d", number1, number2);
}

//checks if both test expressions are false

else {
    printf ("Result: %d < %d", number1, number2);
}

return 0;
}

```

OUTPUT

Enter two integers: 12

23

Result: 12 < 23

2.3.1.4 Nested if

It is possible to include an if...else statement inside the body of another if...else statement.

The syntax of the Nested if is given below.

```

if (condition 1)
{
    if (condition 2)
    {
        Statement A;
    }
    else
    {
        Statement B;
    }
}
else
{
    if (condition 3)
    {
        Statement C;
    }
    else
    {
}

```

```
Statement D;
}
}
```

How if...else nested statement works?

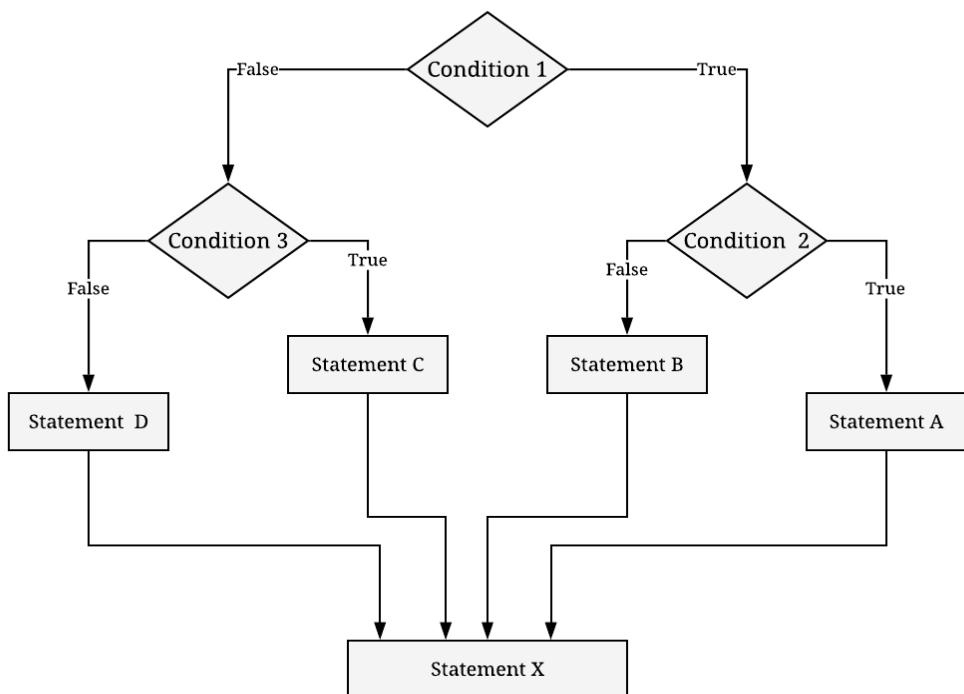
If the test condition 1 is evaluated to true,

- Enter a new loop and check for condition 2
 - ✓ If condition satisfied print statement A
 - ✓ If condition not-satisfied print statement B

If the test condition 1 is evaluated to false,

- Enter a new loop and check for condition 2
 - ✓ If condition satisfied print statement C
 - ✓ If condition not-satisfied print statement D

Flowchart of if-else nested in C



Example 21: Nested if...else

```
#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
```

```

if(number1 >= number2) {
    if(number1 == number2) {
        printf("Result: %d = %d",number1,number2);
    }
    else {
        printf("Result: %d > %d", number1, number2);
    }
}
else {
    printf("Result: %d < %d",number1, number2);
}
return 0;
}

```

OUTPUT

Enter two integers: 12

23

Result: 12 < 23

2.3.2 C Switch Statement

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable. Here, we can define various statements in the multiple cases for the different values of a single variable.

Syntax of switch...case

```

switch (expression)
{
    case constant1:
        // statements
        break;
    case constant2:
        // statements
        break;
    ....
    ....
}

```



```
default:  
    // default statements  
}
```

How does the switch statement work?

The expression is evaluated once and compared with the values of each case label.

- If there is a match, the corresponding statements after the matching label are executed.
For example, if the value of the expression is equal to constant2, statements after case constant2: are executed until break is encountered.
- If there is no match, the default statements are executed.
- If we do not use break, all statements after the matching label are executed.
- By the way, the default clause inside the switch statement is optional.

Rules for switch statement in C language

- 1) The switch expression must be of an integer or character type.
- 2) The case value must be an integer or character constant.
- 3) The case value can be used only inside the switch statement.
- 4) The break statement in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as fall through the state of C switch statement.

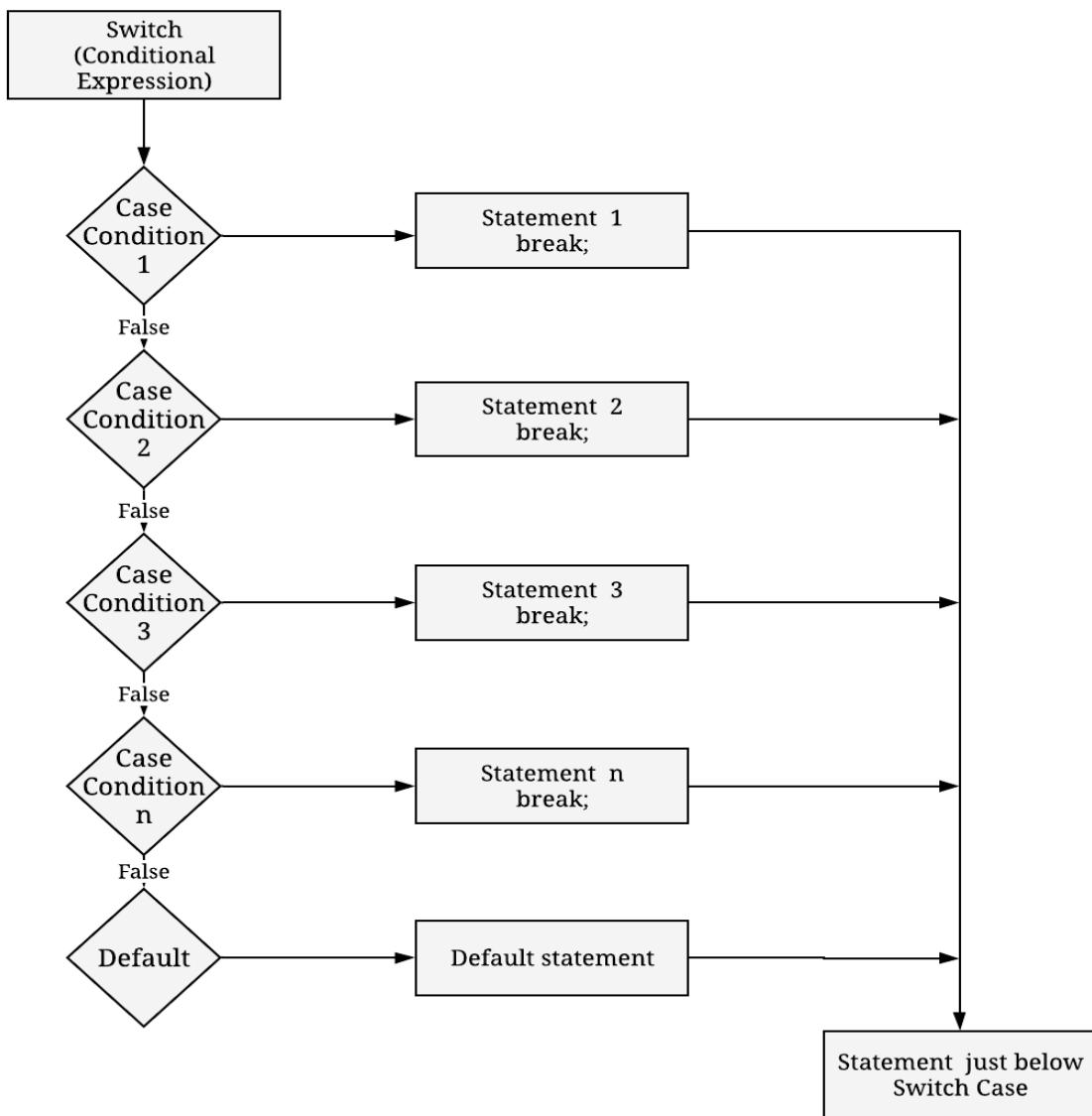
Let's try to understand it by the examples. We are assuming that there are following variables.

```
int x,y,z;  
char a,b;  
float f;
```

Example 22: Simple Calculator

```
// Program to create a simple calculator  
#include <stdio.h>  
int main() {  
    char operator;  
    double n1, n2;  
    printf("Enter an operator (+, -, *, /): ");  
    scanf("%c", &operator);
```

Flowchart of Switch case in C



```

printf("Enter two operands: ");
scanf("%lf %lf",&n1, &n2);
switch(operator)
{
    case '+':
        printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
        break;
    case '-':
        printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
        break;
}
  
```

```

case '*':
    printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
    break;
case '/':
    printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
    break;
// Operator doesn't match any case constant +, -, *, /
default:
    printf("Error! operator is not correct");
}
return 0;
}

```

OUTPUT

Enter an operator (+, -, *,): -

Enter two operands: 32.5

12.4

32.5 - 12.4 = 20.1

The - operator entered by the user is stored in the operator variable. And, two operands 32.5 and 12.4 are stored in variables n1 and n2 respectively.

2.3.3 Unconditional Branching using goto statement

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complicated.

Syntax for goto:

```

label:
//some part of the code;
goto label;

```

Example 23: goto example

```

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

```

```

int num,i=1;
printf("Enter the number whose table you want to print?");
scanf("%d",&num);
table:
printf("%d x %d = %d\n",num,i,num*i);
i++;
if(i<=10)
    goto table;
}

```

OUTPUT

Enter the number whose table you want to print?10

```

10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100

```

When should we use goto?

The only condition in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time.

Consider the following example.

Example 24:

```

#include <stdio.h>
int main()
{
    int i, j, k;
    for(i=0;i<10;i++)
    {
        for(j=0;j<5;j++)
        {
            for(k=0;k<3;k++)
            {

```

```

printf("%d %d %d\n",i,j,k);
if(j == 3)
{
    goto out;
}
}
}
out:
printf("came out of the loop");
}

```

OUTPUT

```

0 0 0
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
0 3 0
came out of the loop

```

2.3.4 C Loops

The looping can be defined as repeating the same process multiple times until a specific condition satisfies.

Why use loops in C language?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantage of loops in C

- 1) It provides code reusability.
- 2) Using loops, we do not need to write the same code again and again.
- 3) Using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C Loops

There are three types of loops in C language that is given below:

- do while
- while
- for

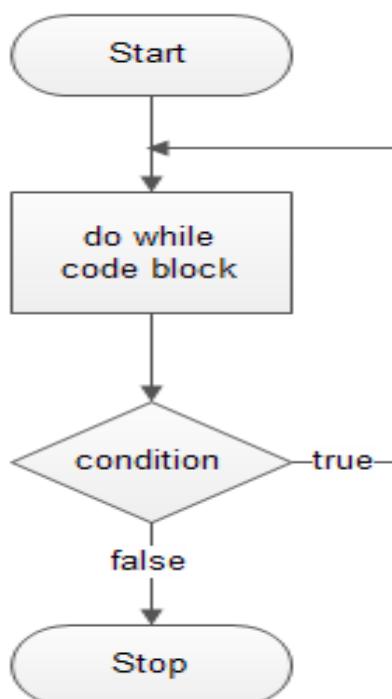
2.3.4.1 do-while loop in C

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

The syntax of do-while loop is given below:

```
do
{
//code to be executed
}
while(condition);
```

Flowchart of do-while loop



Example 25: do-while loop.

```
#include<stdio.h>

int main()
{
int i=1;
do
{
    printf("%d \n",i);
    i++;
}
while(i<=10);
return 0;
}
```

OUTPUT

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

2.3.4.2 While loop in C

The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

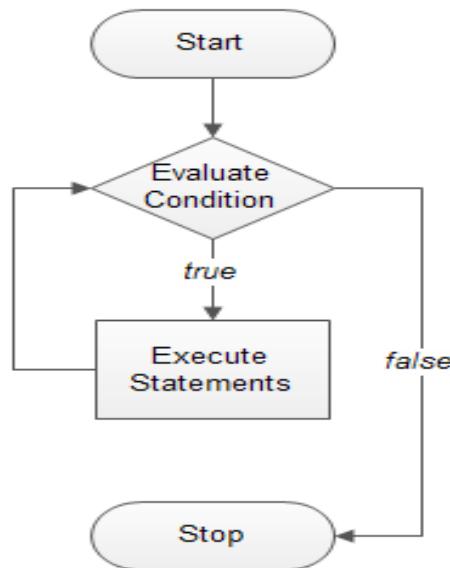
The syntax of while loop in c language is given below:

```
while (condition)
{
//code to be executed
}
```

How while loop works?

- The while loop evaluates the test expression inside the parenthesis ().
- If the test expression is true, statements inside the body of while loop are executed.
Then, the test expression is evaluated again.
- The process goes on until the test expression is evaluated to false.
- If the test expression is false, the loop terminates (ends).

Flowchart of while loop in C



Example 26: To print numbers from 0 to 10 using while loop.

```
#include<stdio.h>

int main()
{
    int i=1;
    while(i<=10)
    {
        printf("%d \n",i);
        i++;
    }
    return 0;
}
```

OUTPUT

```
1
2
3
```

```
4
5
6
7
8
9
10
```

Example 27: Print table for the given number using while loop in C

```
#include<stdio.h>

int main()
{
int i=1,number=0,b=9;
printf("Enter a number: ");
scanf("%d",&number);
while(i<=10)
{
printf("%d \n", (number*i));
i++;
}
return 0;
}
```

OUTPUT

```
Enter a number: 50
50
100
150
200
250
300
350
400
450
500
```

2.3.4.2.1 Infinitive while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```
while(1)
```



```
{  
//statement  
}
```

2.3.4.3 for loop in C

The for loop in C language is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

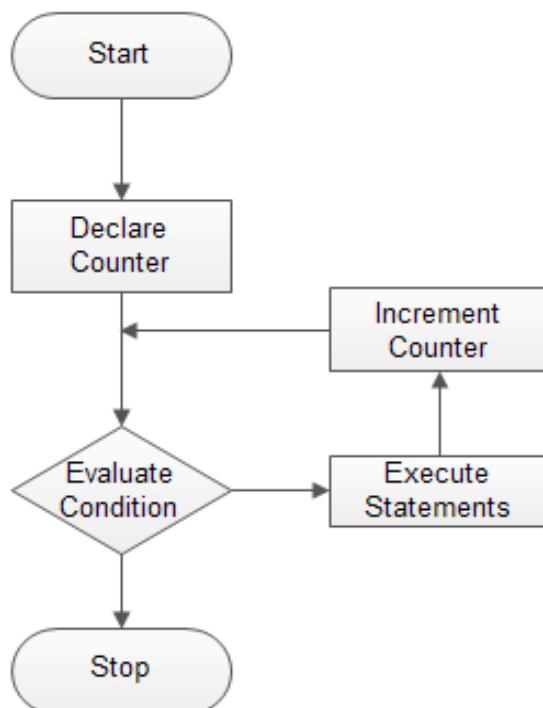
The syntax of for loop is given below:

```
for (initialization_expression;loop_condition;increment_expression)  
{  
// execute statements  
}
```

How for loop works?

1. The initialization_expression executes when the loop first starts. It is typically used to initialize a loop counter variable.
2. The loop_condition expression is evaluated at the beginning of each iteration. The execution of the loop continues until the loop_condition evaluates to false.
3. The increment_expression is evaluated at the end of each iteration. It is used to increase or decrease the loop counter variable.

Flowchart of for loop



Examples 28: for loop

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

OUTPUT

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

2.3.4.3.1 Loop expressions for loop

Expression 1:

1. The expression represents the initialization of the loop variable.
2. We can initialize more than one variable in Expression 1.
3. Expression 1 is optional.
4. In C, we cannot declare the variables in Expression 1. However, it can be an exception in some compilers.

Example 29: Take three numbers and add till the first number is less than 2.

```
#include <stdio.h>
int main()
{
int a,b,c;
for(a=0,b=12,c=23;a<2;a++)
```

```

{
    printf("%d ",a+b+c);
}
return 0;
}

```

OUTPUT

35 36

Example 30: Print all values from 1 to 4.

```

#include <stdio.h>
int main()
{
    int i=1;
    for(;i<5;i++)
    {
        printf("%d ",i);
    }
    return 0;
}

```

OUTPUT

1 2 3 4

Expression 2:

1. Expression 2 is a conditional expression. It checks for a specific condition to be satisfied. If it is not, the loop is terminated.
2. Expression 2 can have more than one condition. However, the loop will iterate until the last condition becomes false. Other conditions will be treated as statements.
3. Expression 2 is optional.
4. Expression 2 can perform the task of expression 1 and expression 3. That is, we can initialize the variable as well as update the loop variable in expression 2 itself.
5. We can pass zero or non-zero value in expression 2. However, in C, any non-zero value is true, and zero is false by default.

Example 31:

```

#include <stdio.h>
int main()

```



```
{
    int i;
    for(i=0;i<=4;i++)
    {
        printf("%d ",i);
    }
    return 0;
}
```

OUTPUT

```
0 1 2 3 4
```

Example 32:

```
#include <stdio.h>
int main()
{
    int i,j,k;
    for(i=0,j=0,k=0;i<4,k<8,j<10;i++)
    {
        printf("%d %d %d\n",i,j,k);
        j+=2;
        k+=3;
    }
    return 0;
}
```

OUTPUT

```
0 0 0
1 2 3
2 4 6
3 6 9
4 8 12
```

Expression 3:

1. Expression 3 is used to update the loop variable.
2. We can update more than one variable at the same time.
3. Expression 3 is optional.

Example 33:

```
#include<stdio.h>
int main ()
{
    int i=0,j=2;
    for(i = 0;i<5;i++,j=j+2)
    {
        printf("%d %d\n",i,j);
    }
    return 0;
}
```

Output

```
0 2
1 4
2 6
3 8
4 10
```

Example 34:

```
#include<stdio.h>
int main ()
{
    int i=0,j=2;
    for(i = 0;i<5;i++,j<<=2)
    {
        printf("%d %d\n",i,j);
    }
    return 0;
}
```

OUTPUT

```
0 2
1 8
2 32
3 128
4 512
```

2.3.4.3.2 Infinitive for loop in C

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

Example for infinite loop

```
#include<stdio.h>
void main ()
{
    for(;;)
    {
        printf("welcome to SBCE");
    }
}
```

If you run this program, you will see above statement running infinite times.

2.3.5 Break and Continue statements

2.3.5.1 C break

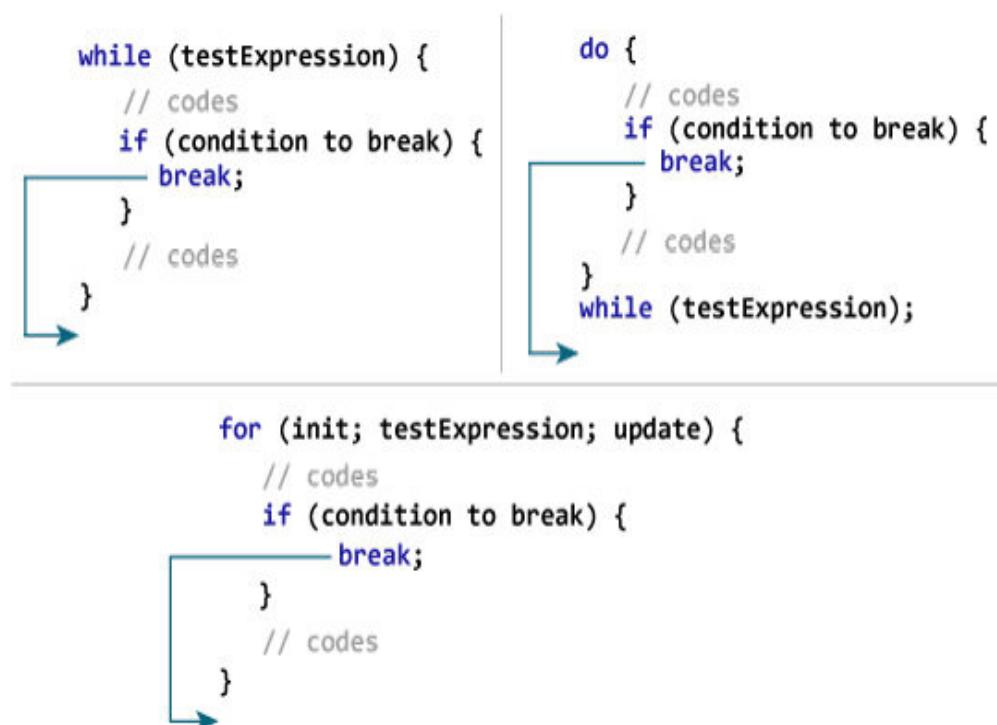
The break statement ends the loop immediately when it is encountered.

Its syntax is:

```
break;
```

The break statement is almost always used with if...else statement inside the loop.

How break statement works?



Example 35: break statement

```
// Program to calculate the sum of numbers (10 numbers max)
// If a negative number is entered, then loop terminates
#include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;
    for (i = 1; i <= 10; ++i) {
        printf("Enter n%d: ", i);
        scanf("%lf", &number);
        // if the user enters a negative number, break the loop
        if (number < 0.0) {
            break;
        }
        sum += number; // sum = sum + number;
    }
    printf("Sum = %.2lf", sum);
    return 0;
}
```

OUTPUT

```
Enter n1: 2.4
Enter n2: 4.5
Enter n3: 3.4
Enter n4: -3
Sum = 10.30
```

2.3.5.2 C Continue

The continue statement skips the current iteration of the loop and continues with the next iteration.

Its syntax is:

```
continue;
```

The continue statement is almost always used with the if...else statement.

How continue statement works?

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

Example 36: continue statement

```
// Program to calculate the sum of numbers (10 numbers max)  
// If a negative number is entered, it's not added to the result  
#include <stdio.h>  
  
int main() {  
    int i;  
    double number, sum = 0.0;  
    for (i = 1; i <= 10; i++)  
    {  
        printf("Enter a n%od: ", i);  
        scanf("%lf", &number);  
        if (number < 0.0) {  
            continue;  
        }  
        sum += number; // sum = sum + number;  
    }  
    printf("Sum = %.2lf", sum);  
    return 0;  
}
```

OUTPUT

```
Enter n1: 1.1
```

```
Enter n2: 2.2
Enter n3: 5.5
Enter n4: 4.4
Enter n5: -3.4
Enter n6: -45.5
Enter n7: 34.5
Enter n8: -4.2
Enter n9: -1000
Enter n10: 12
Sum = 59.70
```

2.3.6 Simple C programs

1) C program to find largest among two numbers using ternary operator.

```
#include <stdio.h>
int main()
{
    int m = 10,n = 20;
    (m > n) ? printf("m is greater than n that is %d > %d",m, n)
               : printf("n is greater than m that is %d > %d",n, m);

    return 0;
}
```

OUTPUT

```
n is greater than m that is 20 > 10
```

2) C program to demonstrate how a conditional operator is used to assign the value to a variable.

```
#include <stdio.h>
int main()
{
    int a=5,b; // variable declaration
    b=((a==5)?(3):(2)); // conditional operator
    printf("The value of 'b' variable is : %d",b);
    return 0;
}
```

```
}
```

OUTPUT

```
The value of 'b' variable is : 3
```

3) C program to swap two numbers.

```
#include<stdio.h>

int main()
{
    int firsnum, secondnum, temp;
    printf("Enter first number: ");
    scanf("%d",&firsnum);
    printf("Enter second number: ");
    scanf("%d",&secondnum);
    printf("Before swapping \n");
    printf("First number: %d \n", firsnum);
    printf("Second number: %d \n", secondnum);
    temp = firsnum;
    firsnum = secondnum;
    secondnum = temp;
    printf("After swapping \n");
    printf("First number: %d \n", firsnum);
    printf("Second number: %d \n", secondnum);
    return 0;
}
```

OUTPUT

```
Enter first number: 23
Enter second number: 34
Before swapping
First number: 23
Second number: 34
After swapping
First number: 34
Second number: 23
```

4) C program to determine whether a number lies between the ranges 1 to 10, or 10 to 20.

```
#include <stdio.h>
int main()
{
int n;
printf("Enter a digit between 1 to 10: ");
scanf("%d",&n);
if((n>0) && (n<=10))
{
printf(" Given number is in between 0 and 10");
}
else if((n>10) && (n<=20))
{
printf("Given number is in between 10 and 20");
}
else
{
printf("Please enter a number in the given range");
}
return 0;
}
```

OUTPUT

```
Enter a digit between 1 to 10: 5
Given number is in between 0 and 10
```

5) Program to Check whether a given alphabet is Vowel or consonant

```
#include <stdio.h>
int main() {
char c;
int lowercase_vowel, uppercase_vowel;
printf("Enter an alphabet: ");
scanf("%c", &c);
lowercase_vowel = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
```

```

uppercase_vowel = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');

if (lowercase_vowel || uppercase_vowel)
    printf("%c is a vowel.", c);
else
    printf("%c is a consonant.", c);
return 0;
}

```

OUTPUT

```

Enter an alphabet: I
I is a vowel
Enter an alphabet: a
a is a vowel

```

6) C program to reverse an Integer

```

#include <stdio.h>

int main() {
    int n, rev = 0, remainder;
    printf("Enter an integer: ");
    scanf("%d", &n);
    while (n != 0) {
        remainder = n % 10;
        rev = rev * 10 + remainder;
        n /= 10;
    }
    printf("Reversed number = %d", rev);
    return 0;
}

```

OUTPUT

```

Enter an integer: 234
Reversed number = 432

```

7) Check whether a given number is Strong or not

```

#include <stdio.h>

int main()

```

```

{
    int num,r,sum=0;
    printf("Enter a number: ");
    scanf("%d",&num);
    int k=num;
    while(k!=0)
    {
        r=k%10;
        int fact=1;
        for(int i=1;i<=r;i++)
        {
            fact=fact*i;
        }
        k=k/10;
        sum=sum+fact;
    }
    if(sum==num)
    {
        printf("\nNumber is a strong");
    }
    else
    {
        printf("\nNumber is not a strong");
    }
    return 0;
}

```

OUTPUT

```

Enter a number: 145
Number is a strong
Enter a number: 167
Number is not a strong

```

8) Check Strong numbers between two input numbers

```

#include <stdio.h>
int main()
{
    int num1,num2,r,sum=0;
    printf("Enter First number: ");
    scanf("%d",&num1);
    printf("Enter Last number: ");
    scanf("%d",&num2);
    for(int j= num1;j<=num2;j++)
    {
        int k=j;
        while(k!=0)
        {
            r=k%10;
            int fact=1;
            for(int i=1;i<=r;i++)
            {
                fact=fact*i;
            }
            k=k/10;
            sum=sum+fact;
        }
        if(sum==j)
        {
            printf("%d ",j);
        }
        sum=0;
    }
    return 0;
}

```

OUTPUT

```

Enter First number: 1
Enter Last number: 1000
1 2 145

```

THIS PAGE IS INTENTIONALLY LEFT BLANK

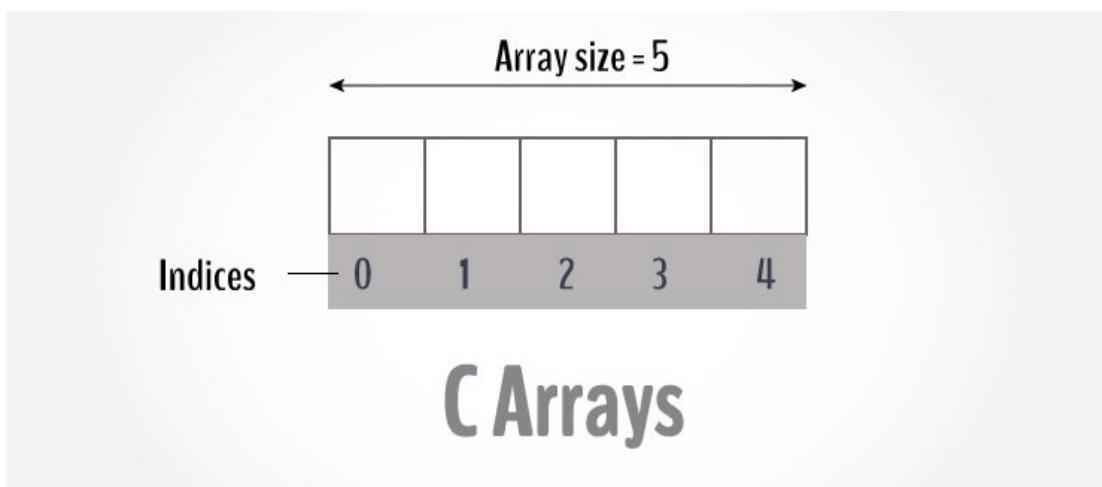
MODULE-III

Arrays and strings	
3.1	Arrays Declaration and Initialization, 1-Dimensional Array, 2-Dimensional Array
3.2	String processing: In built String handling functions (<i>strlen, strcpy, strcat and strcmp, puts, gets</i>)
3.3	Linear search program, bubble sort program, <i>simple programs covering arrays and strings</i>

3.1 Arrays Declaration and Initialization

What is an array in C?

- 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.
- Arrays also has the capability to store the collection of derived data types, such as pointers, structure, etc.
- The array is the simplest data structure where each data element can be randomly accessed by using its index number.



Properties of Array

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.

- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array

- 1) **Code Optimization:** Less code to access the data.
- 2) **Ease of traversing:** Using for loop we can retrieve the elements of an array easily.
- 3) **Ease of sorting:** To sort the elements of the array, we need a few lines of code only.
- 4) **Random Access:** We can access any element randomly using the array.

Disadvantage of C Array

Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit.

3.1.1 Declaration of Array in C

We can declare an array in the C language in the following way.

```
data_type array_name [array_size];
```

Now, let us see the example to declare the array.

```
int marks[5];
```

Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

3.1.2 Initialization of C Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index.

Consider the following example.

1. marks[0]=80; //initialization of array
2. marks[1]=60;
3. marks[2]=70;
4. marks[3]=85;
5. marks[4]=75;

80	60	70	85	75
----	----	----	----	----

marks[0] marks[1] marks[2] marks[3] marks[4]

Example 1: C array

```
#include<stdio.h>

int main(){
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
//traversal of array
for(i=0;i<5;i++)
{
printf("%d \n",marks[i]);
}//end of for loop
return 0;
}
```

OUTPUT

```
80
60
70
85
75
```

3.1.3 C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

```
int marks[5]={20,30,40,50,60};
```

In such case, there is **no requirement to define the size**. So, it may also be written as the following code.

```
int marks[]={20,30,40,50,60};
```

Example 2: The C program to declare and initialize the array in C.

```
#include<stdio.h>

int main(){
int i=0;
```



```

int marks[5]={20,30,40,50,60}//declaration and initialization of array
//traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
}
return 0;
}

```

OUTPUT

```

20
30
40
50
60

```

Example 3: Array Input / Output

```

// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>
int main() {
    int values[5];
    printf("Enter 5 integers: ");
    // Taking input and storing it in an array
    for(int i = 0; i < 5; ++i) {
        scanf("%d", &values[i]);
    }
    printf("Displaying integers: ");
    // Printing elements of an array
    for(int i = 0; i < 5; ++i) {
        printf("%d\n", values[i]);
    }
    return 0;
}

```

OUTPUT

```

Enter 5 integers: 1
-3

```

```
34
0
3
Displaying integers: 1
-3
34
0
3
```

3.1.4 Two-Dimensional Array in C

In C programming, we can create an array of arrays. These arrays are known as multidimensional arrays. A 2D array is organized as matrices which can be represented as the collection of rows and columns.

3.1.4 .1 Declaration of two-dimensional Array in C

The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

For example,

```
float x[3][4];
```

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. The array can be represented as a table with 3 rows and 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

3.1.4.2 Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to

define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

```
// Different ways to initialize two-dimensional array  
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};  
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};  
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Example 4: Sum of two matrices

```
// Different ways to initialize two-dimensional array  
// C program to find the sum of two matrices of order 2*2  
#include <stdio.h>  
int main()  
{  
    float a[2][2], b[2][2], result[2][2];  
    // Taking input using nested for loop  
    printf("Enter elements of 1st matrix\n");  
    for (int i = 0; i < 2; ++i)  
        for (int j = 0; j < 2; ++j)  
    {  
        printf("Enter a%d%d: ", i + 1, j + 1);  
        scanf("%f", &a[i][j]);  
    }  
    // Taking input using nested for loop  
    printf("Enter elements of 2nd matrix\n");  
    for (int i = 0; i < 2; ++i)  
        for (int j = 0; j < 2; ++j)  
    {  
        printf("Enter b%d%d: ", i + 1, j + 1);  
        scanf("%f", &b[i][j]);  
    }  
    // adding corresponding elements of two arrays  
    for (int i = 0; i < 2; ++i)  
        for (int j = 0; j < 2; ++j)  
    {  
        result[i][j] = a[i][j] + b[i][j];  
    }
```

```

    }

// Displaying the sum

printf("\nSum Of Matrix: \n");
for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
        printf("%.1f\t", result[i][j]);

        if (j == 1)
            printf("\n");
    }
return 0;
}

```

OUTPUT

```

Enter elements of 1st matrix
Enter a11: 1
Enter a12: 2
Enter a21: 3
Enter a22: 4
Enter elements of 2nd matrix
Enter b11: 4
Enter b12: 3
Enter b21: 2
Enter b22: 1
Sum Of Matrix:
5.0  5.0
5.0  5.0

```

3.2 String processing

What is a string in C?

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where

the string ends. When we define a string as `char s[10]`, the character `s[10]` is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring string by char array in C language.

```
char ch[10] = {'S', 'T', 'R', 'A', 'W', 'B', 'E', 'R', 'R', 'Y', '\0'};
```

As we know, array index starts from 0, so it will be represented as in the figure given below.

O	1	2	3	4	5	6	7	8	9	10
S	T	R	A	W	B	E	R	R	Y	\0

While declaring string, size is not mandatory. So, we can write the above code as given below:

```
char ch[] = {'S', 'T', 'R', 'A', 'W', 'B', 'E', 'R', 'R', 'Y', '\0'};
```

We can also define the **string by the string literal** in C language. For example:

```
char ch[] = "STRAWBERRY";
```

In such case, '\0' will be appended at the end of the string by the compiler.

3.2.1 Difference between char array and string literal

There are two main differences between char array and literal.

- o We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
- o The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

3.2.2 How to initialize strings?

Initialize of strings can be done in a number of ways.

```
char c[] = "abcd";  
char c[50] = "abcd";  
char c[] = {'a', 'b', 'c', 'd', '\0'};  
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

Example 5: String Example in C

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

```

{
    char ch[15] = {'S', 'T', 'R', 'A', 'W', 'B', 'E', 'R', 'R', 'Y', '\0'};
    char ch2[15] = "STRAWBERRY";
    printf("Char Array Value is: %s\n", ch);
    printf("String Literal Value is: %s\n", ch2);
    return 0;
}

```

Output

```

Char Array Value is: STRAWBERRY
String Literal Value is: STRAWBERRY

```

3.2.3 Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- a) By using the length of string
- b) By using the null character.

Let's discuss each one of them.

a) Using the length of string

Example 6: C program for counting the number of vowels in a string.

```

#include<stdio.h>
int main ()
{
    char s[11] = "haieveryone";
    int i = 0;
    int count = 0;
    while(i<11)
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count++;
        }
    }
}

```

```

    }
    i++;
}
printf ("The number of vowels %d \n",count);
}

```

OUTPUT

The number of vowels 6

b) Using the null character

Example 7: C program for counting the number of vowels by using the null character.

```

#include<stdio.h>
int main ()
{
    char s[20] = " hai everyone ";
    int i = 0;
    int count = 0;
    while(s[i] != '\0')
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count++;
        }
        i++;
    }
    printf("The number of vowels %d\n",count);
    return 0;
}

```

OUTPUT

The number of vowels 6

3.2.4 Accepting string as the input

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

Example 8:

```
#include<stdio.h>
int main ()
{
    char s[50];
    printf("Enter the string?: ");
    scanf("%s",s);
    printf("You entered: %s\n",s);
    return 0;
}
```

OUTPUT

```
Enter the string?: hai Everyone
```

```
You entered: hai
```

Note: In above output the characters after first space is removed.

To overcome this problem of space we use **fgets()**. The function reads a text line or a string from the specified file or console. And then stores it to the respective string variable.

It has syntax given by

```
fgets (string,size,stdin)
```

Example 9:

```
#include<stdio.h>
int main()
{
    char str[30];
    printf("Enter the string?: ");
    fgets(str, 30, stdin);
    printf("You have entered: %s", str);
    return 0;
}
```

OUTPUT

```
Enter the string?: Hai Everyone
```

```
You have entered: Hai Everyone
```

3.3 String handling functions

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	Returns the length of string name.
2)	strcpy(destination, source)	Copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	Concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	Compares the first string with second string. If both strings are same, it returns 0.

3.3.1 C String Length: **strlen()** function

The **strlen()** function returns the length of the given string. It doesn't count null character '\0'.

Example 10:

```
#include<stdio.h>
#include <string.h>
int main()
{
int i;
char ch[20]={'S', 'T', 'R', 'A', 'W', 'B', 'E', 'R', 'R', 'Y', '\0'};
printf("Length of string is: %d\n",i=strlen(ch));
return 0;
}
```

OUTPUT

```
Length of string is: 10
```

Example 11: Calculate Length of String without Using **strlen()** Function

```
#include <stdio.h>
int main() {
char ch[20]={'S','T','R','A','W','B','E','R','R','Y','\0'};
int i;
```

```

for (i = 0; ch[i] != '\0'; i++);
printf("Length of the string: %d \n", i);
return 0;
}

```

OUTPUT

```
Length of the string: 10
```

3.3.2 C String Copy: strcpy () function

The strcpy(destination, source) function copies the source string in destination.

Example 12:

```

#include<stdio.h>
#include <string.h>
int main(){
char ch[20]={'S', 'T', 'R', 'A', 'W', 'B', 'E', 'R', 'R', 'Y', '\0'};
char ch2[20];
strcpy(ch2,ch);
printf("Value of second string is: %s\n",ch2);
return 0;
}

```

OUTPUT

```
Value of second string is: STRAWBERRY
```

Example 13: Copy String Without Using strcpy()

```

#include <stdio.h>
int main()
{
char ch1[20]={'S','T','R','A','W','B','E','R','R','Y','\0'},ch2[20];
int i;
for (i = 0; ch1[i] != '\0'; i++)
{
ch2[i] = ch1[i];
}
ch2[i] = '\0';
printf("String ch2: %s \n", ch2);
}

```

```
    return 0;
}
```

OUTPUT

```
String ch2: STRAWBERRY
```

3.3.3 C String Concatenation: strcat () function

The strcat(first_string, second-string) function concatenates two strings and result is returned to first_string.

Example 14:

```
#include<stdio.h>
#include <string.h>
int main(){
    char ch[10]={'h','e','l','l','o','\0'};
    char ch2[10]={' ','C', '\0'};
    strcat(ch,ch2);
    printf("Value of first string is: %s \n",ch);
    return 0;
}
```

OUTPUT

```
Value of first string is: hello C
```

Example 15: Concatenate Two Strings Without Using strcat()

```
#include <stdio.h>
int main() {
    char ch[10]={'h','e','l','l','o','\0'};
    char ch2[10]={' ','C', '\0'};
    int length, j;
    while (ch[length] != '\0') {
        length++;
    }
    // concatenate ch2 to ch
    for (j = 0; ch2[j] != '\0'; j++,length++)
    {
        ch[length] = ch2[j];
    }
}
```

```

ch[length] = '\0';
printf("After concatenation: %s\n",ch);
return 0;
}

```

OUTPUT

```
After concatenation: hello C
```

3.3.4 C String Comparison: strcmp () function

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using fgets() function which reads string from the console.

Example 16: String comparison using strcmp() function

```

#include<stdio.h>
#include <string.h>
int main(){
    char str1[20],str2[20];
    printf("Enter 1st string: ");
    fgets(str1,20,stdin);//reads string from console
    printf("Enter 2nd string: ");
    fgets(str2,20,stdin);
    if(strcmp(str1,str2)==0)
        printf("Strings are equal");
    else
        printf("Strings are not equal");
    return 0;
}

```

OUTPUT

```
Enter 1st string: hello
```

```
Enter 2nd string: hello
```

Example 17: String comparison without using strcmp() function

```

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

```

```

char ch[20],ch2[20];
int i=0,flag=0;
printf("Enter 1st string:");
fgets(ch,20,stdin);
printf("Enter 2nd string:");
fgets(ch2,20,stdin);
while(ch[i]!='\0'&& ch2[i]!='\0') // while loop
{
    if(ch[i]!=ch2[i])
    {
        flag=1;
        break;
    }
    i++;
}
if(flag)
{
    printf("Strings are not equal \n");
}
else
{
    printf("Strings are equal \n");
}
return 0;
}

```

OUTPUT

```
Enter 1st string:hello
```

```
Enter 2nd string:hello
```

```
Strings are equal
```

3.3.5 C gets() and puts() functions

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

3.3.5.1 C gets() function



The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Syntax:

```
char[] gets(char[]);
```

Example 18: Reading string using gets()

```
#include<stdio.h>

int main ()
{
    char s[30];
    printf("Enter the string? : ");
    gets(s);
    printf("You entered: %s \n",s);
    return 0;
}
```

OUTPUT

```
warning: this program uses gets(), which is unsafe.
```

```
Enter the string? : hai sbce
```

```
You entered : hai sbce
```

NOTE: The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets().

The fgets() makes sure that not more than the maximum limit of characters are read.

Example 18:

```
#include<stdio.h>

int main ()
{
    char s[30];
    printf("Enter the string? : ");
    fgets(s,20,stdin);
    printf("You entered: %s \n",s);
    return 0;
}
```



OUTPUT

```
Enter the string? : hai everyone
```

```
You entered: hai everyone
```

3.3.5.2 C puts() function

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

Syntax:

```
int puts(char[]);
```

Example 19: Read a string using gets() and print it on the console using puts().

```
#include<stdio.h>
int main(){
char name[50];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name); //displays string
return 0;
}
```

OUTPUT

```
Enter your name: jack
```

```
Your name is: jack
```

3.4 Linear search and Bubble sort program, simple programs covering arrays and strings

3.4.1 Linear Search Program to search an element in an array.

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



```

int a[20],i,n,flag,x;
printf("Enter size of array: ");
scanf("%d",&n);
//Entering the elements into an array
printf("Enter %d elements in the array :\n", n);
for(i=0;i<n;i++)
{
    scanf("%d",&a[i]);
}
// printing the elements of an array
printf("\nElements in array are:");
for(i=0;i<n;i++)
{
    printf("%d ",a[i]);
}
//Entering the element to search
printf("\nEnter the number to search:= ");
scanf("%d",&x);
for(i=0;i<n;i++)
{
    if (a[i]==x)
    {
        flag=1;
        break;
    }
}
if(flag)
{
    printf("\nThe number is present in the array at index %d \n",i);
}
else{
    printf("\n\nThe number is not present in the array\n");
}

```

```
    return 0;
}
```

OUTPUT

```
Enter size of array: 5
Enter 5 elements in the array :
1
5
4
7
9
Elements in array are:1 5 4 7 9
Enter the number to search:= 1
The number is present in the array at index 0
```

3.4.2 Bubble sort Program to sort elements in an array

```
#include <stdio.h>
int main()
{
    int a[20],i,j,n,flag,temp;
    printf("Enter size of array: ");
    scanf("%d",&n);
    //Entering the elements into an array
    printf("Enter %d elements in the array :\n", n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    // printing the elements of an array
    printf("\nElements in array are:");
    for(i=0;i<n;i++)
    {
        printf("%d ",a[i]);
    }
    printf("\n\n");
```

```

//Bubble sort algorithm

printf("Bubble sort algorithm began:-----\n\n");
for(i=0;i<n-1;i++)
{
    for (j=0;j<n-i-1;j++)
    {
        if (a[j] > a[j + 1])
        {
            temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
        }
    }
}
printf("Sorted Elements in array are:");
for(i=0,i<n;i++)
{
    printf("%d ",a[i]);
}
printf("\n\n");
return 0;
}

```

OUTPUT

```

Enter size of array: 5
Enter 5 elements in the array :
8
5
7
1
2
Elements in array are:8 5 7 1 2
Bubble sort algorithm began: -----
Sorted Elements in array are:1 2 5 7 8

```

3.4.3 Simple programs covering arrays and strings

Example 1: Defining a matrix with Rows and Column, entering the matrix elements and displaying them

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a[10][10],i,j,row,col;
    printf ("enter the number of rows: \n");
    scanf("%d",&row);
    printf ("enter the number of Coloums: \n");
    scanf("%d",&col);
    // entering the elements of A matrix
    printf("enter the elements row by row for A matrix: \n");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    // Display A matrix
    printf("Display A matrix: \n");
    for(i=0;i<row;i++)
    {
        printf("\n");
        for(j=0;j<col;j++)
        {
            printf("%5d",a[i][j]);
        }
    }
    printf("\n");
}
```

OUTPUT

enter the number of rows:

2

enter the number of Columns:

2

enter the elements row by row for A matrix:

1

2

3

4

Display A matrix:

1 2

3 4

Example 2: Defining two matrices A and B and perform matrix addition and displaying the output as C matrix.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a[10][10],b[10][10],c[10][10],i,j,row,col;
    printf("enter the number of rows: \n");
    scanf("%d",&row);
    printf("enter the number of Columns: \n");
    scanf("%d",&col);
    // entering the elements of A matrix
    printf("enter the elements row by row for A matrix: \n");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
}
```

```

// entering the elements of B matrix
printf("enter the elements row by row for B matrix: \n");
for(i=0;i<row;i++)
{
    for(j=0;j<col;j++)
    {
        scanf("%d",&b[i][j]);
    }
}

// Display A matrix
printf("Display A matrix: \n");
for(i=0;i<row;i++)
{
    printf("\n");
    for(j=0;j<col;j++)
    {
        printf("%5d",a[i][j]);
    }
}
printf("\n");

// Display B matrix
printf("Display B matrix: \n");
for(i=0;i<row;i++)
{
    printf("\n");
    for(j=0;j<col;j++)
    {
        printf("%5d",b[i][j]);
    }
}
printf("\n");

//Adding A and B matrix to get C matrix
for(i=0;i<row;i++)
{

```



```

for(j=0;j<col;j++)
{
    c[i][j]=a[i][j]+b[i][j];
}
// Display C matrix
printf("Display C matrix: \n");
for(i=0;i<row;i++)
{
    printf("\n");
    for(j=0;j<col;j++)
    {
        printf("%5d",c[i][j]);
    }
}
printf("\n");
}

```

OUTPUT

enter the number of rows:

2

enter the number of Columns:

2

enter the elements row by row for A matrix:

1

2

3

4

enter the elements row by row for B matrix:

1

2

3

4

Display A matrix:

1 2

```
3 4  
Display B matrix:
```

```
1 2  
3 4
```

```
Display C matrix:  
2 4  
6 8
```

Example 3: Transposing a matrix A to obtain the output as C matrix.

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    int a[10][10],c[10][10],i,j,row,col;  
    printf ("enter the number of rows: \n");  
    scanf("%d",&row);  
    printf ("enter the number of Columns: \n");  
    scanf("%d",&col);  
    // entering the elements of A matrix  
    printf("enter the elements row by row for A matrix: \n");  
    for(i=0;i<row;i++)  
    {  
        for(j=0;j<col;j++)  
        {  
            scanf("%d",&a[i][j]);  
        }  
    }  
    // Display A matrix  
    printf("Display A matrix: \n");  
    for(i=0;i<row;i++)  
    {  
        printf("\n");  
        for(j=0;j<col;j++)  
        {
```

```

printf("%5d",a[i][j]);
}
}
printf("\n");
//Transposing A matrix to get C matrix
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
c[j][i]=a[i][j];
}
}
// Display C matrix
printf("Display Transposed C matrix: \n");
for(i=0;i<row;i++)
{
printf("\n");
for(j=0;j<col;j++)
{
printf("%5d",c[i][j]);
}
}
printf("\n");
}

```

OUTPUT

```

enter the number of rows:
2
enter the number of Columns:
2
enter the elements row by row for A matrix:
2
6
4
5

```

Display A matrix:

```
2 6
4 5
```

Display Transposed C matrix:

```
2 4
6 5
```

Example 4: Matrix multiplication.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a[10][10],b[10][10],c[10][10],i,j,k,val,row,col;
    printf ("enter the number of rows in matrix: \n");
    scanf("%d",&row);
    printf ("enter the number of Columns in matrix: \n");
    scanf("%d",&col);
    printf ("enter the number of Columns in first matrix: \n");
    scanf("%d",&val);
    // entering the elements of A matrix
    printf("enter the elements row by row for A matrix: \n");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    // entering the elements of B matrix
    printf("enter the elements row by row for B matrix: \n");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
```

```

scanf("%d",&b[i][j]);
}
}

// Display A matrix

printf("Display A matrix: \n");
for(i=0;i<row;i++)
{
printf("\n");
for(j=0;j<col;j++)
{
printf("%5d",a[i][j]);
}
}

printf("\n");

// Display B matrix

printf("Display B matrix: \n");
for(i=0;i<row;i++)
{
printf("\n");
for(j=0;j<col;j++)
{
printf("%5d",b[i][j]);
}
}

printf("\n");

//Multiply A and B matrix to get C matrix

for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
c[i][j] = 0;
for (k = 0; k < val; k++)
{
c[i][j] = c[i][j] + (a[i][k] * b[k][j]);
}
}
}

```



```

    }
}
}

// Display C matrix

printf("Display Multiplier output C matrix: \n");

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

return 0;
}

```

OUTPUT

enter the number of rows in matrix:

2

enter the number of Columns in matrix:

2

enter the number of Columns in first matrix:

2

enter the elements row by row for A matrix:

1

2

3

4

enter the elements row by row for B matrix:

4

3

2

1

Display A matrix:



```
1 2
3 4
```

Display B matrix:

```
4 3
2 1
```

Display Multiplier output C matrix:

```
8 5
20 13
```

Example 5: Write a C program to check whether a given matrix is a diagonal matrix.

```
#include <stdio.h>
int main()
{
    int x[10][10], nr, nc, r, c, flag = 0;
    printf("Enter the number of rows and columns: \n");
    scanf("%d %d", &nr, &nc);
    // checking for square matrix
    if(nr==nc)
    {
        // Entering the elements of matrix
        printf("Enter elements of the matrix:\n");
        for(r=0 ; r<nr ; r++)
        {
            for(c=0 ; c<nc ; c++)
            {
                scanf("%d", &x[r][c]);
            }
        }
        // Checking for diagonal elements
        for(r=0 ; r<nr ; r++)
        {
            for(c=0 ; c<nc ; c++)
            {
                if(r!=c && x[r][c]!= 0)
```

```
{  
    flag=1;  
    break;  
}  
}  
}  
if(flag==0)  
    printf("The matrix is diagonal \n");  
else  
    printf("The matrix is not diagonal \n");  
}  
else  
{  
    printf("The matrix is not a square matrix \n");  
}  
return 0;  
}
```

OUTPUT

```
Enter the number of rows and columns:
```

```
3 3
```

```
Enter elements of the matrix:
```

```
1  
0  
0  
0  
2  
0  
0  
0  
3
```

```
The matrix is diagonal
```

Example 6: Write a C program to sum the diagonal elements of a matrix.

```
#include <stdio.h>
int main()
{
    int x[10][10], nr, nc, r, c, sum =0;
    printf("Enter the number of rows and columns: \n");
    scanf("%d %d", &nr, &nc);
    // checking for square matrix
    if(nr==nc)
    {
        // Entering the elements of matrix
        printf("Enter elements of the matrix:\n");
        for(r=0 ; r<nr ; r++)
        {
            for(c=0 ; c<nc ; c++)
            {
                scanf("%d", &x[r][c]);
            }
        }
        // Checking for diagonal elements
        for(r=0 ; r<nr ; r++)
        {
            for(c=0 ; c<nc ; c++)
            {
                if(r==c)
                {
                    sum = sum + x[r][c];
                }
            }
        }
        printf("The sum of diagonal elements of the matrix is :%d \n",sum);
    }
    else
}
```

```

    {
        printf("The matrix is not a square matrix \n") ;
    }
    return 0;
}

```

OUTPUT

Enter the number of rows and columns:

3 3

Enter elements of the matrix:

1
2
3
4
5
6
7
8
9

The sum of diagonal elements of the matrix is :15

Example 7: Write a C program to find the sum of all row elements of a matrix.

```

#include <stdio.h>
int main()
{
    int x[10][10], nr, nc, r, c, sum;
    printf("Enter the number of rows and columns: \n") ;
    scanf("%d %d", &nr, &nc) ;
    // Entering the elements of matrix
    printf("Enter elements of the matrix:\n") ;
    for(r=0 ; r<nr ; r++)
    {
        for(c=0 ; c<nc ; c++)
        {
            scanf("%d", &x[r][c]) ;
        }
    }
    sum = 0;
    for(r=0 ; r<nr ; r++)
    {
        for(c=0 ; c<nc ; c++)
        {
            sum = sum + x[r][c];
        }
    }
    printf("The sum of all row elements of the matrix is : %d", sum) ;
}

```

```

        }
    }

// Checking for sum elements

printf("The sum of row elements of the matrix is :\n");
for(r=0 ; r<nr ; r++)
{
    sum=0;
    for(c=0 ; c<nc ; c++)
    {
        sum = sum + x[r][c];
    }
}
printf("Sum of row %d = %d \n",r+1,sum);
}

return 0;
}

```

OUTPUT

Enter the number of rows and columns:

3 3

Enter elements of the matrix:

1
2
3
4
5
6
7
8
9

The sum of row elements of the matrix is :

Sum of row 1 = 6

Sum of row 2 = 15

Sum of row 3 = 24

Example 8: Write a C program to find the sum of all column elements of a matrix.

```
#include <stdio.h>
int main()
{
    int x[10][10], nr, nc, r, c, sum;
    printf("Enter the number of rows and columns: \n");
    scanf("%d %d", &nr, &nc);
    // Entering the elements of matrix
    printf("Enter elements of the matrix:\n");
    for(r=0 ; r<nr ; r++)
    {
        for(c=0 ; c<nc ; c++)
        {
            scanf("%d", &x[r][c]);
        }
    }
    // Checking for sum elements
    printf("The sum of row elements of the matrix is :\n");
    for(c=0 ; c<nc ; c++)
    {
        sum=0;
        for(r=0 ; r<nr ; r++)
        {
            {
                sum = sum + x[r][c];
            }
        }
        printf("Sum of column %d = %d \n", c+1, sum);
    }
    return 0;
}
```

OUTPUT

Enter the number of rows and columns:

3 3

Enter elements of the matrix:

1

2

3

4

5

6

7

8

9

The sum of row elements of the matrix is :

Sum of column 1 = 12

Sum of column 2 = 15

Sum of column 3 = 18

Example 9: Write a C program to convert a lowercase string to uppercase.

```
#include<stdio.h>
int main()
{
    char s[20];
    int i = 0;
    printf("Enter a string: ");
    fgets(s,20,stdin);
    while( s[i] != '\0' )
    {
        // if character is in lowercase then subtract 32
        if( s[i] >= 'a' && s[i] <= 'z' )
        {
            s[i] = s[i] - 32;
        }
        i++;
    }
    printf("In Upper Case is: ");
```



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

OUTPUT

```
Enter a string: Hai Sbceans
In Upper Case is: HAI SBCEANS
```

Example 10: Write a C program to convert an uppercase string to lowercase.

```
#include<stdio.h>
int main()
{
    char s[20];
    int i = 0;
    printf("Enter a string: ");
    fgets(s,20,stdin);
    while( s[i] != '\0' )
    {
        // if character is in uppercase then add 32
        if( s[i] >= 'A' && s[i] <= 'Z' )
        {
            s[i] = s[i] + 32;
        }
        i++;
    }
    printf("In Lower Case is: ");
    puts(s);
    return 0;
}
```

OUTPUT

```
Enter a string: Hai SBCEANS
In Lower Case is: hai sbceans
```

Example 11: Write a C program to read an English Alphabet through keyboard and display whether the given Alphabet is in upper case or lower case.

```
#include<stdio.h>
int main()
{
    char s[20];
    int i = 0;
    printf("Enter the Alphabet: ");
    fgets(s,20,stdin);
    if( s[i] >= 'A' && s[i] <= 'Z' )
    {
        printf("A Upper Case Alphabet \n");
    }
    else if (s[i] >= 'a' && s[i] <= 'z' )
    {
        printf("A Lower Case Alphabet \n");
    }
    else
    {
        printf("The entered Value is nor an Alphabet \n");
    }
    return 0;
}
```

OUTPUT

```
Enter the Alphabet: a
A Lower Case Alphabet
Enter the Alphabet: J
A Upper Case Alphabet
Enter the Alphabet: $
The entered Value is nor an Alphabet
```

Example 12: C Program to Remove all Characters in a String except Alphabet and space using another String

```
#include<stdio.h>
int main()
{
    char str1[20],str2[20];
    int i, j;
    printf("Enter string: ");
    fgets(str1,20,stdin);

    for(i=0,j=0;str1[i]!='\0';i++)
    {
        if((str1[i]>='A' && str1[i]<='Z')||(str1[i]>='a' && str1[i]<='z')||(str1[i]==' '))
        {
            str2[j]=str1[i];
            j++;
        }
    }
    printf("Displaying string: %s \n",str2);
    return 0;
}
```

OUTPUT

```
Enter string: Ha3565i Bu2399ddy
```

```
Displaying string: Hai Buddy
```

Example 13: C Program to Remove all Characters in a String except Alphabet and space without using another String

```
#include<stdio.h>
int main()
{
    char str1[100];
    int i,j;
    printf("Enter string: ");
```

```

fgets(str1,100,stdin);

for(i=0;str1[i]!='\0';i++)
{
    while(!((str1[i]>='A' && str1[i]<='Z')||(str1[i]>='a' && str1[i]<='z')||(str1[i]==' ')||str1[i]
    == '\0'))
    {
        for(j=i;str1[j]!='\0';j++)
        {
            str1[j]=str1[j+1];
        }
        str1[j] = '\0';
    }
}
printf("Displaying string: %s \n",str1);
return 0;
}

```

OUTPUT

Enter string: Ha76766i Bud88898dy

Displaying string: Hai Buddy

Example 14: C Program to Find Frequency of Characters in a String

```

#include<stdio.h>

int main()
{
    char ch,str[100];
    int i, freq=0;
    printf("Enter string: ");
    fgets(str,100,stdin);
    printf("Enter a character to check its frequency of occurrence: ");
    scanf("%c",&ch);
    for(i=0;str[i]!='\0';i++)
    {
        if(ch==str[i])
    }
}

```

```
{  
    freq ++;  
}  
}  
printf("The frequency of %c is= %d \n",ch,freq);  
return 0;  
}
```

OUTPUT

```
Enter string: Hai everyone today is a beautiful day  
Enter a character to check its frequency of occurrence: a  
The frequency of a is= 5
```

THIS PAGE IS INTENTIONALLY LEFT BLANK

MODULE-IV

Working with functions	
4.1	Introduction to modular programming, writing functions, formal parameters, actual parameters
4.2	Pass by Value, Recursion, Arrays as Function Parameters
4.3	structure, union, Storage Classes, Scope and life time of variables, <i>simple programs using functions</i>

4.1 Working with functions

Introduction to modular programming

Modular programming is the practice of splitting the code into different blocks/files (modules) and then importing those code so that it could be used in many different projects.

Typically, code that is been reused goes into its own module so as to be imported rather than copy and paste. This makes our programs more readable, easier to scale and nicer to work with. A module is typically any program file that contains only functions and classes. In our case as we are dealing with C programming, only functions will be considered.

Points which should be taken care of prior to modular program development:

1. Limitations of each and every module should be decided.
2. In which way a program is to be partitioned into different modules.
3. Communication among different modules of the code for proper execution of the entire program.

Advantages of Using Modular Programming Approach:

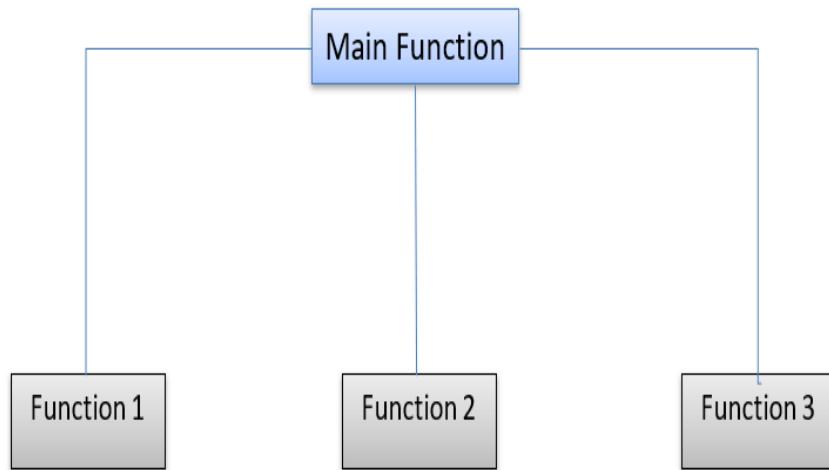
1. **Ease of Use:** This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go, we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
2. **Reusability:** It allows the user to reuse the functionality with a different interface without typing the whole program again.
3. **Ease of Maintenance:** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

4.1.1 Functions in C

What is a Function in C?

Function in C programming is a reusable block of code that makes a program easier to understand, test and can be easily modified without changing the main/calling program.

Functions divide the code and modularize the program for better and effective results. In short, a larger program is divided into various subprograms which are called as functions.



Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- Functions can be called any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions, that means same function can be called for multiple programs.

4.1.2 Types of function

There are two types of function in C programming:

- **Standard library functions**
- **User-defined functions**

4.1.2.1 Standard library functions

The standard library functions are built-in functions in C programming.

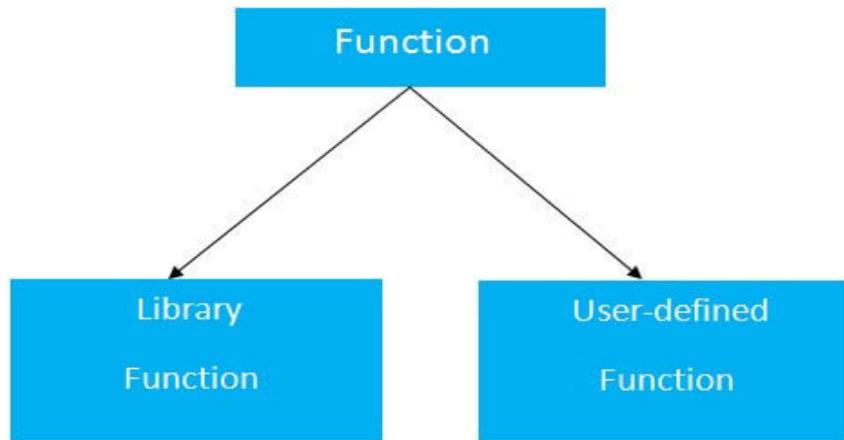
These functions are defined in header files. For example,

- The `printf()` is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the `stdio.h` header file. Hence, to use the `printf()` function, we need to include the `stdio.h` header file using `#include <stdio.h>`.

- The `sqrt()` function calculates the square root of a number. The function is defined in the `math.h` header file. Hence, to use the `sqrt()` function, we need to include the `math.h` header file using `#include <math.h>`.

4.1.2.2 User-defined function

A user can create functions as per his/her need. Such functions created by the users are known as user-defined functions.



4.1.3 Working of a function in C

In order to understand the working of Function in C, let us assume the case of a user defined function as mentioned below.

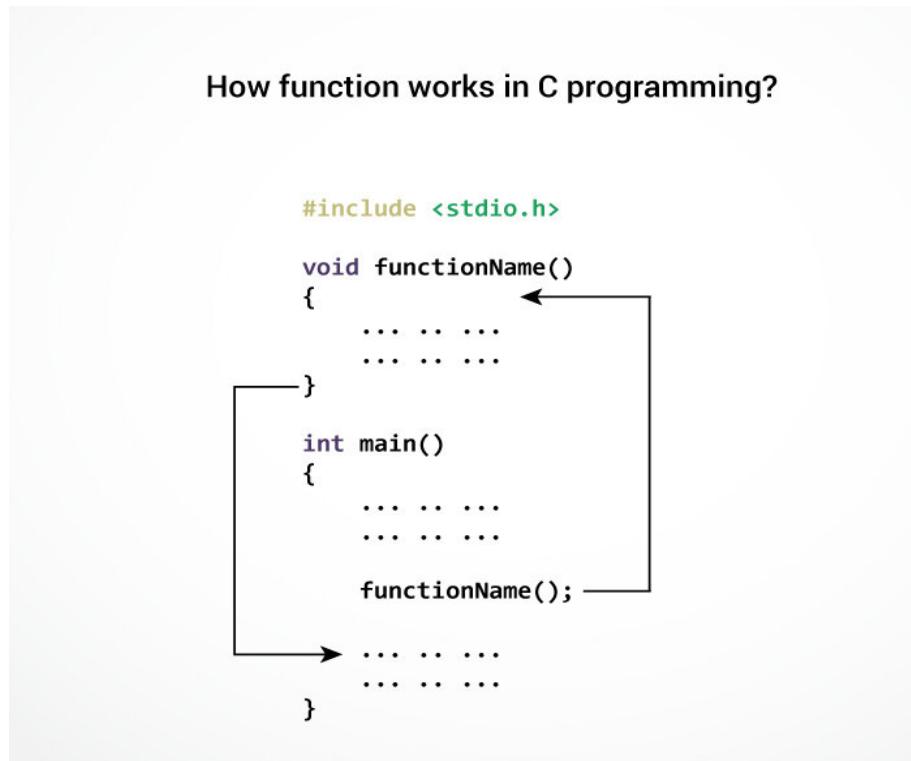
```

#include <stdio.h>
void functionName()
{
    ...
}
int main()
{
    ...
}

functionName ();
...
}
  
```

The execution of a C program begins from the `main()` function. When the compiler encounters `functionName ()`; control of the program jumps to `void functionName()`.

And, the compiler starts executing the codes inside `functionName()`. The control of the program jumps back to the `main()` function once code inside the function definition is executed.



4.1.4 Declaration / Defining a function

```
return_type function_name (argument list)
{
    Set of statements – Block of code
}
```

return_type: Return type can be of any data type such as `int`, `double`, `char`, `void`, `short` etc.

Don't worry you will understand these terms better once you go through the examples below.

function_name: It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing its name. Like `primechecker` , `addnumbers` etc

argument list: Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument say `(int a,int b)`.

Block of code: Set of C statements, which will be executed whenever a call will be made to the function.

4.1.5 Calling a function

Control of the program is transferred to the user-defined function when called within the program.

Syntax of function call:

functionName(argument1, argument2, ...);

When a function is called, the control of the program is transferred to the function defined and the compiler starts executing the codes inside the body of a function.

4.1.6 Example of a User-defined function

Example 1: To add two integers using user-defined function `addNumbers()`.

```
#include <stdio.h>

int addNumbers(int a, int b);      // function prototype

int main()
{
    int n1,n2,sum;
    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);
    sum = addNumbers(n1, n2);      // function call
    printf("sum = %d\n",sum);
    return 0;
}

int addNumbers(int a, int b)      // Defining a function
{
    int result;
    result = a+b;
    return result;              // Return statement
}
```

OUTPUT

```
Enters two numbers: 10 20
```

```
sum = 30
```

Note: A **function prototype** is

- Simply the declaration of a function that specifies function's name, parameters and return type but not containing the block of codes.
- Informs the compiler that the function may later be used in the program.

- Not needed if the user-defined function is defined before the function is called.

A Return statement

- **Return 0:** This statement in the main function means that the program executed successfully. Syntax--**return 0;**
- **Return value:** This statement in the user defined function means that the program will return a value back to the main program. Syntax--**return x;**
- **Return expression:** This statement in the user defined function means that the program will first execute the expression and then return the value back to the main program. Syntax--**return (x*y/z);**
- **Return:** The return statement does not return value to the caller.
Syntax—**return;**

4.1.7 Different aspects of function calling in C

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.

- Function **without** arguments and **without** return value
- Function **without** arguments and **with** return value
- Function **with** arguments and **without** return value
- Function **with** arguments and **with** return value

4.1.7.1 Function **without** arguments and **without** return value

Example 2: Checking a given number is prime or not.

```
#include <stdio.h>
void checkPrimeNumber();
int main()
{
    checkPrimeNumber();
    return 0;
}
//returns no value and no argument passed
void checkPrimeNumber()
{
    int n, i, flag = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
```

```

for(i=2;i <= n-1; i++)
{
    if(n%i == 0)
    {
        flag = 1;
    }
}
if(flag == 1)
    printf("%d is not a prime number.", n);
else
    printf("%d is a prime number.", n);
}

```

The `checkPrimeNumber()` function takes input from the user, checks whether it is a prime number or not and displays it on the screen.

The empty parentheses in `checkPrimeNumber();` statement inside the `main()` function indicates that **no argument** is passed to the function.

The return type of the function is **void**. Hence, **no value** is returned from the function.

4.1.7.2 Function **without** arguments and **with** return value

Example 3: Checking a given number is prime or not.

```

#include <stdio.h>
int checkPrimeNumber();
int main()
{
    int num ;
    num = checkPrimeNumber();
    if (num == 1)
        printf("The number is not a prime number.");
    else
        printf("The number is a prime number.");
    return 0;
}
//returns a value but no argument passed
int checkPrimeNumber()

```

```
{
    int n, i, flag=0;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    for(i=2;i <= n-1; i++)
    {
        if(n%i == 0)
        {
            flag = 1;
            break;
        }
    }
    return flag;
}
```

The empty parentheses in `checkPrimeNumber()`; statement inside the `main()` function indicates that **no argument** is passed to the function.

The return type of the function is **int**. Hence, **value** is returned from the function.

4.1.7.3 Function **with** arguments and **without** return value

Example 4: Checking a given number is prime or not.

```
#include <stdio.h>
void checkPrimeNumber(int n);
int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    checkPrimeNumber(n);
    return 0;
}
//returns no value but argument passed
void checkPrimeNumber(int n)
{
    int i, flag = 0;
```

```

for(i=2;i <= n-1; i++)
{
    if(n%i == 0)
    {
        flag = 1;
    }
}
if(flag == 1)
    printf("%d is not a prime number.",n);
else
    printf("%d is a prime number.",n);
}

```

The parentheses in `checkPrimeNumber(int n)` is not empty; thus statement inside the `main()` function indicates that `argument` is passed into the function.

The return type of the function is `void`. Hence, `no value` is returned from the function.

4.1.7.4 Function with arguments and with return value

Example 5: Checking a given number is prime or not.

```

#include <stdio.h>

int checkPrimeNumber(int n);

int main()
{
    int n,num;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    num = checkPrimeNumber(n);
    if(num == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.",n);
    return 0;
}

//returns value and arguments are passed
int checkPrimeNumber(int n)

```

```

{
    int i, flag = 0;
    for(i=2;i <= n-1; i++)
    {
        if(n%i == 0)
        {
            flag = 1;
            break;
        }
    }
    return flag;
}

```

The parentheses in `checkPrimeNumber(int n)` is not empty; thus statement inside the `main()` function indicates that `argument` is passed into the function.

The return type of the function is `int`. Hence, `value` is returned from the function.

4.1.8 Actual and Formal arguments in C

4.1.8.1 Actual arguments

Arguments which are mentioned in the function call is known as the actual argument.

For example:

```
func1(12, 23);
```

Here 12 and 23 are actual arguments.

Actual arguments can be constant, variables, expressions etc.

1	<code>func1(a, b); // here actual arguments are variable</code>
2	<code>func1(a + b, b + a); // here actual arguments are expression</code>

4.1.8.2 Formal Arguments

Arguments which are mentioned in the definition of the function is called formal arguments. Formal arguments are very similar to local variables inside the function. Just like local variables, formal arguments are destroyed when the function ends. Here `n` is the formal argument.

1	int factorial (int n)
2	{
3	// write logic here
4	}

Things to remember about actual and formal arguments.

1. Order, number, and type of the actual arguments in the function call must match with formal arguments of the function.
2. If there is type mismatch between actual and formal arguments then the compiler will try to convert the type of actual arguments to formal arguments if it is legal, Otherwise, a garbage value will be passed to the formal argument.
3. Changes made in the formal argument do not affect the actual arguments.

Example 6: The following program demonstrates above behaviour.

```
#include<stdio.h>
void func_1(int);
int main()
{
    int x = 10;
    printf("Before function call\n");
    printf("x = %d\n", x);
    func_1(x);
    printf("After function call\n");
    printf("x = %d\n", x);
    return 0;
}
```

```
void func_1(int a)
{
    a += 1;
    a++;
    printf("\n a = %d\n\n", a);
}
```

OUTPUT

Before function call

```
x = 10
a = 12
After function call
x = 10
```

Here the value of variable x is 10 before the function `func_1()` is called, after `func_1()` is called, the value of x inside `main()` is still 10. The changes made inside the function `func_1()` doesn't affect the value of x. This happens because when we pass values to the functions, a copy of the value is made and that copy is passed to the formal arguments. Hence Formal arguments work on a copy of the original value, not the original value itself, that's why changes made inside `func_1()` is not reflected inside `main()`. This process is known as passing arguments using **Call by Value**.

4.2 Working with functions continued.....

4.2.1 Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we cannot modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- 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.

4.2.1.1 Call by Value Example:

Example 7: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int,int);
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b);
```

```

swap(a,b);
printf("After swapping values in main a = %d, b = %d\n",a,b);
}

void swap (int a, int b)
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b);
}

```

OUTPUT

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

4.2.2 Recursion in C

Till now we have used multiple functions that can be called into the main program or other functions, but in some case, it is useful to have functions that call themselves. Any such function which calls itself is called recursive function, and are called recursive calls.

In C, such function which calls itself is called recursive function and the process is called recursion. While using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

4.2.2.1 Recursion function

The structure of a recursion program

```

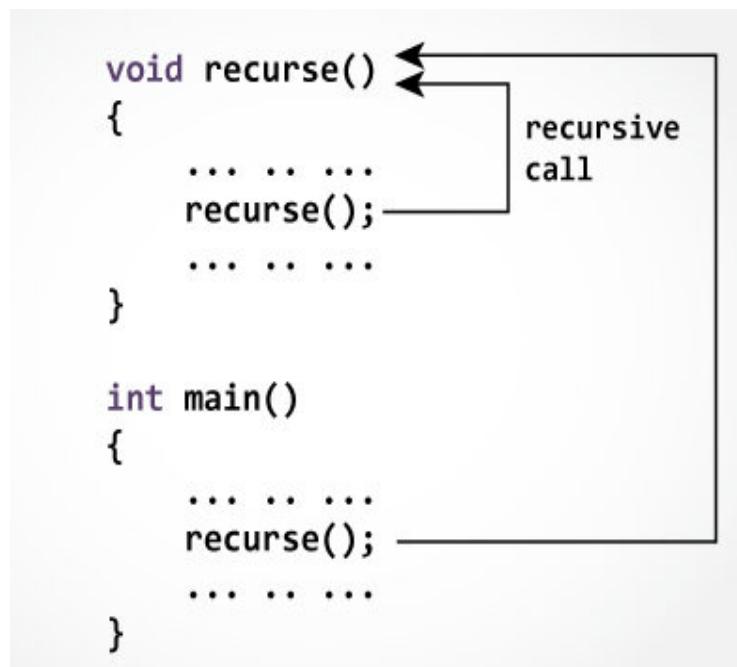
void recurse()
{
    ...
    recurse();
    ...
}

```

```
int main()
{
    .....
    recurse();
    .....
}
```

4.2.2.2 How recursion works?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0. If the base case is not reached or not defined, then the stack overflow problem may arise.



Let us understand with an example.

Example 8: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>

int sum(int n);

int main() {
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum = %d \n", result);
    return 0;
}
```

```

int sum(int n) {
    if(n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}

```

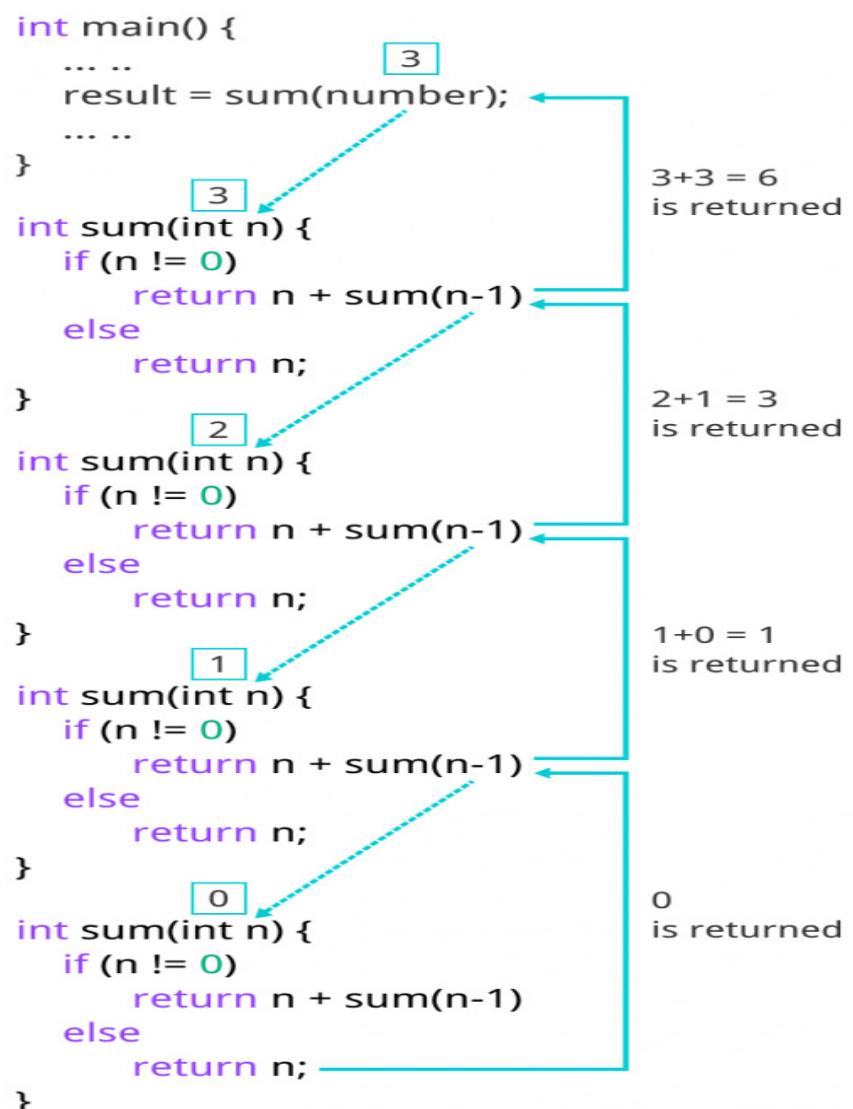
OUTPUT

```

Enter a positive integer: 3
sum = 6

```

We can understand the above program of the recursive method call by the figure given below:



Example 9: Computing factorial of a number

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number to calculate factorial: ");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d \n",f);
}
int fact(int n)
{
    if (n==0)
    {
        return 1;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```

OUTPUT

```
Enter the number to calculate factorial: 5
```

```
factorial = 120
```

We can understand the above program of the recursive method call by the figure given below:

```

return 5 * factorial(4) = 120
  ↘ return 4 * factorial(3) = 24
    ↘ return 3 * factorial(2) = 6
      ↘ return 2 * factorial(1) = 2
        ↘ return 1 * factorial(0) = 1

```

$$1 * 2 * 3 * 4 * 5 = 120$$

4.2.3 Arrays as Function Parameters / Passing arrays as parameter to function

In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

Now let's see a few examples where we will **pass a single array element as argument** to a function, **a one-dimensional array** to a function and **a multidimensional array** to a function.

4.2.3.1 Passing a single array element to a function

Let's write a very simple program, where we will declare and define an array of integers in our main() function and pass one of the array elements to a function, which will just print the value of the element.

Example 10: Passing a single array element to a function

```

#include<stdio.h>
void dispArrayelement(int a);
int main()
{
    int array1[] = {1,2,3,4,5,6 };
    dispArrayelement(array1[4]);
    return 0;
}

```

```
void dispArrayelement(int a)
{
    printf("The array element is %d \n", a);
}
```

OUTPUT

```
The array element is 5
```

4.2.3.2 Passing a complete One-dimensional array to a function

To understand how this is done, let's write a function to find out average of all the elements of the array and print it.

We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

Example 11: Passing a complete One-dimensional array to a function

```
#include<stdio.h>

float findAverage(int marks[]);

int main()
{
    float avg;
    int marks[] = {99, 90, 96, 93, 95};
    avg = findAverage(marks);
    printf("Average marks = %.2f \n", avg);
    return 0;
}

float findAverage(int marks[])
{
    int i, sum = 0;
    float avg;
    for (i = 0; i <= 4; i++)
    {
        sum = sum + marks[i];
    }
    avg = (sum / 5);
    return avg;
}
```

OUTPUT

Average marks = 94.00

4.2.3.3 Passing a Multi-dimensional array to a function

Here again, we will only pass the name of the array as argument.

Example 12: Passing a Multi-dimensional array to a function

```
#include<stdio.h>
void displayArray(int arr[3][3]);
int main()
{
    int arr[3][3], i, j;
    printf("Please enter 9 numbers for the array: \n");
    for (i = 0; i < 3; ++i)
    {
        for (j = 0; j < 3; ++j)
        {
            scanf("%d", &arr[i][j]);
        }
    }
    displayArray(arr);
    printf("\n");
    return 0;
}
void displayArray(int arr[3][3])
{
    int i, j;
    printf("The complete array is: \n");
    for (i = 0; i < 3; ++i)
    {
        printf("\n");
        for (j = 0; j < 3; ++j)
        {
            printf("%d\t", arr[i][j]);
        }
    }
}
```

```
    }  
}
```

OUTPUT

```
Please enter 9 numbers for the array:
```

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

The complete array is:

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

4.2.3.4 More Examples

Example 13: Display the minimum number in an array.

```
#include<stdio.h>  
  
int minarray(int arr[],int size);  
  
int main()  
{  
    int i=0,min=0;  
    int numbers[]={4,5,7,3,8,2};  
    min=minarray(numbers,6);  
    printf("minimum number is %d \n",min);  
    return 0;  
}  
  
int minarray(int arr[],int size)  
{  
    int min=arr[0];  
    int i=0;  
    for(i=1;i<size;i++)
```



```

{
    if(min>arr[i])
    {
        min=arr[i];
    }
}
return min;
}

```

OUTPUT

```
minimum number is 2
```

Example 14: To sort the elements of an array in ascending array

```

#include<stdio.h>
void Bubble_Sort(int[]);
int main ()
{
    int arr[10] = {10,9,7,101,23,44,12,78,34,23};
    Bubble_Sort(arr);
    return 0;
}
void Bubble_Sort(int a[])
{
    int i, j,temp;
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}

```

```

    }
}

printf("Printing Sorted Element List ...\\n");
for(i = 0; i<10; i++)
{
    printf("%d\\n",a[i]);
}
}

```

OUTPUT

Printing Sorted Element List ...

```

7
9
10
12
23
23
34
44
78
101

```

4.3 Working with functions continued....

4.3.1 C Structure

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.

In structure, data is stored in the form of **records**.

4.3.1.1 Defining a structure in C

Each member in the struct statement is a normal variable definition like int I; or float f; or some other valid variable definition and the **struct tag** is optional.

Example 15: To declare the student structure

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    char gender;
};
```

Here **struct Student** declares a structure to hold the details of a student which consists of 4 data fields, namely **name**, **age**, **branch** and **gender**. These fields are called **structure elements or members**.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag**.

4.3.1.2 Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

1) Declaring Structure variables separately

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    char gender;
};

struct Student S1, S2; //declaring variables of struct Student
```

2) Declaring Structure variables with structure definition

```
struct Student
{
```



```

char name[25];
int age;
char branch[10];
char gender;
}S1, S2;

```

Note: Here S1 and S2 are variables of structure Student. However, this approach is not much recommended.

4.3.1.3 Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a **dot** (.) operator also called **period** or **member access** operator.

Example 16: Declaring Structure variables separately

```

#include<stdio.h>
#include<string.h>
struct Student
{
    char name[25];
    int age;
    char branch[10];
    char gender[10];
};
int main()
{
    struct Student s1;
    s1.age = 18;
    strcpy(s1.name, "john");
    strcpy(s1.branch, "ECE");
    strcpy(s1.gender, "Male");
    printf("Name of Student 1: %s\n", s1.name);
    printf("Gender of Student 1: %s\n", s1.gender);
    printf("Age of Student 1: %d\n", s1.age);
}

```

```
    printf("Branch of Student 1: %s\n", s1.branch);
    return 0;
}
```

OUTPUT

```
Name of Student 1: john
Gender of Student 1: Male
Age of Student 1: 18
Branch of Student 1: ECE
```

Example 17: Declaring Structure variables with structure definition

```
#include<stdio.h>
#include<string.h>
struct Student
{
    char name[25];
    int age;
    char branch[10];
    char gender[10];
}s1;
int main()
{
    s1.age = 18;
    strcpy(s1.name, "john");
    strcpy(s1.branch, "ECE");
    strcpy(s1.gender, "Male");
    printf("Name of Student 1: %s\n", s1.name);
    printf("Gender of Student 1: %s\n", s1.gender);
    printf("Age of Student 1: %d\n", s1.age);
    printf("Branch of Student 1: %s\n", s1.branch);
    return 0;
}
```

OUTPUT

```
Name of Student 1: john
Gender of Student 1: Male
```

Age of Student 1: 18

Branch of Student 1: ECE

We can also use `scanf()` to give values to structure members through terminal.

```
scanf(" %s ", s1.name);  
scanf(" %d ", &s1.age);
```

4.3.1.4 Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

Type1 initialization:

```
struct Patient  
{  
    float height;  
    int weight;  
    int age;  
};  
struct Patient p1 = { 170 , 73, 23 }; //initialization directly
```

Type2 initialization:

```
struct Patient  
{  
    float height;  
    int weight;  
    int age;  
};  
//initialization of each member separately  
struct Patient p1;  
p1.height = 170;  
p1.weight = 73;  
p1.age = 23;
```

4.3.1.5 Array of Structure

We can also declare an array of structure variables. in which each element of the array will represent a structure variable. Example: `struct Student s[5];`

The below program defines an array `s` of size 5. Each element of the array `s` is of type Student.

Example 18:

```
#include<stdio.h>

struct Student
{
    char name[10];
    int roll;
};

struct Student s[5];
int i;
void Enquiry();
// main program
int main()
{
    Enquiry();
    return 0;
}
//function
void Enquiry()
{
    for(i = 0; i < 3; i++)
    {
        printf("\nEnter record of Student No: %d :\n", i+1);
        printf("\nEnter name: \t");
        scanf("%s", s[i].name);
        printf("\nEnter Roll number: \t");
        scanf("%d", &s[i].roll);
    }
    for(i = 0; i < 3; i++)
    {
        printf("\nDisplaying Student No record: %d :\n", i+1);
        printf("\nStudent name is %s\n", s[i].name);
        printf("\nRoll number is %d\n", s[i].roll);
    }
}
```

OUTPUT

```
Enter record of Student No: 1 :
```

```
Name: jay
```

```
Enter Roll number: 1
```

```
Enter record of Student No: 2 :
```

```
Name: sam
```

```
Enter Roll number: 2
```

```
Enter record of Student No: 3 :
```

```
Name: dan
```

```
Enter Roll number: 3
```

```
Displaying Student No record: 1 :
```

```
Student name is jay
```

```
Roll number is 1
```

```
Displaying Student No record: 2 :
```

```
Student name is sam
```

```
Roll number is 2
```

```
Displaying Student No record: 3 :
```

```
Student name is dan
```

```
Roll number is 3
```

We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

Example 19:

```
#include<stdio.h>
struct Student
{
    char name[10];
    int roll;
};

void display(struct Student st);
int main()
{
    struct Student s;
    printf("\nEnter Student record:\n");
```

```

printf("\nStudent name:\t");
scanf("%s", s.name);
printf("\nEnter Student roll no.:\t");
scanf("%d", &s.roll);
display(s);
return 0;
}

//Function to be called
void display(struct Student stu)
{
    printf("\nStudent name is %s\n", stu.name);
    printf("\nRoll number of %s is %d \n", stu.name, stu.roll);
}

```

OUTPUT

```

Enter Student record:
Student name: Sam
Enter Student roll no.: 10
Student name is Sam
Roll number of Sam is 10

```

4.3.1.6 Nested Structures

Nesting of structures, is also permitted in C language. Nested structures mean, that one structure has another structure as member **variable**.

Example 20: In case **Dateofbirth structure is defined inside **Student** structure**

```

#include<stdio.h>
struct Student
{
    char name[10];
    int roll;
    struct Dateofbirth
    {
        int date;
        int month;
        int year;
    }
}
```

```

}dob;
};

void display(struct Student st);

int main()
{
    struct Student s;
    printf("\nEnter Student record:\n");
    printf("\nStudent name:\t");
    scanf("%s", s.name);
    printf("\nEnter Student roll no:\t");
    scanf("%d", &s.roll);
    printf("\nEnter Student DOB date:\t");
    scanf("%d", &s.dob.date);
    printf("\nEnter Student DOB month:\t");
    scanf("%d", &s.dob.month);
    printf("\nEnter Student DOB year:\t");
    scanf("%d", &s.dob.year);
    display(s);
    return 0;
}

//Function to be called

void display(struct Student stu)
{
    printf("\nStudent name is %s\n", stu.name);
    printf("\nRoll number of %s is %d \n", stu.name, stu.roll);
    printf("\nDate of Birth of %s is %d-%d-%d\n", stu.name, stu.dob.date, stu.dob.month, stu.dob.year);
}

```

OUTPUT

```

Enter Student record:
Student name: Sam
Enter Student roll no: 10
Enter Student DOB date: 05
Enter Student DOB month: 07

```



Enter Student DOB year: 1983

Student name is Sam

Roll number of Sam is 10

Date of Birth of Sam is 5-7-1983

Example 21: In case Dateofbirth structure is defined separately outside Student structure

```
#include<stdio.h>
struct Dateofbirth
{
    int date;
    int month;
    int year;
};

struct Student
{
    char name[10];
    int roll;
    struct Dateofbirth dob;
};

void display(struct Student st);
int main()
{
    struct Student s;
    printf("\nEnter Student record:\n");
    printf("\nStudent name:\t");
    scanf("%s", s.name);
    printf("\nEnter Student rollno:\t");
    scanf("%d", &s.roll);
    printf("\nEnter Student DOB date:\t");
    scanf("%d", &s.dob.date);
    printf("\nEnter Student DOB month:\t");
    scanf("%d", &s.dob.month);
    printf("\nEnter Student DOB year:\t");
    scanf("%d", &s.dob.year);
```

```

        display(s);
        return 0;
    }

//Function to be called

void display(struct Student stu)
{
    printf("\nStudent name is %s\n", stu.name);
    printf("\nRoll number of %s is %d \n",stu.name,stu.roll);
    printf("\nDate of Birth of %s is %d-%d-%d
\n",stu.name,stu.dob.date,stu.dob.month,stu.dob.year);
}

```

OUTPUT

```

Enter Student record:
Student name: Sam
Enter Student roll no: 10
Enter Student DOB date: 05
Enter Student DOB month: 07
Enter Student DOB year: 1983
Student name is Sam
Roll number of Sam is 10
Date of Birth of Sam is 5-7-1983

```

4.3.1.7 **typedef** in C

The **typedef** is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax of **typedef**

```
typedef <existing_name> <alias_name>;
```

Example 22: A program to add two numbers normally

```

#include <stdio.h>
int main()

```

```
{
    int x=10;
    int y=20;
    int z;
    z=x+y;
    printf("The sum of %d and %d is %d \n",x,y,z);
    return 0;
}
```

OUTPUT

The sum of 10 and 20 is 30

Now we will apply **typedef** to the above code

Example 23: A program to add two numbers using typedef

```
#include <stdio.h>
int main()
{
    typedef int sample;
    sample x=10;
    sample y=20;
    sample z;
    z=x+y;
    printf("The sum of %d and %d is %d \n",x,y,z);
    return 0;
}
```

OUTPUT

The sum of 10 and 20 is 30

We can see that **int** is replaced by the variable **sample**. Thus we can observed that the **typedef** keyword provides a nice shortcut by providing an alternative name for an already existing variable. This keyword is useful when we are dealing with the long data type especially, structure declarations.

4.3.1.7.1 Structure definition using **typedef**

Let's take a simple code example to understand how we can define a structure in C using **typedef** keyword.

Method 1:**Example 24:**

```
#include<stdio.h>
#include<string.h>
struct Student
{
    char name[25];
    int age;
    char branch[10];
    char gender[10];
};

typedef struct Student Record;
int main()
{
    Record s1;
    s1.age = 18;
    strcpy(s1.name, "john");
    strcpy(s1.branch, "ECE");
    strcpy(s1.gender, "Male");
    printf("Name of Student 1: %s\n", s1.name);
    printf("Gender of Student 1: %s\n", s1.gender);
    printf("Age of Student 1: %d\n", s1.age);
    printf("Branch of Student 1: %s\n", s1.branch);
    return 0;
}
```

OUTPUT

```
Name of Student 1: john
Gender of Student 1: Male
Age of Student 1: 18
Branch of Student 1: ECE
```

Method 2:**Example 25:**

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

```

#include<string.h>
typedef struct Student
{
    char name[25];
    int age;
    char branch[10];
    char gender[10];
}Record;
int main()
{
    Record s1;
    s1.age = 18;
    strcpy(s1.name, "john");
    strcpy(s1.branch, "ECE");
    strcpy(s1.gender, "Male");
    printf("Name of Student 1: %s\n", s1.name);
    printf("Gender of Student 1: %s\n", s1.gender);
    printf("Age of Student 1: %d\n", s1.age);
    printf("Branch of Student 1: %s\n", s1.branch);
    return 0;
}

```

OUTPUT

```

Name of Student 1: john
Gender of Student 1: Male
Age of Student 1: 18
Branch of Student 1: ECE

```

4.3.2 C Union

Like structure, Union in c language is a user-defined data type that is used to store the different type of elements. At once, only one member of the union can occupy the memory.

In other words, we can say that the **size of the union in any instance is equal to the size of its largest element.**

4.3.2.1 Defining union

The **union** keyword is used to define the union.

The syntax to define union in c.

```
union union_name
{
    data_type member1;
    data_type member2;
    .
    data_type memberN;
};
```

Let's see the example to define union for an employee in c.

```
union employee
{
    int id;
    char name[50];
    float salary;
};
```

4.3.2.2 Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

Type1:

```
union car
{
    char name[50];
    int price;
};

int main()
{
    union car car1, car2;
    return 0;
}
```

Example 26: Demonstrating Type 1 union variables creation

```
#include<stdio.h>
#include<string.h>
union car
{
    char name[50];
    int price;
};

int main()
{
    union car car1, car2;
    car1.price = 450000;
    car2.price = 950000;
    strcpy(car1.name, "Alto");
    strcpy(car2.name, "Swift");
    printf("Name of Car 1: %s\n", car1.name);
    printf("Name of Car 2: %s\n", car2.name);
    printf("Price of Car 1: %d\n", car1.price);
    printf("Price of Car 2: %d\n", car2.price);
    return 0;
}
```

OUTPUT

```
Name of Car 1: Alto
Name of Car 2: Swift
Price of Car 1: 1869900865
Price of Car 2: 1718187859
```

Type2:

```
union car
{
    char name[50];
    int price;
} car1, car2;
```

Example 27: Demonstrating Type 2 union variables creation

```
#include<stdio.h>
#include<string.h>
union car
{
    char name[50];
    int price;
} car1,car2;

int main()
{
    car1.price = 450000;
    car2.price = 950000;
    strcpy(car1.name, "Alto");
    strcpy(car2.name, "Swift");
    printf("Name of Car 1: %s\n", car1.name);
    printf("Name of Car 2: %s\n", car2.name);
    printf("Price of Car 1: %d\n", car1.price);
    printf("Price of Car 2: %d\n", car2.price);
    return 0;
}
```

OUTPUT

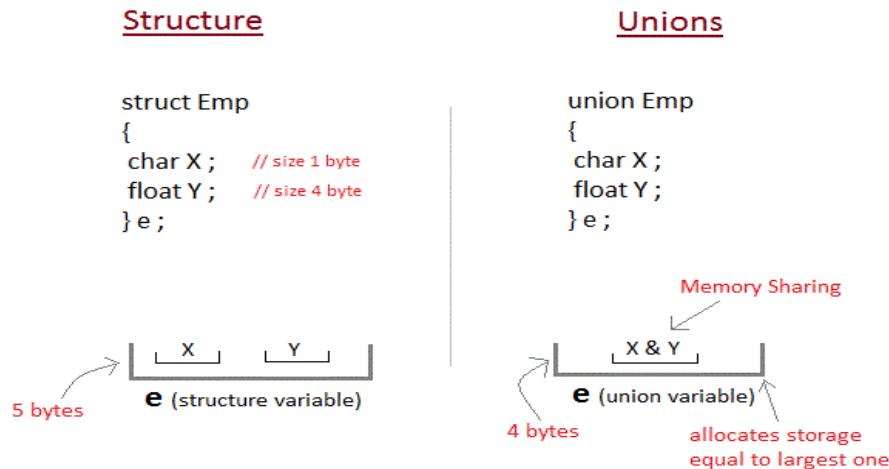
```
Name of Car 1: Alto
Name of Car 2: Swift
Price of Car 1: 1869900865
Price of Car 2: 1718187859
```

Note: Car name is displayed but Car price is a garbage value. This is because name has larger memory size. So only name will have actual value.

4.3.2.3 Difference between structures and unions

Unions are conceptually similar to structures. The syntax to declare/define a union is also similar to that of a structure. The only differences are in terms of storage. In structure each member has its own storage location, whereas all members of union use a single shared

memory location which is equal to the size of its largest data member.



This implies that although a **union** may contain many members of different types, it cannot handle all the members at the same time.

Example 28: To demonstrate the difference between unions and structures

```
#include <stdio.h>
union unionWorker
{
    //defining a union
    char name[32];// 32 bytes
    float salary;//4 bytes
    int workerID;//4 bytes
};

struct structWorker
{
    char name[32];//32 bytes
    float salary;//4 bytes
    int workerID;//4 bytes
};

int main()
{
    union unionWorker uWorker;
    struct structWorker sWorker;
    printf("size of union = %lu bytes", sizeof(uWorker));
    printf("\nsize of structure = %lu bytes", sizeof(sWorker));
}
```

```
    return 0;
}
```

OUTPUT

```
size of union = 32 bytes
size of structure = 40 bytes
```

Why this difference in the size of union and structure variables?

Here, the size of **sWorker** is 40 bytes because

- the size of name[32] is 32 bytes
- the size of salary is 4 bytes
- the size of workerID is 4 bytes

However, the size of **uWorker** is 32 bytes. It's because the size of a union variable **will always be the size of its largest element**.

In the above example, the size of its largest element, (name[32]), is 32 bytes.

With a union, all members share **the same memory**.

4.3.2.4 Accessing Union Members

Example 29:

```
#include <stdio.h>

union Worker {
    float salary;
    int workerID;
};

int main()
{
    union Worker record;
    record.salary = 750.50;
    record.workerID = 100;
    printf("Salary = %.1f\n", record.salary);
    printf("ID Number of worker = %d\n", record.workerID);
    return 0;
}
```

OUTPUT

```
Salary = 0.0
ID Number of worker = 100
```

Note: when `record.workerID` is assigned a value `record.salary` will no longer hold 750.50 .

This is because only the last entered data can be stored in the union. It overwrites the data previously stored in the union.

4.3.2.5 Comparing Structures and unions:

Structure	Union
You can use a struct keyword to define a structure.	You can use a union keyword to define a union.
Every member within structure is assigned a unique memory location.	In union, a memory location is shared by all the data members.
Changing the value of one data member will not affect other data members in structure.	Changing the value of one data member will change the value of other data members in union.
It enables you to initialize several members at once.	It enables you to initialize only the first member of union.
The total size of the structure is the sum of the size of every data member.	The total size of the union is the size of the largest data member.
It is mainly used for storing various data types.	It is mainly used for storing one of the many data types that are available.
It occupies space for each and every member written in inner parameters.	It occupies space for a member having the highest size written in inner parameters.
You can retrieve any member at a time.	You can access one member at a time in the union.
It supports flexible array.	It does not support a flexible array.

Advantages of structure

Here are pros/benefits for using structure:

- Structures gather more than one piece of data about the same subject together in the same place.
- It is helpful when you want to gather the data of similar data types and parameters like first name, last name, etc.
- It is very easy to maintain as we can represent the whole record by using a single name.
- In structure, we can pass complete set of records to any function using a single parameter.
- You can use an array of structure to store more records with similar types.

Advantages of union

Here, are pros/benefits for using union:

- It occupies less memory compared to structure.
- When you use union, only the last variable can be directly accessed.
- Union is used when you have to use the same memory location for two or more data members.
- It enables you to hold data of only one data member.
- Its allocated space is equal to maximum size of the data member.

Disadvantages of structure

Here are cons/drawbacks for using structure:

- If the complexity of IT project goes beyond the limit, it becomes hard to manage.
- Change of one data structure in a code necessitates changes at many other places. Therefore, the changes become hard to track.
- Structure is slower because it requires storage space for all the data.
- You can retrieve any member at a time in structure whereas you can access one member at a time in the union.
- Structure occupies space for each and every member written in inner parameters while union occupies space for a member having the highest size written in inner parameters.
- Structure supports flexible array. Union does not support a flexible array.

Disadvantages of union

Here, are cons/drawbacks for using union:

- You can use only one union member at a time.
- All the union variables cannot be initialized or used with varying values at a time.
- Union assigns one common storage space for all its members.

4.3.3 Storage Classes in C

What is Storage Class in C?

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class in C is used to describe the following things:

- The variable scope.
- The location where the variable will be stored.
- The initialized value of a variable.
- A lifetime of a variable.
- Who can access a variable?

Thus, a storage class is used to represent the information about a variable.

NOTE: A variable value is not only associated with a data type, but also with a storage class. There are total four types of standard storage classes. The table below represents the storage classes in C.

Storage class	Purpose
auto	It is a default storage class or local variable.
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 which is stored inside a Register.

4.3.3.1 The auto Storage Class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

Example, auto int age;

Example 30: Demonstrating auto Storage Class

```
#include <stdio.h>
int main()
{
    auto int j = 1;
    {
        auto int j= 2;
        {
            auto int j = 3;
            printf ("%d\n", j);
        }
        printf ("%d\n",j);
    }
}
```

```
    printf("%d\n", j);
}
```

OUTPUT

```
3
2
1
```

4.3.3.2 The extern Storage Class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized as it has already been defined in the original file.

Example, `extern void display();`

Example 31: Demonstrating extern Storage Class

First File: main.c

```
#include <stdio.h>
extern i;
main() {
    printf("value of the external integer is = %d\n", i);
    return 0;
}
```

Second File: original.c

```
#include <stdio.h>
i=48;
```

OUTPUT

```
value of the external integer is = 48
```

Global variables are accessible throughout the file whereas static variables are accessible only to the particular part of a code.

4.3.3.3 The static Storage Class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

- Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
- Static global variables are global variables visible **only to the file in which it is declared.**

Example: static int count = 10;

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

Example 32: Demonstrating static Storage Class

```
#include <stdio.h> /* function declaration */  
  
void next();  
  
static int counter = 7; /* global variable */  
  
int main()  
{  
    while(counter<10)  
    {  
        next();  
        counter++;  
    }  
    return 0;  
}  
  
void next()  
{  
    static int iteration = 13; /* local static variable */  
    iteration++;  
    printf("iteration=%d and counter= %d\n", iteration, counter);  
}
```

OUTPUT

```
iteration=14 and counter= 7  
iteration=15 and counter= 8  
iteration=16 and counter= 9
```

4.3.3.4 The register Storage Class

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

Example: `register int age;`

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

Example 33: Demonstrating register Storage Class

```
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a); //Trying to access the address of a register variable.
}
```

OUTPUT

```
sample25.c:5:13: error: address of register variable requested
printf("%u",&a); //Trying to access the address of a register variable.
                  ^
1 error generated.
```

4.3.4 Scope and life time of variables

Scope

The area where a variable can be accessed is known as scope of variable.

Lifetime

The time period for which a variable exists in the memory is known as lifetime of variable.

1.1 Scope of Automatic/local variable

Local variable can be used only in the function in which it is declared.

1.2 Lifetime of Automatic/local variable

Lifetime of local variables starts when control enters the function in which it is declared and it is destroyed when control exists from the function.

2.1 Scope of Global variable

These variables are globally accessed from any part of the program. They are declared before main function.

2.2 Lifetime of Global variable

Global variables exist in the memory as long as the program is running. These variables are destroyed from the memory when the program terminates. These variables occupy memory longer than local variables.

3.1 Scope of Static variable

a) **Static (local):** Can be used only in the function in which it is declared.

b) **Static (global):** Can be globally accessed from any part of the program.

3.2 Lifetime of Static variable

a) **Static (local):** Exist in the memory as long as the program is running.

b) **Static (global):** Exist in the memory as long as the program is running.

4.1 Scope of register variable

Register variables can be used only in the function in which it is declared.

4.2 Lifetime of register variable

Lifetime of register variables starts when control enters the function in which it is declared and it is destroyed when control exists from the function.

4.3.5 Simple programs using functions

Program1: Print Prime Numbers from an Integer X to an integer Y

```
#include <stdio.h>
int primechecker(int n);
int main() {
    int n1, n2, i, check;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("Prime numbers from %d to %d are: ", n1, n2);
    for (i = n1; i<=n2; i++) {
        check = primechecker(i);
        if (check == 0)
            printf(" %d ", i);
    }
    return 0;
}
// user-defined function
```

```

int primechecker(int n) {
    int j, flag = 0;
    for (j = 2; j <= n-1; j++)
    {
        if (n % j == 0)
        {
            flag = 1;
            break;
        }
    }
    return flag;
}

```

OUTPUT

Enter two positive integers: 19 79

Prime numbers from 19 to 79 are: 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79

Program2: Sum of Natural Numbers Using Recursion

```

#include <stdio.h>
int summing(int n);
int main() {
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    printf("Sum of %d natural numbers is = %d \n", num, summing(num));
    return 0;
}
//User-Defined function
int summing(int n)
{
    int sum=0;
    if(n != 0)
    {
        sum = n + summing(n - 1);
    }
}

```

```

        return sum;
    }
else
    return n;
}

```

OUTPUT

```

Enter a positive integer: 10
Sum of 10 natural numbers is = 55

```

Program3: Check whether a given Integer can be expressed as a Sum of Two Prime Numbers

```

#include <stdio.h>
int primechecker(int n);
int main()
{
    int input,i,check=0;
    printf("Enter a positive integers: ");
    scanf("%d", &input);
    printf("\n");
    for (i = 2; i<=input/2; i++)
    {
        if (primechecker(i)==0)
        {
            if (primechecker(input-i)==0)
            {
                check = 1;
                printf("%d Can be expressed as a sum of %d and %d \n",input,i,input-i);
            }
        }
    }
    if (check == 0)
    {
        printf("%d cannot be expressed as the sum of two prime numbers.\n",input);
    }
}

```

```

        return 0;
    }

// user-defined function
int primechecker(int n)
{
    int j, flag = 0;
    for (j = 2; j <= n/2; j++)
    {
        if (n % j == 0)
        {
            flag = 1;
            break;
        }
    }
    return flag;
}

```

OUTPUT

```

Enter a positive integer: 46
46 Can be expressed as a sum of 3 and 43
46 Can be expressed as a sum of 5 and 41
46 Can be expressed as a sum of 17 and 29
46 Can be expressed as a sum of 23 and 23

```

Program 4: Reverse a sentence using recursion

```

#include <stdio.h>
void reverseSentence();
int main() {
    printf("Enter a sentence: ");
    reverseSentence();
    printf("\n ");
    return 0;
}
//user defined function

```

```

void reverseSentence() {
    char c;
    scanf("%c", &c);
    if (c != '\n') {
        reverseSentence();
        printf("%c", c);
    }
}

```

OUTPUT

```

Enter a sentence: hai everyone
enoyreve iah

```

Program 5: Find Root of the quadratic equation $cx^2 + bx + a = 0$

```

#include<stdio.h>
#include<math.h>
float a,b,c,D,r1,r2,ip;
void quadraticsolution(float D,float b,float a);
int main()
{
    printf("Enter the variables c, b and a: ");
    scanf("%f%f%f",&c,&b,&a);
    D = b*b - 4*a*c;
    printf("The value of Discriminent is %.2f \n",D);
    quadraticsolution(D,b,a);
    return 0;
}
//user defined function
void quadraticsolution(float D,float b,float a)
{
    if (D>=0)
    {
        r1=(-b + sqrt(D))/(2 * a);
        r2=(-b - sqrt(D))/(2 * a);
    }
}

```

```

        printf(" The roots are %.2f and %.2f \n",r1,r2);
    }
else
{
    rp= -b / (2 * a);
    ip= sqrt(-D)/(2 * a);
    printf(" The roots are :\n");
    printf(" %2.f + j%2.f \n",rp,ip);
    printf(" %2.f - j%2.f \n",rp,ip);
}
}

```

OUTPUT

```

Enter the variables c, b and a: 3 4 1
The value of Discriminent is 4.00
The roots are -1.00 and -3.00
Enter the variables c, b and a: 4 3 1
The value of Discriminent is -7.00
The roots are:
-2 + j1.32
-1.50 - j1.32

```

Program 6: Program to calculate power using recursion

```

#include <stdio.h>
int power(int n1, int n2);
int main() {
    int base, a, result;
    printf("Enter base number: ");
    scanf("%d", &base);
    printf("Enter power number: ");
    scanf("%d", &a);
    result = power(base, a);
    printf("%d^%d = %d \n", base, a, result);
}

```

```

        return 0;
    }

// user-defined function
int power(int base, int a)
{
    if (a != 0)
        return (base * power(base, a - 1));
    else
        return 1;
}

```

OUTPUT

```
Enter base number: 2
```

```
Enter power number: 5
```

```
2^5 = 32
```

Program 7: Swapping numbers using Function Call by Value

```

#include <stdio.h>
void swapnum( int var1, int var2 );
int main()
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);
    swapnum(num1, num2);
    return 0;
}
void swapnum( int var1, int var2 )
{
    int temp;
    temp = var1 ;
    var1 = var2 ;
    var2 = temp;
    printf("\n After swapping: %d, %d \n", var1, var2);
}

```

OUTPUT

Before swapping: 35, 45

After swapping: 45, 35

Program 8: Finding largest of three numbers A, B and C given.

```
#include<stdio.h>

int largestnum(int a, int b ,int c);

int main()
{
    int a, b,c,result;
    printf("Enter 3 numbers A,B,C for comparison: ");
    scanf("%d%d%d",&a,&b,&c);
    result = largestnum(a,b,c);
    printf("The greater number is: %d \n", result);
    return 0;
}

// User defined function

int largestnum(int a, int b, int c)
{
    if((a > b)&&(a>c))
    {
        return a;
    }
    else if (b>c)
    {
        return b;
    }
    else
    {
        return c;
    }
}
```

OUTPUT

```
Enter 3 numbers A, B, C for comparison: 20 30 10
```

```
The greater number is: 30
```

Program 9: Find the Absolute Value of a Number by Defining a User-defined Function

```
#include<stdio.h>

float absolute (int n);

int main()
{
    int num, result;
    printf("Enter a number: ");
    scanf("%d", &num);
    result = absolute(num);
    printf("Absolute value of %d = %d \n", num, result);
    return 0;
}

// User defined function

float absolute (int n)
{
    if(n<0)
    {
        return -n;
    }
    else
    {
        return n;
    }
}
```

OUTPUT

```
Enter a number: -10
```

```
Absolute value of -10 = 10
```

Program 10: Find the Sum of Last Two Digits of an Integer Number using multiple functions

```
#include<stdio.h>
int addTwoDigits(int n);
int lastDigit(int n);
int secondLastDigit(int n);
int main()
{
    int sum = 0,number;
    printf("Enter an integer number: ");
    scanf("%d",&number);
    sum = addTwoDigits(number);
    printf("Sum of last two digits is = %d\n",sum);
    return 0;
}
// User defined function 1
int addTwoDigits(int n)
{
    int result =0;
    result = lastDigit(n) + secondLastDigit(n);
    return result;
}
// User defined function 2
int lastDigit(int n)
{
    return (n%10);
}
// User defined function 3
int secondLastDigit(int n)
{
    return ((n/10)%10);
}
```

OUTPUT

Enter an integer number: 2345

Sum of last two digits is = 9

Program 11: Write a function namely myFact in C to find the factorial of a given number. Also, write another function in C namely nCr which accepts two positive integer parameters n and r and returns the value of the mathematical function $C(n,r)$ ($n! / (r! \times (n - r)!)$). The function nCr is expected to make use of the factorial function myFact.

```
#include <stdio.h>
double myFact(int);
double ncr(int, int);
int main(){
    int n, r;
    double ncr1, npr1;
    printf("Enter the value of n and r\n");
    scanf("%d%d", &n, &r);
    ncr1 = ncr(n, r);
    printf("%dC%d = %.1f\n", n, r, ncr1);
    return 0;
}
double ncr(int n, int r) {
    double result;
    result = myFact(n)/(myFact(r)*myFact(n-r));
    return result;
}
double myFact(int n) {
    int c;
    double result = 1;
    for (c = 1; c <= n; c++)
        result = result*c;
    return result;
}
```

OUTPUT

Enter the value of n and r

53

$5C3 = 10.0$

Program 12: Write a function namely myFact in C to find the factorial of a given number. Also, write another function in C namely nCr and nPr which accepts two positive integer parameters n and r and returns the value of the mathematical function $P(n,r)n!/(n-r)!$ and $C(n,r) (n! / (r! x (n - r)!))$. The function nCr and nPr are expected to make use of the factorial function myFact.

```
#include <stdio.h>
double myFact(int);
double ncr(int, int);
double npr(int, int);
int main(){
    int n, r;
    double ncr1, npr1;
    printf("Enter the value of n and r\n");
    scanf("%d%d", &n, &r);
    ncr1 = ncr(n, r);
    npr1 = npr(n, r);
    printf("%dC%d = %.1f\n", n, r, ncr1);
    printf("%dP%d = %.1f\n", n, r, npr1);
    return 0;
}
double ncr(int n, int r) {
    double result;
    result = myFact(n)/(myFact(r)*myFact(n-r));
    return result;
}
double npr(int n, int r) {
    double result;
    result = myFact(n)/myFact(n-r);
    return result;
}
double myFact(int n) {
```

```
int c;  
double result = 1;  
for (c = 1; c <= n; c++)  
    result = result*c;  
return result;  
}
```

OUTPUT

```
Enter the value of n and r
```

```
5 3
```

```
5C3 = 10.0
```

```
5P3 = 60.0
```



THIS PAGE IS INTENTIONALLY LEFT BLANK

MODULE-V

Pointers and Files	
5.1	Basics of Pointer: declaring pointers, accessing data through pointers, NULL pointer, array access using pointers, pass by reference effect
5.2	File Operations: open, close, read, write, append
5.3	Sequential access and random access to files: In built file handling functions (<i>rewind()</i> , <i>fseek()</i> , <i>ftell()</i> , <i>feof()</i> , <i>fread()</i> , <i>fwrite()</i>), <i>simple programs covering pointers and files.</i>

5.1 Basics of Pointer

5.1.1 What is Pointer in C Programming?

The pointer in C language is **a variable** which stores the **address of another variable**. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 bytes. Consider the following example to define a pointer which stores the address of an integer.

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

A few important points to remember:

- ***** is used to access the value stored in the pointer variable.
- **&** is used to store the address of a given variable.

5.1.2 Declaring a pointer

5.1.2.1 Pointer to integer, double, character, float

The pointer in C language can be declared using ***** (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

```
int *p1 /*Pointer to an integer variable*/  
double *p2 /*Pointer to a variable of data type double*/  
char *p3 /*Pointer to a character variable*/  
float *p4 /*pointer to a float variable*/
```

You can also declare pointers in these ways.

```
int *p1;  
int * p2;
```

5.1.2.2 Pointer to array

```
int arr[10];  
int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.
```

5.1.2.3 Pointer to a function

```
void show (int);  
void(*p)(int) = &display; // Pointer p is pointing to the address of a function
```

5.1.3 Advantages of pointer

- 1) Pointer **reduces the code and improves the performance**; it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
- 2) We can **return multiple values from a function** using the pointer.
- 3) It makes you able to **access any memory location** in the computer's memory.

5.1.4 Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

5.1.5 Address Of (&) Operator

The address of operator '&' returns the address of a variable. But we need to use %u to display the address of a variable.

Example 1: Program to print the value and address of a variable

```
#include<stdio.h>  
int main(){  
int number=50;  
printf("value of number is %d, address of number is %u",number,&number);  
return 0;  
}
```

OUTPUT

```
value of number is 50, address of number is 3781690888
```

5.1.6 NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Example 2: Demonstrating NULL pointer

```
#include<stdio.h>

int main(){
    int *p = NULL;
    printf("value is %d",p);
    return 0;
}
```

OUTPUT

```
value is 0
```

5.1.7 Working of Pointers

5.1.7.1 Pointer Example: Assigning addresses to Pointers

Example 3:

```
#include<stdio.h>

int main()
{
    int* pc, c;
    c = 5;
    pc = &c;
    printf("The value of c is %d \n", c);
    printf("The value of pc is %u which is the address \n", pc);
    return 0;
}
```

OUTPUT

```
The value of c is 5
```

```
The value of pc is 3775546876 which is the address
```

5.1.7.2 Pointer Example: Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the * operator.

Example 4:

```
#include<stdio.h>

int main()
{
int* pc, c;
c = 5;
pc = &c;
printf("The value of *pc is %d \n", *pc);
return 0;
}
```

OUTPUT

```
The value of *pc is 5
```

Here, the address of c is assigned to the pc pointer. To get the value stored in that address, we used *pc.

5.1.7.3 Pointer Example: Changing Value Pointed by Pointers

Example 5:

```
#include<stdio.h>

int main()
{
int* pc, c;
c = 5;
pc = &c;
c = 1;
printf("The value of *pc is %d \n", *pc);
printf("The value of c is %d \n", c);
return 0;
}
```

OUTPUT

```
The value of *pc is 1
```

```
The value of c is 1
```

We have assigned the address of c to the pc pointer. Then, we changed the value of c to 1.

Since pc and the address of c is the same, *pc gives us 1.

Example 6:

```
#include<stdio.h>
int main()
{
int* pc, c, d;
c = 5;
pc = &c;
printf("The value of *pc is %d \n", *pc);
d = -15;
pc = &d;
printf("The value of *pc is %d \n", *pc);
return 0;
}
```

OUTPUT

```
The value of *pc is 5
The value of *pc is -15
```

5.1.7.4 Pointer Example: Using pointers to print the address and value.**Example 7:**

```
#include<stdio.h>
int main(){
int number=100;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // printing p gives the address of number.
printf("Value of p variable is %d \n",*p); // if we print *p, we will get the value stored at the
address contained by p.
return 0;
}
```

OUTPUT

```
Address of p variable is e31d1608
Value of *p variable is 50
```

5.1.7.5 Pointer Example: Working of Pointers

Example 8:

```
#include <stdio.h>

int main()
{
    int* pc, c;
    c = 22;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c);
    pc = &c;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);
    c = 11;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);
    *pc = 2;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c);
    return 0;
}
```

OUTPUT

```
Address of c: 0x7ffee1fc85fc
Value of c: 22
Address of pointer pc: 0x7ffee1fc85fc
Content of pointer pc: 22
Address of pointer pc: 0x7ffee1fc85fc
Content of pointer pc: 11
Address of c: 0x7ffee1fc85fc
Value of c: 2
```

5.1.8 Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c. Then,

```
int c, *pc;
// pc is address but c is not
```



```

pc = c; // Error
// &c is address but *pc is not
*pc = &c; // Error
// both &c and pc are addresses
pc = &c;
// both c and *pc values
*pc = c;

```

5.1.9 Arrays and Pointers

In order to know the usage of arrays with the pointers we need to first know the relationship between them.

5.1.9.1 Relationship Between Arrays and Pointers

Example 9:

```

#include <stdio.h>
int main()
{
    int x[4]={1,2,3,4};
    int i;
    for(i = 0; i < 4; i++)
    {
        printf("&x[%d] = %p\n", i, &x[i]);
    }
    printf("Address of array x: %p \n", x);
    printf("Value of x[0]: %d \n", x[0]);
    printf("Value of *x : %d \n", *x);
    return 0;
}

```

OUTPUT

```

&x[0] = 0x7ffee033b5f0
&x[1] = 0x7ffee033b5f4
&x[2] = 0x7ffee033b5f8
&x[3] = 0x7ffee033b5fc
Address of array x: 0x7ffee033b5f0
Value of x[0]: 1

```

Value of `*x` : 1

Note:

1. In the output there is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler).
2. Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.
3. It is clear from the example that `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.
4. Similarly,
 - `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
 - `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
 - `&x[3]` is equivalent to `x+3` and `x[3]` is equivalent to `*(x+3)`.

Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

Example 10:

This example gives us a detailing about, how pointers can be used to read and sum the elements of an array.

```
#include <stdio.h>
int main()
{
    int i, x[4], sum = 0;
    printf("Enter 4 numbers: \n");
    for(i = 0; i < 4; i++) {
        scanf("%d", x+i); // Equivalent to scanf("%d", &x[i]);
        sum = sum + (*x+i); // Equivalent to sum += x[i]
    }
    printf("Sum of the array elements entered = %d \n", sum);
    return 0;
}
```

OUTPUT

Enter 4 numbers:

23

34

45

67

Sum of the array elements entered = 169

Here, we have declared an array x of 4 elements. To access these elements, we used pointers.

Example 11:

This example gives us a detailing about, how pointers can be used to read and display the elements of an array.

```
#include <stdio.h>
int main()
{
    int i, x[4], sum = 0;
    printf("Enter 4 numbers: \n");
    for(i = 0; i < 4; i++) {
        // Equivalent to scanf("%d", &x[i]);
        scanf("%d", x+i);
    }
    printf("Print the entered elements of an array \n");
    for(i = 0; i < 4; i++) {
        // Equivalent to printf("%d",x[i]);
        printf("%d \n", *(x+i));
    }
    return 0;
}
```

OUTPUT

Enter 4 numbers:

12

34

56

78

Print the entered elements of an array

12

34

56

78

5.1.9.2 Pointer to an Array in C

Pointer to an array is also known as array pointer. We are using the pointer to access the components of the array.

```
int a[3] = {3, 4, 5 };
int *ptr = a;
```

In the above code we have a pointer ptr that focuses to the **0th** component of the array that is **3**. We can likewise declare a pointer that can point to whole array rather than just a single component of the array.

Syntax:

data type (*var name)[size of array];

Declaration of the pointer to an array:

```
int b[5] = { 1, 2, 3, 4, 5 };
int (* ptr)[5] = &b;
```

Example 12: To demonstrate pointer to an array.

```
#include <stdio.h>

int main()
{
    int b[5] = { 1, 2, 3, 4, 5 };
    int(*a)[5];
    int i = 0;
    // Points to the whole array b
    a = &b;
    for (i = 0; i < 5; i++)
        printf("%d\n",*(a+i));
    return 0;
}
```

OUTPUT

```
1
2
3
4
5
```

5.1.9.3 Arrays of Pointers in C

“Array of pointers” is an array of the pointer variables. It is also known as pointer arrays.

Syntax:

```
int *var_name[array_size];
```

Declaration of an array of pointers:

```
int *ptr[3];
```

Example 13: To demonstrate pointer to an array.

```
#include <stdio.h>
const int SIZE = 3;
int main()
{
    int arr[] = { 1, 2, 3 };
    int i, *ptr[SIZE];
    for (i = 0; i < SIZE; i++)
    {
        ptr[i] = &arr[i];
    }
    for (i = 0; i < SIZE; i++)
    {
        printf("Value of arr[%d] = %d\n", i, *ptr[i]);
    }
    return 0;
}
```

OUTPUT

```
Value of arr[0] = 1
Value of arr[1] = 2
Value of arr[2] = 3
```

5.1.9.4 Sample programs demonstrating Working of Pointers

Example 14: Demonstrates how pointers can be used to access the elements of an array.

```
#include <stdio.h>
int main() {
    int x[5] = {1, 2, 3, 4, 5};
```

```

int* ptr;
// ptr is assigned the address of the third element
ptr = &x[2];
printf("*ptr-2 = %d\n", *(ptr-2));
printf("*ptr-1 = %d\n", *(ptr-1));
printf("*ptr = %d\n", *ptr);
printf("*ptr+1 = %d\n", *(ptr+1));
printf("*ptr+2 = %d\n", *(ptr+2));
return 0;
}

```

OUTPUT

```

*(ptr-2) = 1
*(ptr-1) = 2
*ptr = 3
*(ptr+1) = 4
*(ptr+2) = 5

```

Example 15: Demonstrates how pointers can be used to access the elements of an array and their address.

```

#include<stdio.h>
int main()
{
    int arr[5] = {100, 200, 300, 400, 500}, i;
    int *ptr = arr;
    for(i = 0; i < 5; i++)
        printf("&arr[%d] = %p\t arr[%d] = %d\n", i, ptr+i, i, *(ptr+i));
    return 0;
}

```

OUTPUT

&arr[0] = 0x7ffee35fc5f0	arr[0] = 100
&arr[1] = 0x7ffee35fc5f4	arr[1] = 200
&arr[2] = 0x7ffee35fc5f8	arr[2] = 300
&arr[3] = 0x7ffee35fc5fc	arr[3] = 400
&arr[4] = 0x7ffee35fc600	arr[4] = 500

Example 16: Gives us a detailing about modifying array elements using the pointer.

```
#include<stdio.h>
int main()
{
    int arr[5] = {100, 200, 300, 400, 500}, i;
    int *ptr = arr;
    printf("The array elements Before modification \n");
    for(i = 0; i < 5; i++)
    {
        printf("arr[%d] = %d\n", i, *(ptr+i));
    }
    //changing 3rd element(300) as 1000.
    *(ptr+2) = 1000;
    printf("The array elements After modification \n");
    for(i = 0; i < 5; i++)
    {
        printf("arr[%d] = %d\n", i, *(ptr+i));
    }
    return 0;
}
```

OUTPUT

The array elements Before modification

```
arr[0] = 100
arr[1] = 200
arr[2] = 300
arr[3] = 400
arr[4] = 500
```

The array elements After modification

```
arr[0] = 100
arr[1] = 200
arr[2] = 1000
arr[3] = 400
arr[4] = 500
```

Example 17: Gives us a detailing about swapping the array elements using the pointer.

```
#include<stdio.h>
int main()
{
    int arr[5] = {100, 200, 300, 400, 500}, i;
    int *ptr = arr;
    printf("The array elements Before Swapping \n");
    for(i = 0; i < 5; i++)
    {
        printf("arr[%d] = %d\n", i, *(ptr+i));
    }
    //Swapping with a third variable
    *(ptr+5) = *(ptr+2);
    *(ptr+2) = *(ptr+3);
    *(ptr+3) = *(ptr+5);
    printf("The array elements After Swapping \n");
    for(i = 0; i < 5; i++)
    {
        printf("arr[%d] = %d\n", i, *(ptr+i));
    }
    return 0;
}
```

OUTPUT

The array elements Before Swapping

```
arr[0] = 100
arr[1] = 200
arr[2] = 300
arr[3] = 400
arr[4] = 500
```

The array elements After Swapping

```
arr[0] = 100
arr[1] = 200
arr[2] = 400
arr[3] = 300
```

```
arr[4] = 500
```

Example 18: Gives us a detailing about swapping the two array elements using the pointer without a third variable.

```
#include<stdio.h>
int main()
{
    int arr[5] = {100, 200, 300, 400, 500}, i;
    int *ptr = arr;
    printf("The array elements Before Swapping \n");
    for(i = 0; i < 5; i++)
    {
        printf("arr[%d] = %d\n", i, *(ptr+i));
    }
    //Swapping without a third variable
    *(ptr+2) = *(ptr+2)+ *(ptr+3);
    *(ptr+3) = *(ptr+2)- *(ptr+3);
    *(ptr+2) = *(ptr+2)- *(ptr+3);
    printf("The array elements After Swapping \n");
    for(i = 0; i < 5; i++)
    {
        printf("arr[%d] = %d\n", i, *(ptr+i));
    }
    return 0;
}
```

OUTPUT

The array elements Before Swapping

```
arr[0] = 100
arr[1] = 200
arr[2] = 300
arr[3] = 400
arr[4] = 500
```

The array elements After Swapping

```
arr[0] = 100
arr[1] = 200
arr[2] = 400
arr[3] = 300
arr[4] = 500
```

5.1.10 Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. 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 19: Basic example to add 100 to a number using call by reference.

```
#include<stdio.h>
void change(int *num);
int main()
{
    int x=100;
    printf("Before function call x = %d \n", x);
    change(&x);//passing reference in function
    printf("After function call x = %d \n", x);
    return 0;
}
void change(int *num)
{
    printf("Before adding value inside function num=%d \n", *num);
    *num = (*num) + 100;
    printf("After adding value inside function num=%d \n", *num);
}
```

OUTPUT

```
Before function call x = 100
```

Before adding value inside function num=100

After adding value inside function num=200

After function call x = 200

Example 20: An example to swap values of two variables using call by reference.

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values a = %d, b = %d\n",a,b);
    swap(&a,&b);
    return 0;
}
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in function a = %d, b = %d\n",*a,*b);
}
```

OUTPUT

Before swapping the values a = 10, b = 20

After swapping values in function a = 20, b = 10

5.1.11 Dynamic memory allocation in C

An array is basically a collection of fixed number of values because of which once declared, cannot be changed.

But sometimes the size of the array declared may not be sufficient. In such cases allocating memory manually during run-time can be a practical solution and is known as dynamic memory allocation in C programming.

Allocation of memory dynamically can be done by library functions called **malloc()**, **calloc()**, **realloc()** and **free()** , which are defined in the **<stdlib.h>** header file.

5.1.11.1 malloc()

The name "malloc" stands for memory allocation.

The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example:

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes.

And, the pointer ptr holds the address of the first byte in the allocated memory.

Note: The expression results in a NULL pointer if the memory cannot be allocated.

5.1.11.2 calloc()

The name "calloc" stands for contiguous allocation.

The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*)calloc(20, sizeof(float));
```

The above statement allocates contiguous space in memory for 20 elements of type float.

5.1.11.3 realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, ptr is reallocated with a new size x.

5.1.11.4 free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Example 21: malloc() and free()

```
//Program to calculate the sum of n numbers entered by the user
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) malloc(n * sizeof(int));
    if(ptr == NULL)      // if memory cannot be allocated
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter the elements:\n");
    for(i = 0; i < n; i++)
    {
        scanf("%d", ptr + i);
        sum = sum + *(ptr + i);
    }
    printf("Sum of entered elements is = %d\n", sum);
    free(ptr); // deallocating the memory
    return 0;
}
```

OUTPUT

```
Enter number of elements: 5
```

```
Enter the elements:
```

```
1
2
4
5
8
```

```
Sum of entered elements is = 20
```

Example 22: calloc() and free()

```
//Program to calculate the sum of n numbers entered by the user
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL) // if memory cannot be allocated
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter the elements:\n");
    for(i = 0; i < n; i++)
    {
        scanf("%d", ptr + i);
        sum = sum + *(ptr + i);
    }
    printf("Sum of entered elements is = %d\n", sum);
    free(ptr); // deallocating the memory
    return 0;
}
```

OUTPUT

```
Enter number of elements: 5
```

```
Enter the elements:
```

```
23
```

```
45
```

```
34
```

```
67
```

```
21
```

```
Sum of entered elements is = 190
```



Example 23: realloc() and free()

```
//Program to allocate and then reallocate the memory
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr, i, n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);
    //Allocating the memory
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Addresses of previously allocated memory:\n");
    for(i = 0; i < n1; i++)
    {
        printf("%p\n", (ptr + i));
    }
    //Enter a new size for reallocation
    printf("\nEnter the new size: ");
    scanf("%d", &n2);
    // Relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));
    printf("Addresses of newly allocated memory: \n");
    for(i = 0; i < n2; i++)
    {
        printf("%p\n", (ptr + i));
    }
    free(ptr);
    return 0;
}
```

OUTPUT

```
Enter size: 3
Addresses of previously allocated memory:/
0x7f80ac4059d0
0x7f80ac4059d4
```

```
0x7f80ac4059d8
Enter the new size: 5
Addresses of newly allocated memory:
0x7f80ac4059d0
0x7f80ac4059d4
0x7f80ac4059d8
0x7f80ac4059dc
0x7f80ac4059e0
```

5.2 File Operations

5.2.1 Introduction to File Handling

Till now we have discussed console I/O functions for taking input from the user and displaying the output. This is ok for a small amount of data, but in real life situations we need to deal with large amounts of the data. In such cases, it is necessary to store the data permanently so that user can access and alter that data if required. Thus, there is a need of files to store large amount of data permanently.

What is a file?

A File is a collection of information in the secondary memory having some file name, stored in the directory.

There are 2 types of files:

a) Text files (ASCII): Text files contain ASCII codes of digits, alphabetic and symbols. The format for this file is **.txt**.

b) Binary files: Binary file contains collection of bytes (0's and 1's). Binary files are compiled version of text files. The format for this file is **.dat**.

5.2.2 C File Operations

The major operations can be performed on file are:

1. Opening a file.
2. Reading data from a file.
3. Writing data in a file.
4. Closing a file.
5. Appending a file.

5.2.2.1 Opening a file: **fopen()**

We must open a file before it can be read, write, or update. The **fopen()** function is used to open a file.

The syntax is:

```
FILE *fopen( const char * filename, const char * mode)
```

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "c://some_folder/some_file.ext".
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

"r" or "rt"	Open a text file for reading, The file must already exist.
"w" or "wt"	Open a text file for writing. If the file already exists, all the data will lose. If it doesn't exist, it will be created.
"a" or "at"	Open a text file for appending. Data will be added at the end of the existing file. If file doesn't exist, it will be created.
rb	Open a binary file for reading, The file must already exist.
wb	Open a binary file for writing. If the file already exists, its contents will be overwritten. If it doesn't exist, it will be created.
ab	Open a binary file for appending. Data will be added at the end of the existing file. If file doesn't exist, it will be created.
"rt+" or "r+t"	Open text file for reading and writing. The file must already exist.
"wt+" or "w+t"	Open text file for reading and writing. If file doesn't exist, it will be created.
"at+" or "a+t"	Open text file for reading and appending. If file doesn't exist, it will be created.
"rb+" or "r+b"	Open binary file for reading and writing. The file must already exist.
"wb+" or "w+b"	Open binary file for reading and writing. If file doesn't exist, it will be created.
"ab+" or "a+b"	Open binary file for reading and appending. If file doesn't exist, it will be created.

The fopen function works in the following way.

- Firstly, it searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

5.2.2.2 Reading a File

To read the file, we must open it first using any of the mode, for example if you only want to read the file then open it in “r” mode. Based on the mode selected during file opening, we are allowed to perform certain operations on the file.

There are three different functions dedicated to reading data from a file:

a) fgetc(file_pointer): It returns the next character from the file pointed to by the file pointer. When the end of the file has been reached, the EOF is sent back.

The syntax is:

```
int fgetc(FILE *stream)
```

Example 24:

```
#include<stdio.h>
int main()
{
FILE *fp;
char c;
fp=fopen("ece1.txt","r");
while((c=fgetc(fp))!=EOF)
{
printf("%c",c);
}
fclose(fp);
return 0;
}
```

OUTPUT

```
J
```

b) fgets(buffer, n, file_pointer): It reads n-1 characters from the file and stores the string in a buffer in which the NULL character '\0' is appended as the last character.

The syntax is:

```
char* fgets(char *s, int n, FILE *stream)
```

Example 25:

```
#include<stdio.h>

int main()
{
FILE *fp;
char *text[100];
fp=fopen("ece2.txt","r");
printf("%s",fgets(text,100,fp));
fclose(fp);
return 0;
}
```

OUTPUT

```
Welcome to SBCE
```

c) fscanf(file pointer, conversion specifiers, variable_addresses): It is used to parse and analyse data. It reads characters from the file and assigns the input to a list of variable pointers variable_addresses using conversion specifiers. Keep in mind that as with scanf, fscanf doesn't stop reading a string when space or newline is encountered.

The syntax is:

```
int fscanf(FILE *stream, const char *format [, argument, ...])
```

Example 26:

```
#include <stdio.h>

int main()
{
FILE *fp;
char buff[100];//creating char array to store data of file
fp = fopen("ece.txt", "r");
while(fscanf(fp, "%s", buff)!=EOF)
{
printf("%s ", buff);
}
fclose(fp);
```



```
    return 0;  
}
```

OUTPUT

```
Welcome to department of ECE
```

5.2.2.3 Writing to a file

To write the file, we must open the file in a mode that supports writing. For example, if you open a file in “r” mode, you won’t be able to write the file as “r” is read only mode that only allows reading.

In C, when you write to a file, newline characters '\n' must be explicitly added.

The stdio library offers the necessary functions to write to a file:

a) fputc(char, file_pointer): It writes a character to the file pointed to by file_pointer.

The syntax is:

```
int fputc(int c, FILE *stream)
```

Example 27:

```
#include <stdio.h>  
  
int main()  
{  
    FILE *fp;  
    fp = fopen("ece1.txt", "w");//opening file  
    fputc('J',fp);//writing single character into file  
    return 0;  
}
```

OUTPUT

```
Will create a ece1.txt file in same folder as the program and it will have the content a single  
character J.
```

b) fputs(str, file_pointer): It writes a string to the file pointed to by file_pointer.

The syntax is :

```
int fputs(const char *s, FILE *stream)
```

Example 28:

```
#include<stdio.h>  
  
int main()  
{  
    FILE *fp;
```



```
fp=fopen("ece2.txt","w");
fputs("Welcome to SBCE",fp);
fclose(fp);
return 0;
}
```

OUTPUT

Will create a ece2.tx file in same folder as the program and it will store the content
Welcome to SBCE

c) **fprintf(file_pointer, str, variable_lists):** It prints a string to the file pointed to by file_pointer. The string can optionally include format specifiers and a list of variables variable_lists.

The syntax is:

```
int fprintf(FILE *stream, const char *format [, argument, ...])
```

Examples 29:

```
#include <stdio.h>
int main()
{
    FILE *fp;
    fp = fopen("ece.txt", "w");//opening file
    fprintf(fp, " Welcome to department of ECE \n");//writing data into file
    fclose(fp);//closing file
    return 0;
}
```

OUTPUT

Will create a ece.txt file in same folder as the program and it will store the content
Welcome to department of ECE

5.2.2.4 Closing a file: **fclose()**

The fclose() function is used to close a file. The file must be closed after performing all the operations on it.

The syntax is:

```
int fclose( FILE *fp)
```

5.2.2.5 Appending a file

For appending file use the append mode ‘a’. To understand further let us consider an example which opens a file in read mode and displays the output using fgets (), then a text is appended to the existing file using append ‘a’ mode. After this the content is again opened in read mode and output is displayed using fgets().

Example 30:

```
#include <stdio.h>
int main()
{
    FILE *fp;
    char ch[100];
    //opening the file before appending
    fp = fopen("ece2.txt", "r");
    printf("The file Before appending: ");
    printf("%s",fgets(ch,100,fp));
    printf("\n");
    fclose(fp);
    // Append mode is been called
    fp = fopen("ece2.txt", "a");
    // Write content to file
    fprintf(fp,"A NACC accredited institution.");
    fclose(fp);
    //opening the file after appending
    fp = fopen("ece2.txt", "r");
    printf("The file After appending: ");
    printf("%s",fgets(ch,100,fp));
    printf("\n");
    fclose(fp);
    return 0;
}
```

OUTPUT

```
Welcome to SBCE.
Welcome to SBCE. A NACC accredited institution.
```

5.2.3 Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

5.2.3.1 Writing to a binary file

To write into a binary file, you need to use the fwrite () function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

The Syntax is:

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 31: Demonstrates Writing to a binary file

```
#include <stdio.h>

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;
    if ((fptr = fopen("ece5.dat","wb"))== NULL)
    {
        printf("Error! opening file");
    }
    for(n = 1; n < 5; n++)
    {
        num.n1 = n;
        num.n2 = 5*n;
        num.n3 = 5*n + n;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
}
```

```
fclose(fptr);
return 0;
}
```

OUTPUT

Will create a ece.dat file and fill store he following

```
n1: 1  n2: 5  n3: 6  n1: 2  n2: 10  n3: 12  n1: 3  n2: 15  n3: 18  n1: 4  n2: 20  n3: 24
```

5.2.3.2 Reading from a binary file

Function fread() also take 4 arguments similar to the fwrite() function as above.

The Syntax is:

```
fread(addressData, sizeData, numbersData, pointerToFile)
```

Example 32: Demonstrates Reading from a binary file

```
#include <stdio.h>

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;
    if ((fptr = fopen("ece5.dat","rb")) == NULL)
    {
        printf("Error! opening file");
    }
    for(n = 1; n < 5; n++)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\t n2: %d\t n3: %d \n", num.n1, num.n2, num.n3);
    }
    fclose(fptr);
    return 0;
}
```



OUTPUT

```
n1: 1  n2: 5  n3: 6
n1: 2  n2: 10 n3: 12
n1: 3  n2: 15 n3: 18
n1: 4  n2: 20 n3: 24
```

5.2.3.3 Appending to a binary file

Example 33: Demonstrates Appending to a binary file

```
#include <stdio.h>
struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    // Printig the values before appending
    printf("Display the values Before Appending\n");
    fptr = fopen("ece1.dat","rb");
    for(n = 1; n < 5; n++)
    {
        fread(&num, sizeof(struct threeNum), 1,fptr);
        printf("n1: %d\t n2: %d\t n3: %d \n", num.n1,num.n2,num.n3);
    }
    fclose(fptr);

    // Appendng the values to the file
    fptr = fopen("ece1.dat","ab");
    for(n = 5; n < 10; n++)
    {
        num.n1 = n;
        num.n2 = 5*n;
        num.n3 = 5*n + n;
    }
}
```

```

fwrite(&num, sizeof(struct threeNum), 1, fptr);
}

fclose(fptr);

// Printig the values after appending
printf("Display the values After Appending\n");
fptr = fopen("eceil.dat", "rb");
for(n = 1; n < 10; n++)
{
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\t n2: %d\t n3: %d \n", num.n1, num.n2, num.n3);
}
fclose(fptr);
return 0;
}

```

OUTPUT

Display the values Before Appending

```

n1: 1  n2: 5  n3: 6
n1: 2  n2: 10 n3: 12
n1: 3  n2: 15 n3: 18
n1: 4  n2: 20 n3: 24

```

Display the values After Appending

```

n1: 1  n2: 5  n3: 6
n1: 2  n2: 10 n3: 12
n1: 3  n2: 15 n3: 18
n1: 4  n2: 20 n3: 24
n1: 5  n2: 25 n3: 30
n1: 6  n2: 30 n3: 36
n1: 7  n2: 35 n3: 42
n1: 8  n2: 40 n3: 48
n1: 9  n2: 45 n3: 54

```

5.3 Sequential access and random access to files

5.3.1 What is Sequential Access?

Sequential Access to a data file means that the computer system reads or writes information to the file sequentially, starting from the beginning of the file and proceeding step by step.

Note: The type of file access discussed in previous topics is sequential access.

5.3.2 What is Random Access?

Random Access to a file means that the computer system can read or write information anywhere in the data file. This type of operation is also called “Direct Access” because the computer system knows where the data is stored (using Indexing) and hence goes “directly” and reads the data.

5.3.3 Advantages of Sequential access and random access to files

Sequential access has advantages when you access information in the same order all the time.

Also is faster than random access.

Random access file on the other hand, has the advantage that you can search through it and find the data you need more easily (using indexing for example).

5.3.4 Random access/Direct access files

In general, we access the data from a file character by character or record by record from the beginning of the file to the end of file called sequential access. If the size of file is small then this takes reasonable time but if a file has 1000's of records this process will take remarkably longer time, which is not acceptable.

Say for example, a file has 10000 records and we have to get 7999th record, if we use sequential access, then we have to go through 7998 records to get the required record.

This can be overcome through directly accessing a record from the required position.

This can be done in two steps that are,

- I. Sending the file active pointer to the required position using fseek() .
- II. Then reading the record using fread() as we learned in the previous session.

5.3.5 C fseek() function

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax is:

```
fseek(FILE *stream, long int offset, int whence)
```

There are 3 constants used in the fseek() function for whence:

Different whence in fseek()	
Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Example 34: Binary file

```
#include <stdio.h>

struct threeNum

{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;
    if ((fptr = fopen("ece1.dat","rb")) == NULL)
    {
        printf("Error! opening file");
    }

    // Moves the cursor to the end of the file
    fseek(fptr,-sizeof(struct threeNum), SEEK_END);
    for(n = 1; n <10; n++)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\nn2: %d\nn3: %d\n", num.n1, num.n2, num.n3);
        fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
    }
    fclose(fptr);
    return 0;
}
```

OUTPUT

```
n1: 9  n2: 45  n3: 54
n1: 8  n2: 40  n3: 48
n1: 7  n2: 35  n3: 42
n1: 6  n2: 30  n3: 36
n1: 5  n2: 25  n3: 30
n1: 4  n2: 20  n3: 24
n1: 3  n2: 15  n3: 18
n1: 2  n2: 10  n3: 12
n1: 1  n2: 5   n3: 6
```

Example 35: Text file

```
#include <stdio.h>
int main()
{
    FILE *fp;
    char ch[100];
    fp = fopen("myfile.txt", "w+");// Creating a file +
    fputs("Welcome to SBCE", fp);
    fseek( fp,11, SEEK_SET);
    fputs("Department of ECE", fp);
    fclose(fp);
    //opening the file after modificaton
    fp = fopen("myfile.txt", "r");
    printf("The file's Content: ");
    printf("%s",fgets(ch,100,fp));
    printf("\n");
    fclose(fp);
    return 0;
}
```

OUTPUT

```
The file's Content: Welcome to Department of ECE
```

5.3.6 C rewind() function

The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax is:

```
void rewind(FILE *stream)
```

Example 36: Demonstrating rewind() function

```
#include<stdio.h>
int main()
{
FILE *fp;
char ch[100];
fp=fopen("ece.txt","r");
printf("%s",fgets(ch,100,fp));
rewind(fp);//moves the file pointer at beginning of the file
printf("%s",fgets(ch,100,fp));
fclose(fp);
return 0;
}
```

OUTPUT

```
Welcome to department of ECE.
```

```
Welcome to department of ECE.
```

5.3.7 C ftell() function

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

Syntax is:

```
long int ftell(FILE *stream) \
```

Example 37: Demonstrating ftell() function

```
#include <stdio.h>
int main ()
{
FILE *fp;
int length;
```



```

fp = fopen("ece.txt", "r");
fseek(fp, 0, SEEK_END);
length = ftell(fp);
fclose(fp);
printf("Size of file: %d bytes \n", length);
return 0;
}

```

OUTPUT

Size of file: 31 bytes

5.3.8 Cfeof() function

The C library function `int feof(FILE *stream)` tests the end-of-file indicator for the given stream.

Syntax is:

```
int feof(FILE *stream)
```

Example 38: Demonstrating feof() function

```

#include <stdio.h>
int main () {
    FILE *fp;
    int c;
    fp = fopen("ece.txt","r");
    while(1) {
        c = fgetc(fp);
        if( feof(fp) )
        {
            break ;
        }
        printf("%c", c);
    }
    fclose(fp);
    return 0;
}

```

OUTPUT

Welcome to department of ECE.

5.3.9 Simple programs covering pointers and files

Example1: C Program to copy contents of one file to other.

```
#include <stdio.h>
#include <stdlib.h> // For exit()
int main()
{
    FILE *fptr1, *fptr2;
    char filename1[100],filename2[100],ch;
    // Open one file for reading
    printf("Enter the file name to open for reading \n");
    scanf("%s", filename1);
    fptr1 = fopen(filename1, "r");
    // Open another file for writing
    printf("Enter the file name to open for writing \n");
    scanf("%s", filename2);
    fptr2 = fopen(filename2, "w");
    if ((fptr1 == NULL)||(fptr2 == NULL))
    {
        printf("Cannot open one of the files \n");
        exit(1);
    }
    // copy contents from file1 to file2
    while ((ch = fgetc(fptr1))!= EOF)
    {
        fputc(ch, fptr2);
    }
    printf("\n Contents copied from %s to %s \n ", filename1,filename2);
    fclose(fptr1);
    fclose(fptr2);
    return 0;
}
```

OUTPUT

```
Enter the file name to open for reading
ece.txt
```

Enter the file name to open for writing

ece4.txt

Contents copied from ece.txt to ece4.txt

Example2: C Program to merge contents of two files into a third file.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fptr1,*fptr2 ,*fptr3;
    char filename1[100],filename2[100],filename3[100],ch;
    // Open two files to be merged
    printf("Enter the file1 name to open for reading: ");
    scanf("%s", filename1);
    fptr1= fopen(filename1, "r");
    printf("Enter the file2 name to open for reading: ");
    scanf("%s", filename2);
    fptr2= fopen(filename2, "r");
    // Open file to store the result
    printf("Enter the file3 name to copy the merged content of both files: ");
    scanf("%s", filename3);
    fptr3 = fopen(filename3, "w");
    //Check for file existence
    if (fptr1 == NULL || fptr2 == NULL || fptr3 == NULL)
    {
        printf("Could not open files");
        exit(1);
    }
    // Copy contents of first file to Third file
    while ((ch = fgetc(fptr1)) != EOF)
    {
        fputc(ch, fptr3);
    }
    // Copy contents of second file to Third file
```

```

while ((ch = fgetc(fptra2)) != EOF)
{
    fputc(ch, fptra3);
}

printf("Merged %s and %s content into %s \n",filename1,filename2,filename3);
fclose(fptra1);
fclose(fptra2);
fclose(fptra3);

return 0;
}

```

OUTPUT

```

Enter the file1 name to open for reading: ece.txt
Enter the file2 name to open for reading: ece1.txt
Enter the file3 name to copy the merged content of both files: ece2.txt
Merged ece.txt and ece1.txt content into ece2.txt

```

Example3: C Program to delete an existing file.

```

#include<stdio.h>

int main()
{
    char filename[100];
    printf("Enter the file name to delete: ");
    scanf("%s", filename);
    if (remove(filename) == 0)
        printf("Deleted file %s successfully \n",filename);
    else
        printf("Unable to delete the file %s \n",filename);
    return 0;
}

```

OUTPUT

```

Enter the file name to delete: ece3.txt
Deleted file ece3.txt successfully
Enter the file name to delete: ece7.txt
Unable to delete the file ece7.txt

```

Example 4: C Program to check the number of lines in a given file.

```
#include <stdio.h>
int main()
{
    FILE *fptr;
    int numlines = 0;
    char filename[100], ch;
    printf("Enter file name to check: ");
    scanf("%s", filename);
    fptr = fopen(filename, "r");
    while ( (ch = getc(fptr))!= EOF)
    {
        //Count whenever ch is \n (new line) is encountered
        if (ch =='\n')
        {
            //increment variable numlines by 1
            numlines=numlines+1;
        }
    }
    fclose(fptr);
    printf("There are %d lines in file %s \n", numlines, filename);
    return 0;
}
```

OUTPUT

```
Enter file name to check: ece2.txt
```

```
There are 5 lines in file ece2.txt
```

Example 5: C Program to delete a specific line from a text file.

```
#include <stdio.h>
int main()
{
    FILE *fptr1, *fptr2;
    char filename[100];
    char ch;
```

```

int deleline, numline = 1;
//asks user for file name
printf("Enter file name: ");
scanf("%s", filename);
//open file in read mode
fptr1 = fopen(filename, "r");
//until the last character of file is obtained
while (( ch = getc(fptr1)) != EOF)
{
    printf("%c", ch);
}
//take fptr1 to start point
rewind(fptr1);
fptr2 = fopen("copy.c", "w");
printf("\n Enter the line number to be deleted:");
scanf("%d", &deleline);
//open new file in write mode
while ((ch = getc(fptr1))!= EOF)
{
    if (ch == '\n')
    {
        numline=numline+1;
    }
    //except the line to be deleted
    if (numline != deleline)
    {
        //copy all lines in file copy.c
        putc(ch, fptr2);
    }
}
fclose(fptr1);
fclose(fptr2);
//remove original file
remove(filename);

```

```

//rename the file copy.c to original name
rename("copy.c", filename);

//Now print the content of file after deletion
printf("\n The contents of file after being modified are as follows:\n");
fptr1 = fopen(filename, "r");
while ((ch = getc(fptr1))!= EOF)

{
    printf("%c", ch);
}
fclose(fptr1);

return 0;
}

```

OUTPUT

Enter file name: ece2.txt

Plastic bags are a major cause of environmental pollution.

Plastic as a substance is non-biodegradable and thus remain in the environment for hundreds of years.

It has become very essential to ban plastic bags before they ruin our planet completely.

Many countries around the globe have either put a ban on the plastic bag or Levi tax on it. However, the problem hasn't been solved completely because the implementation of these measures hasn't been as successful.

Enter the line number to be deleted:3

The contents of file after being modified are as follows:

Plastic bags are a major cause of environmental pollution.

Plastic as a substance is non-biodegradable and thus remain in the environment for hundreds of years.

Many countries around the globe have either put a ban on the plastic bag or Levi tax on it. However, the problem hasn't been solved completely because the implementation of these measures hasn't been as successful

Example 6: C Program to count number of vowels, Numbers, Special Symbols and Consonants in a given text file.

```

#include <stdio.h>

int main()
{

```

```

char ch, filename[100],vowels = 0,numbers=0,specialsymb=0;
FILE *fptr;
printf("Enter the File name: ");
scanf("%os", filename);
fptr = fopen(filename, "r");
printf("Reading the content of [ %s ]\n", filename);
while((ch = fgetc(fptr)) != EOF)
{
    if(ch == 'a' || ch == 'A' || ch == 'e' || ch == 'E' || ch == 'i' || ch == 'I' || ch == 'o' || ch == 'O' ||
    ch == 'u' || ch == 'U')
    {
        vowels++;
    }
    if(ch>=48 && ch<=57)
    {
        numbers++;
    }
    if((ch>=32 && ch<=47)||(ch>=58 && ch<=64)|| (ch>=91 && ch<=96)|| (ch>=123 &&
    ch<=127))
    {
        specialsymb++;
    }
    printf("%c", ch);
}
printf("\n\n");
printf("NUMBER OF VOWELS: %hu \n", vowels);
printf("\n");
printf("NUMBER OF NUMBERS: %hu \n", numbers);
printf("\n");
printf("NUMBER OF SPECIAL SYMBOLS: %hu \n", specialsymb);
fclose(fptr);
return 0;
}

```

OUTPUT

Enter the File name: ece2.txt

Reading the content of [ece2.txt]

Plastic bags are a major cause of environmental pollution.

Plastic as a substance is non-biodegradable and thus remain in the environment for hundreds of years.

Many countries around the globe have either put a ban on the plastic bag or Levi tax on it.

However, the problem hasn't been solved completely because the implementation of these measures hasn't been as successful.

NUMBER OF VOWELS: 122

NUMBER OF NUMBERS: 0

NUMBER OF SPECIAL SYMBOLS: 65

THIS PAGE IS INTENTIONALLY LEFT BLANK

C PROGRAMMING LAB

(Practical part of EST 102, Programming in C)

LIST OF LAB EXPERIMENTS

1	Familiarization of Hardware Components of a Computer
2	Familiarization of Linux environment – How to do Programming in C with Linux
3	Familiarization of console I/O and operators in C <ul style="list-style-type: none"> i) Display “Hello World” ii) Read two numbers, add them and display their sum iii) Read the radius of a circle, calculate its area and display it iv) Evaluate the arithmetic expression $((a -b / c * d + e) * (f +g))$ and display its solution. Read the values of the variables from the user through console.
4	Read 3 integer values and find the largest among them.
5	Read a Natural Number and check whether the number is prime or not.
6	Read a Natural Number and check whether the number is Armstrong or not.
7	Read n integers, store them in an array and find their sum and average.
8	Read n integers, store them in an array and search for an element in the array using an algorithm for Linear Search.
9	Read n integers, store them in an array and sort the elements in the array using Bubble Sort algorithm.
10	Read a string (word), store it in an array and check whether it is a palindrome word or not.
11	Read two strings (each one ending with a \$ symbol), store them in arrays and concatenate them without using library functions.
12	Read a string (ending with a \$ symbol), store it in an array and count the number of vowels, consonants and spaces in it.
13	Read two input each representing the distances between two points in the Euclidean space, store these in structure variables and add the two distance values.
14	Using structure, read and print data of n employees (Name, Employee Id and Salary).
15	Declare a union containing 5 string variables (Name, House Name, City Name, State and Pin code) each with a length of C_SIZE (user defined constant). Then, read and display the address of a person using a variable of the union.

16	Find the factorial of a given Natural Number n using recursive and non-recursive functions.
17	Read a string (word), store it in an array and obtain its reverse by using a user defined function.
18	<p>Write a menu driven program for performing matrix addition, multiplication and finding the transpose.</p> <p>Use functions to</p> <ul style="list-style-type: none"> (i) Read a matrix. (ii) Find the sum of two matrices. (iii) Find the product of two matrices. (iv) Find the transpose of a matrix (v) Display a matrix.
19	<p>Do the following using pointers</p> <ul style="list-style-type: none"> i) add two numbers ii) swap two numbers using a user defined function
20	Input and print the elements of an array using pointers.
21	Compute sum of the elements stored in an array using pointers and user defined function.
22	<p>Create a file and perform the following</p> <ul style="list-style-type: none"> i) Write data to the file ii) Read the data in a given file & display the file content on console iii) append new data and display on console
23	Open a text input file and count number of characters, words and lines in it; and store the results in an output file.

Experiment 1:	Familiarization of Hardware Components of a Computer.
----------------------	---

What is Computer?

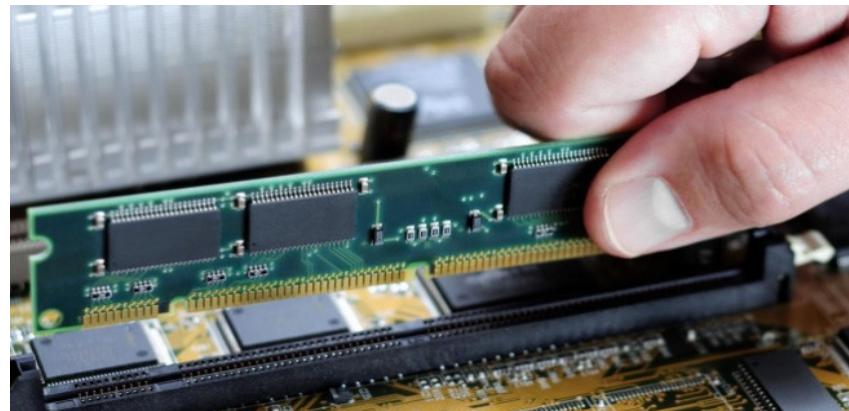
A computer is a programmable electronic device that accepts raw data as input and processes it with a set of instructions (a program) to produce the result as output.

The basic parts of a computer:

1. **Processor:** It executes instructions from software and hardware.



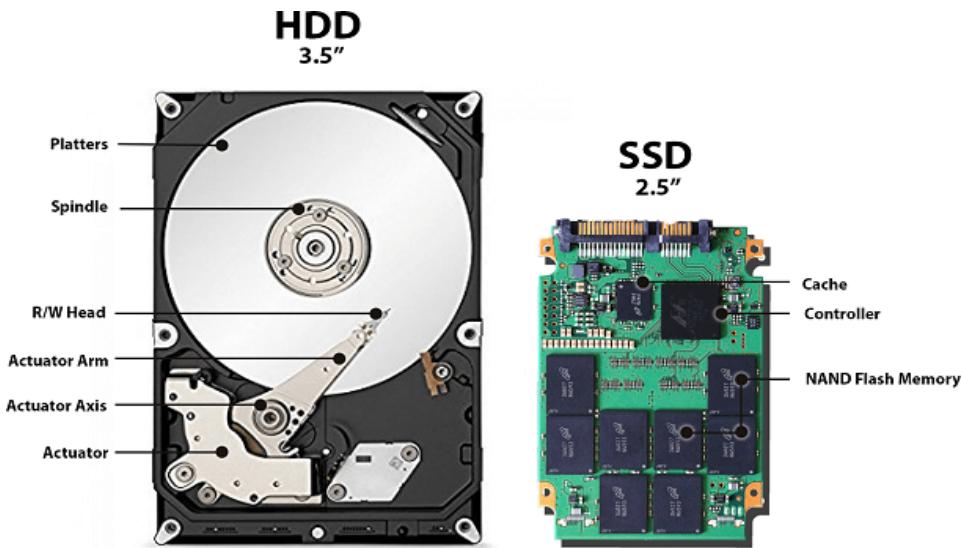
2. **Memory:** It is the primary memory for data transfer between the CPU and storage e.g., RAM (Random Access Memory).



3. **Motherboard:** It is the part that connects all other parts or components of a computer.



4. **Storage Device:** It permanently stores the data, e.g., SSDs, hard drive.



5. **Input Device:** It allows you to communicate with the computer or to input data, e.g., a keyboard, mouse.



6. **Output Device:** It enables you to see the output, e.g., monitor.

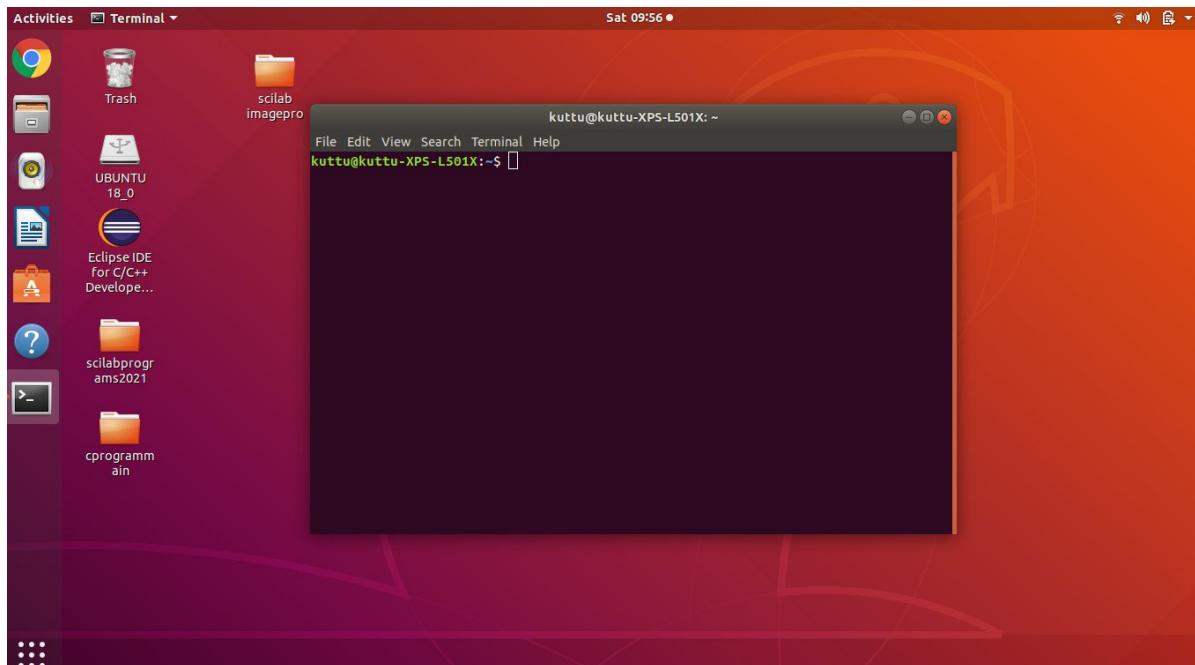


Experiment 2:	Familiarization of Linux environment – How to do Programming in C with Linux.
----------------------	---

Compile and Run a C Program on Ubuntu Linux:

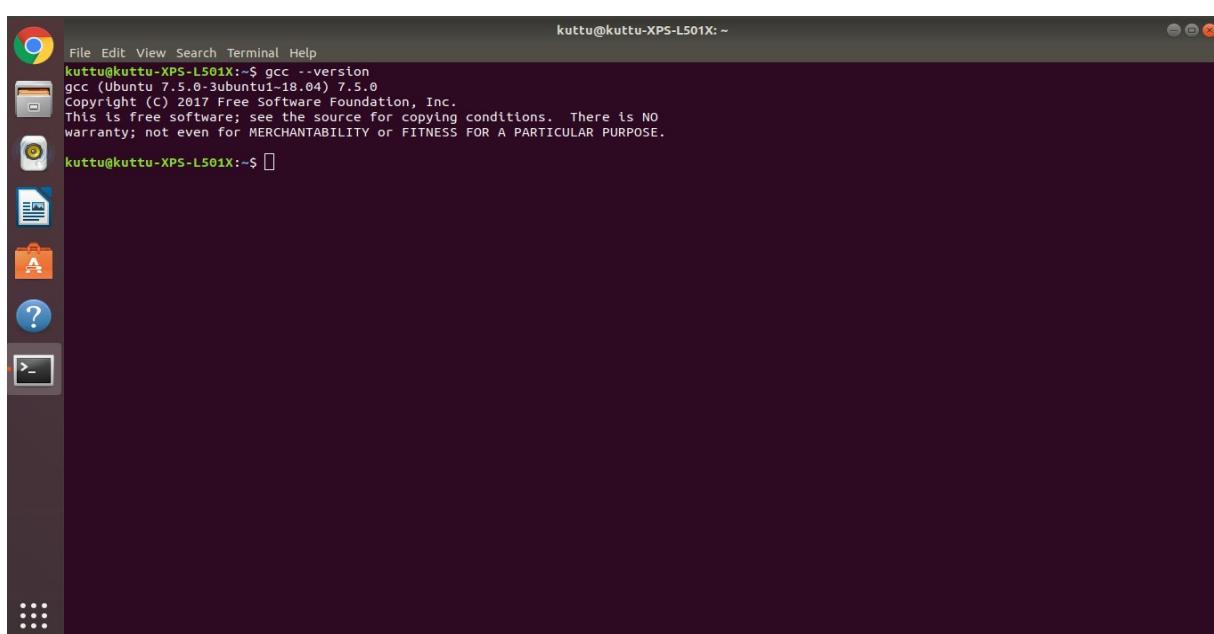
Step 1. Open up a terminal and check for GCC version:-

Open up a terminal by clicking the right button of mouse and selecting Open Terminal from it.



Now type the following command in the terminal window to check for GCC Version.

gcc --version



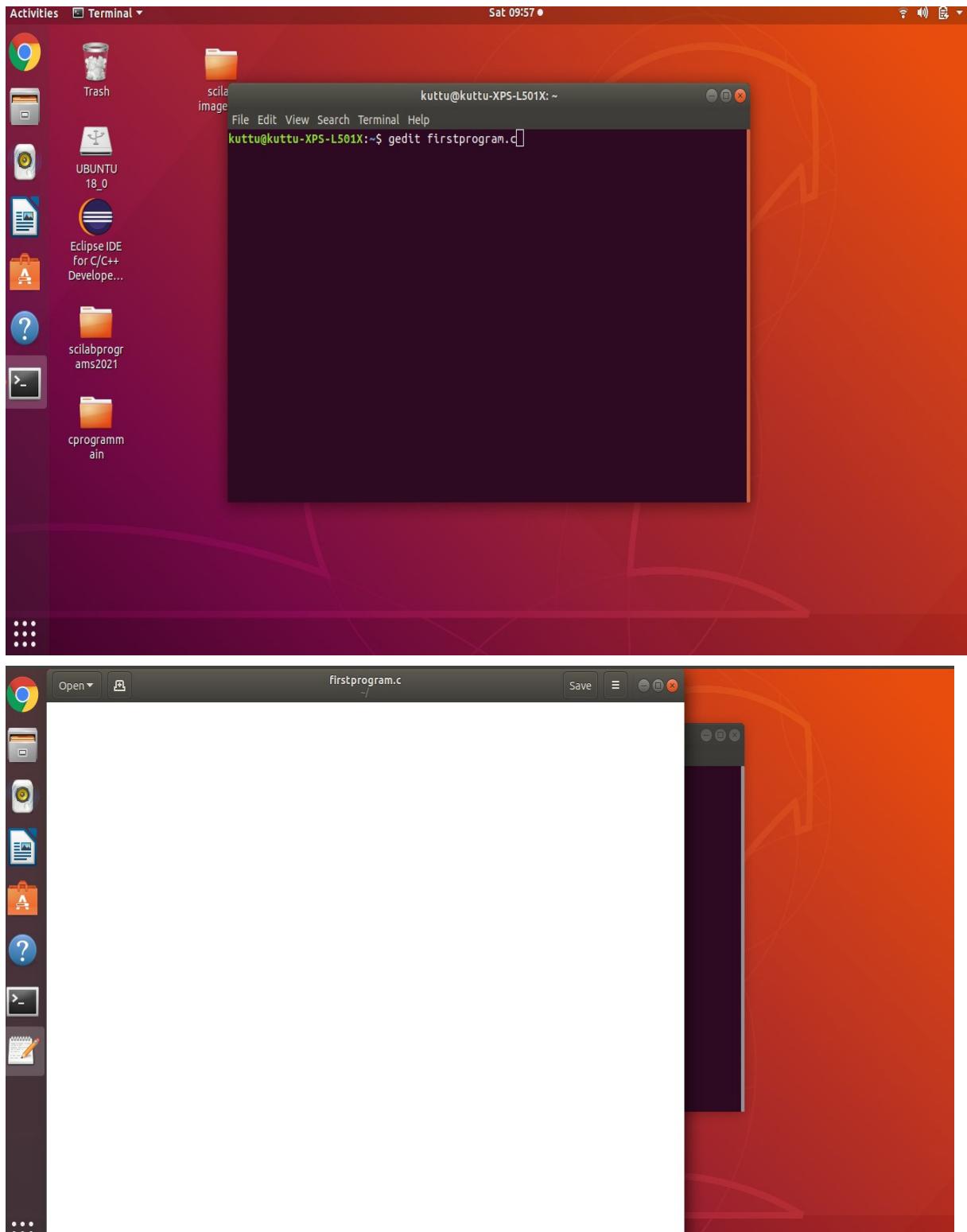
If the above content is displayed on the terminal then we are good to go with the programming.

Step 2. Use a text editor to create the C source code.

Type the following command

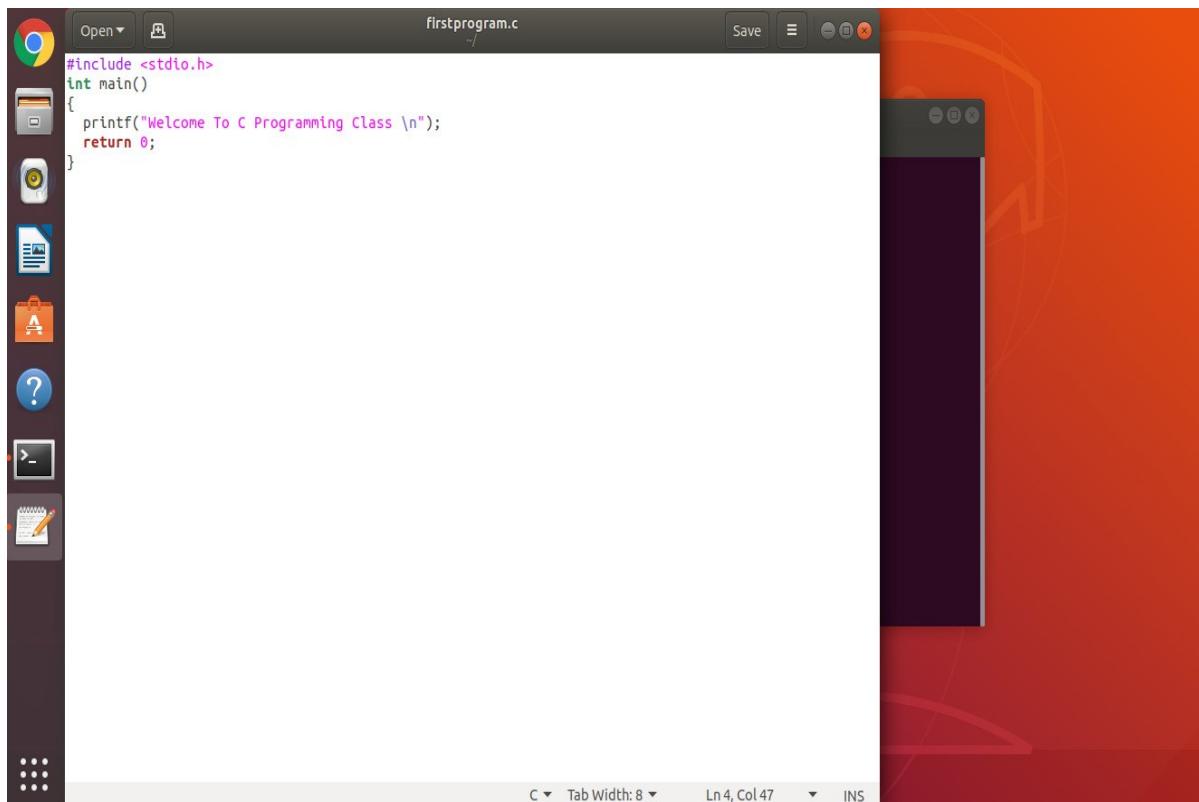
gedit firstprogram.c

This will open up a text file *firstprogram* with extension (.c).



Now enter the C source code shown below:

```
#include <stdio.h>
int main()
{
    printf("Welcome To C Programming Class \n");
    return 0;
}
```



After writing the code save the file and close the text editor.

Note: The file will be by default stored in the root directory. It is also possible to store the file in any user defined directory also.

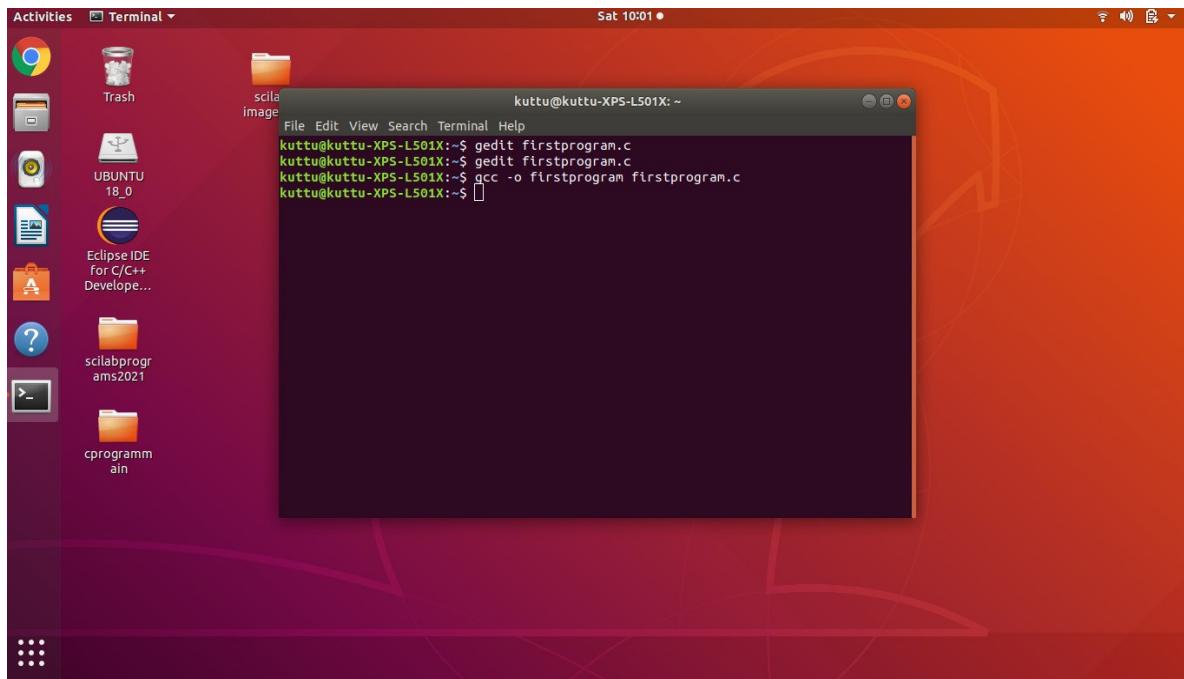
Step 3. Compile the program.

Type the following command in Terminal window.

```
gcc -o firstprogram firstprogram.c
```

This command will invoke the GNU C compiler to compile the file **firstprogram.c** and output (-o) the result to an executable called **firstprogram**. This file will be present in the same directory as the program file.

Note: Suppose if we had saved the program file in some other directory than the root directory, then using cd command we need to browse to that location and then type the above command.



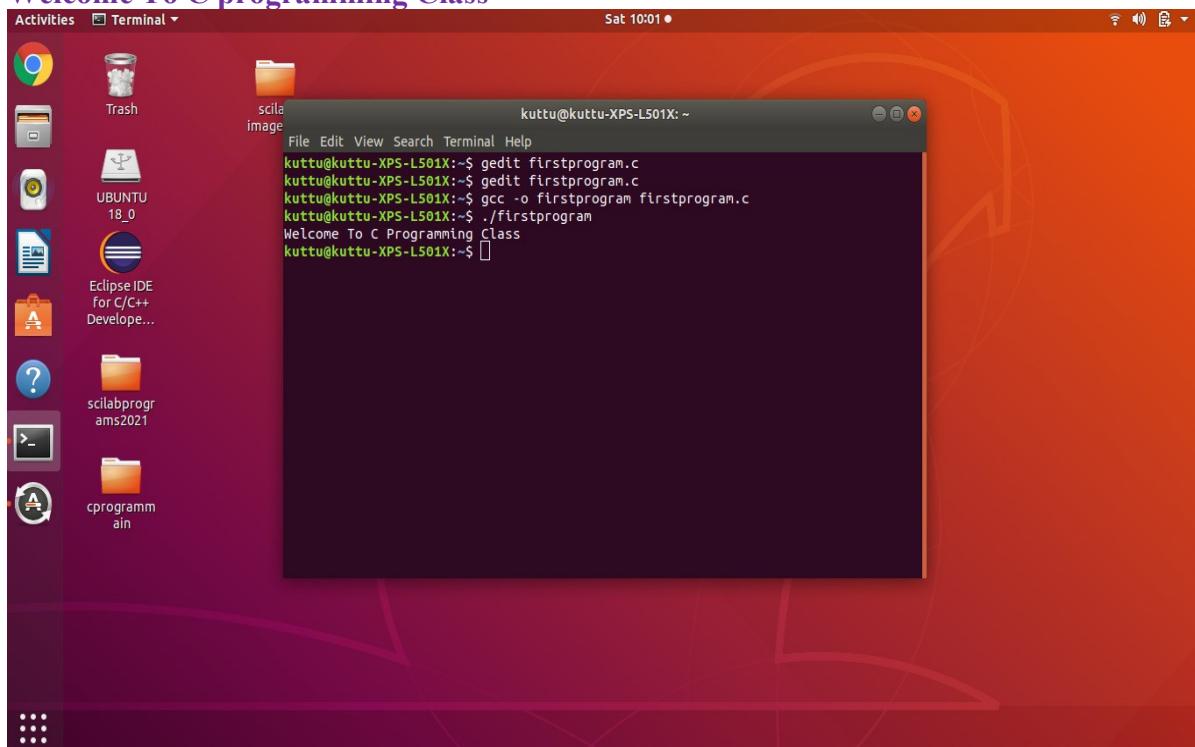
Step 4. Execute the program.

Type the command

. /firstprogram

This should result in the output

Welcome To C programming Class



Thus, we have successfully completed the creation of a C program and its execution in Linux Ubuntu 18.04.

Some useful Commands in Linux:

1. PWD Command

The **pwd** is short for *Present Working Directory*. It's a command-line utility tool that returns the path to the directory you're in at that moment.

2. CD Command

The **cd** command stands for “change directory,” and it allows you to navigate from one directory to another.

3. MV Command

The **mv** command is a utility command that moves files and folders from one location to another. The mv command can move a single file, multiple files, and directories.

4. RM Command

The **rm** command is short for "remove." It's used to delete files and directories.

5. MKDIR Command

The **mkdir** command is "make a directory." To create a directory, pass the name of the directory along with mkdir command.

6. LS Command

The **ls** is the list command in Linux, and it shows full list of files or contents of a directory. Just type ls and press the Enter key. The entire contents of the directory will be shown.

7. TOUCH Command

The **touch** is a Linux command that's used to quickly create a text file.

8. GEDIT Command

The **gedit** is the official text editor of the GNOME desktop environment. While aiming at simplicity and ease of use, gedit is a powerful general purpose text editor. It can be used to create and edit all kinds of text files.

Experiment 3:

Familiarization of console I/O and operators in C.

i) Display “Hello World”

```
/* Display “Hello World”*/
#include<stdio.h>
int main()
{
    printf("Hello World \n");
    return 0;
}
```

OUTPUT

Hello World

ii) Read two numbers, add them and display their sum

```
/*Read two numbers, add them and display their sum*/
#include<stdio.h>
int main(){
    int a,b,c;
    printf("Enter the value of a:= ");
    scanf("%d",&a);
    printf("Enter the value of b:= ");
    scanf("%d",&b);
    c=a+b;
    printf("The sum of %d and %d is %d \n",a,b,c);
    return 0;
}
```

OUTPUT

Enter the value of a:= 10

Enter the value of b:= 20

The sum of 10 and 20 is 30

iii) Read the radius of a circle, calculate its area and display it

```
/* Read the radius of a circle, calculate its area and display it*/
#include <stdio.h>
#define pi 3.14
```

```

int main(){
    int radius;
    float area;
    printf("Enter the radius of then circle:= ");
    scanf("%d",&radius);
    area=pi*radius*radius;
    printf("The area of a circle with %d cm is %.2f cm square \n",radius,area);
    return 0;
}

```

OUTPUT

```

Enter the radius of then circle:= 5
The area of a circle with 5 cm is 78.50 cm square

```

iv) Evaluate the arithmetic expression $((a -b / c * d + e) * (f +g))$ and display its solution.

Read the values of the variables from the user through console.

```

/*Evaluate the arithmetic expression ((a -b / c * d + e) * (f +g)) */
# include<stdio.h>
int main()
{
    float a,b,c,d,e,f,g,h;
    printf("Enter the value of a,b,c,d,e,f,g,h:= ");
    scanf("%f%f%f%f%f%f",&a,&b,&c,&d,&e,&f,&g);
    h=((a-b/c*d+e)*(f+g));
    printf("The output of the expression  is %.2f \n",h);
    return 0;
}

```

OUTPUT

```

Enter the value of a,b,c,d,e,f,g,h:= 9 8 7 6 5 4 3
The output of the expression  is 50.00

```

Experiment 4:

Read 3 integer values and find the largest among them.

```
//problem 4:--Read 3 integer values and find the largest amoung them
#include <stdio.h>
int main() {
    int a, b, c;
    printf("Enter three numbers: \n");
    scanf("%d %d %d", &a, &b, &c);
    if (a >= b && a >= c)
        printf("%d is the largest number.\n", a);
    else if (b >= c)
        printf("%d is the largest number.\n", b);
    else
        printf("%d is the largest number.\n", c);
    return 0;
}
```

OUTPUT

```
Enter three numbers:
```

```
20 30 40
```

```
40 is the largest number.
```

Experiment 5:

Read a Natural Number and check whether the number is prime or not.

```
// Read a Natural Number and check whether the number is prime or not
#include <stdio.h>
int main() {
    int n, i, flag = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    for (i = 2; i <= n / 2; i++) {
        // condition for non-prime
        if (n % i == 0)
    {
```

```

    flag = 1;
    break;
}
}

if (n == 1) {
    printf("1 is neither prime nor composite.");
}
else {
    if (flag == 0)
        printf("%d is a prime number.", n);
    else
        printf("%d is not a prime number.", n);
}
return 0;
}

```

OUTPUT

```

Enter a positive integer: 1
1 is neither prime nor composite.

Enter a positive integer: 3
3 is a prime number.

Enter a positive integer: 4
4 is not a prime number.

```

Experiment 6:	Read a Natural Number and check whether the number is Armstrong or not.
----------------------	---

```

//Read a Natural Number and check whether the number is Armstrong or not

#include <stdio.h>
#include <math.h>
int main() {
    int num, originalnum, remainder, n = 0;
    float result = 0.0;
    printf("Enter an integer: ");
    scanf("%d", &num);

```

```

originalnum = num;
// store the number of digits of num in n
for (originalnum = num; originalnum != 0; n++)
{
    originalnum /= 10;
}
for (originalnum = num; originalnum != 0; originalnum /= 10)
{
    remainder = originalnum % 10;
    // store the sum of the power of individual digits in result
    result += pow(remainder, n);
}
printf("%d\n",n);

// if num is equal to result, the number is an Armstrong number
if (result == num)
    printf("%d is an Armstrong number.", num);
else
    printf("%d is not an Armstrong number.", num);
return 0;
}

```

OUTPUT

```

Enter an integer: 153
153 is an Armstrong number.
Enter an integer: 431
431 is not an Armstrong number.

```

Experiment 7:	Read n integers, store them in an array and find their sum and average.
----------------------	---

```

/*Read n integers, store them in an array and find their sum and average*/
#include <stdio.h>
int main()
{
    int a[20],i,n,sum,avg;

```

```

printf("Enter size of array: ");
scanf("%d",&n);

//Entering the elements into an array
printf("Enter %d elements in the array :\n",n);
for(i=0;i<n;i++)
{
    scanf("%d", &a[i]);
}

// printing the elements of an array and summing
printf("\nElements in array are:");
for(i=0;i<n;i++)
{
    printf("%d ", a[i]);
    sum += a[i];
}
printf("\n\nThe sum of Elements in array is:= %d ",sum);
avg = sum/n;
printf("\n\nThe Average of Elements in array is:= %d\n",avg);
return 0;
}

```

OUTPUT

Enter 5 elements in the array :

5
4
3
2
1

Elements in array are:5 4 3 2 1

The sum of Elements in array is:= 15

The Average of Elements in array is:= 3

Experiment 8:	Read n integers, store them in an array and search for an element in the array using an algorithm for Linear Search.
----------------------	--

```
/*Read n integers, store them in an array and search for an element in the array using an
algorithm for Linear Search*/
#include <stdio.h>
int main()
{
    int a[20],i,n,flag,x;
    printf("Enter size of array: ");
    scanf("%d",&n);
    //Entering the elements into an array
    printf("Enter %d elements in the array :\n", n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    // printing the elements of an array
    printf("\nElements in array are:");
    for(i=0;i<n;i++)
    {
        printf(" %d ",a[i]);
    }
    //Entering the element to search
    printf("\nEnter the number to search:= ");
    scanf("%d",&x);
    for(i=0;i<n;i++)
    {
        if (a[i]==x)
        {
            flag=1;
            break;
        }
    }
}
```

```

if(flag == 0){
    printf("The number is not present in the array \n");
}
else{
    printf("The number is present in the array\n");
}
return 0;
}

```

OUTPUT

```

Enter size of array: 5
Enter 5 elements in the array :
5
4
3
2
1
Elements in array are:5 4 3 2 1
Enter the number to search:= 4
The number is present in the array

```

Experiment 9:	Read n integers, store them in an array and sort the elements in the array using Bubble Sort algorithm.
----------------------	---

```

/*Read n integers, store them in an array and sort the elements in the array using Bubble Sort
algorithm*/
#include <stdio.h>
int main()
{
    int a[20],i,j,n,flag,x,temp;
    printf("Enter size of array: ");
    scanf("%d",&n);
    //Entering the elements into an array
    printf("Enter %d elements in the array :\n", n);
    for(i=0;i<n;i++)

```

```

{
    scanf("%d",&a[i]);
}

// printing the elements of an array
printf("\nElements in array are:");
for(i=0;i<n;i++)
{
    printf("%d ",a[i]);
}
printf("\n");
printf("Bubble sort algorithm began----- \n");
for(i=0;i<n-1;i++)
{
    for (j=0;j<n-i-1;j++)
    {
        if (a[j] > a[j + 1])
        {
            temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
        }
    }
}
printf("Sorted Elements in array are:");
for(i=0;i<n;i++)
{
    printf("%d ",a[i]);
}
printf("\n");
return 0;
}

```

OUTPUT

```

Enter size of array: 5
Enter 5 elements in the array :

```

23 10 9 32 45

Elements in array are:23 10 9 32 45

Bubble sort algorithm began:-----

Sorted Elements in array are:9 10 23 32 45

Experiment 10:

Read a string (word), store it in an array and check whether it is a palindrome word or not.

```
/*Read a string (word), store it in an array and check whether it is a palindrome word or not*/
#include <stdio.h>
#include <string.h>
int main()
{
    char string01[20];
    int i,length;
    int flag = 0;
    printf("Enter a string:");
    scanf("%s", string01);
    length = strlen(string01);
    for(i=0;i < length ;i++)
    {
        if(string01[i] != string01[length-i-1])
        {
            flag = 1;
            break;
        }
    }
    if(flag==1) {
        printf("%s is not a palindrome\n", string01);
    }
    else {
        printf("%s is a palindrome\n", string01);
    }
    return 0;
```

```
}
```

OUTPUT

```
Enter a string:TENET
TENET is a palindrome
```

Experiment 11:	Read two strings (each one ending with a \$ symbol), store them in arrays and concatenate them without using library functions.
-----------------------	---

```
/* Read two strings (each one ending with a $ symbol), store them in arrays and concatenate
them without using library functions */

#include <stdio.h>

int main() {

    char s1[50], s2[50];

    int i, j;

    printf("Enter the string1: ");
    fgets(s1,50,stdin);

    printf("Enter the string2: ");
    fgets(s2,50,stdin);

    // store length of s1 in the i variable

    i = 0;

    while (s1[i] != '$') {

        i++;

    }

    // concatenate s2 to s1

    for (j = 0; s2[j] != '$'; j++, i++) {

        s1[i] = s2[j];

    }

    // terminating the s1 string

    s1[i] = '\0';

    printf("After concatenation: ");

    printf("%s\t",s1);

    printf("\n");

    return 0;

}
```

OUTPUT

```
Enter the string1: hai $  
Enter the string2: Jack$  
After concatenation: hai Jack
```

Experiment 12:	Read a string (ending with a \$ symbol), store it in an array and count the number of vowels, consonants and spaces in it.
-----------------------	--

```
/* Read a string (ending with a $ symbol), store it in an array and count the number of vowels, consonants and spaces in it.*/  
#include <stdio.h>  
int main() {  
char line[1000];  
int vowels, consonant, digit, space;  
vowels = consonant = digit = space = 0;  
printf("Enter a line of string: ");  
fgets(line,1000,stdin);  
for (int i = 0; line[i] != '$'; ++i)  
{  
    if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' || line[i] == 'o' || line[i] == 'u'  
        line[i] == 'A' || line[i] == 'E' || line[i] == 'I' || line[i] == 'O' || line[i] == 'U')  
    {  
        ++vowels;  
    }  
    else if ((line[i] >= 'a' && line[i] <= 'z') || (line[i] >= 'A' && line[i] <= 'Z'))  
    {  
        ++consonant;  
    }  
    else if (line[i] >= '0' && line[i] <= '9')  
    {  
        ++digit;  
    }  
    else if (line[i] == ' ')  
    {
```

```

    ++space;
}
}

printf("Vowels: %d", vowels);
printf("\nConsonants: %d", consonant);
printf("\nDigits: %d", digit);
printf("\nWhite spaces: %d", space);
printf("\n");
return 0;
}

```

OUTPUT

Enter a line of string: An essay is nothing but a piece of content which is written from the perception of writer or author.

Vowels: 33

Consonants: 75

Digits: 2

White spaces: 23

Experiment 13:	Read two input each representing the distances between two points in the Euclidean space, store these in structure variables and add the two distance values.
-----------------------	---

```

/*Read two input each representing the distances between two points in the Euclidean
space,store these in structure variables and add the two distance values.*/
#include <stdio.h>
struct point
{
int x;
int y;
};
int main()
{
struct point p1,p2,p3;
printf("Enter the first point(x1,y1):");

```

```

scanf("%d%d",&p1.x,&p1.y);
printf("Enter the second point(x2,y2):");
scanf("%d%d",&p2.x,&p2.y);
p3.x=p1.x+p2.x;
p3.y=p1.y+p2.y;
printf("New point after addition(x3,y3) is(%d,%d) :\n",p3.x,p3.y);
return 0;
}

```

OUTPUT

```

Enter the first point(x1,y1):3 4
Enter the second point(x2,y2):5 6
New point after addition(x3,y3) is:(8,10)

```

Experiment 14:	Using structure, read and print data of n employees (Name, Employee Id and Salary).
-----------------------	---

```

/*Using structure, read and print data of n employees (Name, Employee Id and Salary)*/
#include<stdio.h>
struct employee
{
char name[20];
int employid;
float salary;
};
int main()
{
int i,n;
printf("Enter number of employee whose date needs to be generated: ");
scanf("%d",&n);
struct employee emp[n];
for(i = 0; i < n; i++)
{
printf("\nEnter details of employee %d\n\n", i+1);
printf("Enter employee name: ");

```

```

scanf("%s", emp[i].name);
printf("Enter employee id: ");
scanf("%d", &emp[i].employid);
printf("Enter salary: ");
scanf("%f", &emp[i].salary);
}
printf("\n");
printf("Name\tEmployId\tSalary\n");
for(i = 0; i < n; i++ )
{
printf("%s\t%d\t%5.2f\n",emp[i].name,emp[i].employid,emp[i].salary);
}
return 0;
}

```

OUTPUT

Enter number of employee whose date needs to be generated: 3

Enter details of employee 1

Enter employee name: Raj

Enter employee id: 1020

Enter salary: 10000

Enter details of employee 2

Enter employee name: Sam

Enter employee id: 1030

Enter salary: 15000

Enter details of employee 3

Enter employee name: jack

Enter employee id: 1040

Enter salary: 20000

Name	EmployId	Salary
------	----------	--------

Raj	1020	10000.00
-----	------	----------

Sam	1030	15000.00
-----	------	----------

jack	1040	20000.00
------	------	----------

Experiment 15:	Declare a union containing 5 string variables (Name, House Name, City Name, State and Pin code) each with a length of C_SIZE (user defined constant). Then, read and display the address of a person using a variable of the union.
-----------------------	---

```
/*Declare a union containing 5 string variables (Name, House Name, City Name, State and
Pin code) each with a length of C_SIZE (user defined constant). Then, read and display the
address of a person using a variable of the union*/
#include <stdio.h>
#define C_SIZE 20
union Address
{
char name[C_SIZE];
char housename[C_SIZE];
char cityname[C_SIZE];
char state[C_SIZE];
char pin[C_SIZE];
};
int main()
{
union Address Data;
printf("Enter name:");
scanf("%s",Data.name);
printf("Enter house name:");
scanf("%s",Data.housename);
printf("Enter city name:");
scanf("%s",Data.cityname);
printf("Enter state name:");
scanf("%s",Data.state);
printf("Enter pin:");
scanf("%s",Data.pin);
printf("Printing the Data values ....\n");
printf("Name: %s \n", Data.name);
printf("House Name : %s\n", Data.housename);
```

```

printf("City Name : %s\n", Data.cityname);
printf("State Name : %s \n", Data.state);
printf("Pin: %s \n\n", Data.pin);
return 0;
}
/* the output is the last value only as union can hold only one value at a time */

```

OUTPUT

```

Enter name:JACK
Enter house name:LISvilla
Enter city name:Kollam
Enter state name:Kerala
Enter pin:691021
Printing the Data values ....
Name: 691021
House Name : 691021
City Name 691021
State Name : 691021
Pin: 691021

```

Experiment 16:	Find the factorial of a given Natural Number n using recursive and non-recursive functions.
-----------------------	---

```

/*Find the factorial of a given Natural Number n using recursive and non-recursive
functions*/
#include <stdio.h>
//FACTORIAL implementation using non-recursive function
long int factnonrec(int n)
{ int i;long int x=1;
for(i=1;i<=n;i++)
x=x*i;
return x;
}
//FACTORIAL implementation using recursive function

```

```

long int factrec(int n)
{
    if (n >= 1)
    {
        return (n*factrec(n-1));
    }
    else{
        return 1;
    }
}

// main program-----
int main()
{
    int n;
    printf("Enter the number ");
    scanf("%d",&n);
    if (n<=0)
    {
        printf("Error! Factorial of a negative number doesn't exist.\n");
    }
    else
    {
        printf("Factorial using non-recursive function %d!=%ld\n",n,factnonrec(n));
        printf("Factorial using recursive function %d!=%ld\n",n,factrec(n));
    }
    return 0;
}

```

OUTPUT

```

Enter the number:4
Factorial using non-recursive function 4!=24
Factorial using recursive function 4!=24

```

Experiment 17:	Read a string (word), store it in an array and obtain its reverse by using a user defined function.
-----------------------	---

```
/*Read a string (word), store it in an array and obtain its reverse by using a user defined
function*/
#include <stdio.h>
#include <string.h>
void revstring1(char Str[]);
int main()
{
char str[100];
printf("Enter the string: ");
fgets(str,100,stdin);
revstring1(str);
return 0;
}
void revstring1(char Str[])
{
char revstr[100];
int i, j, length;
j = 0;
length = strlen(Str);
for (i = length - 1; i >= 0; i--)
{
    revstr[j] = Str[i];
    j++;
    revstr[j] = '\0';
}
printf("Reversed String is : %s\n", revstr);
}
```

OUTPUT

Enter the string: Hai everyone welcome to SBCE

Reversed String is :

ECBS ot emoclew enoyreve iaH

Experiment 18:	<p>Write a menu driven program for performing matrix addition, multiplication and finding the transpose.</p> <p>Use functions to</p> <ul style="list-style-type: none"> (i) Read a matrix. (ii) Find the sum of two matrices. (iii) Find the product of two matrices. (iv) Find the transpose of a matrix (v) Display a matrix.
-----------------------	--

```
/* Write a menu driven program for performing matrix addition, multiplication and finding
the transpose. */

#include <stdio.h>

void readmatrix(int x[10][10],int row,int col);
void displaymatrix(int y[10][10],int row,int col);
void addmatrix(int x[10][10],int y[10][10],int row,int col) ;
void multiplymatrix(int x[10][10],int y[10][10],int row,int col,int val) ;
void transposematrix(int any[10][10],int row,int col);
int main()
{
    int a[10][10],b[10][10],row,col,val;
    printf ("Enter the number of rows:");
    scanf("%d",&row);
    printf ("Enter the number of Columns:");
    scanf("%d",&col);
    printf("Enter the number of Columns in first matrix:");
    scanf("%d",&val);
    //reading A matrix
    printf("\nReading the A matrix. ....\n");
    readmatrix(a,row,col);
    //reading B matrix
    printf("\nReading the B matrix. ....\n");
    readmatrix(b,row,col);
    //Display A matrix
    printf("\nDisplaying A matrix.....\n");
}
```

```

displaymatrix(a,row,col);
//Display B matrix
printf("\nDisplaying B matrix.....\n");
displaymatrix(b,row,col);
//Displaying sum of two matrices
printf("\nDisplaying the sum of A and B matrices ..... \n");
addmatrix(a, b,row,col);
//Displaying product of two matrices
printf("\nDisplaying the product of A and B matrices ..... \n");
multiplymatrix(a,b,row,col,val) ;
//Displaying transpose of A matrix
printf("\nDisplaying transpose of A matrix. ....\n");
transposematrix(a,row,col);
//Displaying transpose of B matrix
printf("\nDisplaying transpose of B matrix.....\n");
transposematrix(b,row,col);
printf("\n");
return 0;
}

//Function to read the matrix value
void readmatrix(int read[10][10],int row,int col)
{
int i,j;
for(i=0;i<row;i++)
{
    for(j=0;j<col;j++)
    {
        scanf("%d",&read[i][j]);
    }
}
}

//Function to display the matrix value
void displaymatrix(int disp[10][10],int row,int col)
{
}

```

```

int i,j;
for(i=0;i<row;i++)
{
printf("\n");
for(j=0;j<col;j++)
{
printf("%5d",disp[i][j]);
}
}
}

// Function to sum two matrices
void addmatrix(int x[10][10],int y[10][10],int row,int col)
{
int i,j,sum[10][10];
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
sum[i][j]=x[i][j]+y[i][j];
}
}
displaymatrix(sum,row,col);
}

//Function to Multiply A and B matrix to get C matrix
void multiplymatrix(int x[10][10],int y[10][10],int row,int col,int val)
{
int i,j,k,mul[10][10];
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
{
mul[i][j]=0;
for (k = 0; k < val; k++)
{
{
}
}
}
}
}

```



```

        mul[i][j] = mul[i][j] + (x[i][k] * y[k][j]);
    }
}
}
displaymatrix(mul, row, col);
}

//Function to Transpose any matrix
void transposematrix(int any[10][10], int row, int col)
{
    int i, j, c[10][10];
    {
        for(i=0; i<row; i++)
        {
            for(j=0; j<col; j++)
            {
                c[j][i] = any[i][j];
            }
        }
    }
    displaymatrix(c, row, col);
}

```

OUTPUT

Enter the number of rows:2

Enter the number of Columns:2

Enter the number of Columns in first matrix:2

Reading the A matrix.....

1
2
3
4

Reading the B matrix.....

4
3
2



1

Displaying A matrix.....

1 2

3 4

Displaying B matrix.....

4 3

2 1

Displaying the sum of A and B matrices.....

5 5

5 5

Displaying the product of A and B matrices.....

8 5

20 13

Displaying transpose of A matrix.....

1 3

2 4

Displaying transpose of B matrix.....

4 2

3 1

Experiment 19:

Do the following using pointers

i) add two numbers

ii) swap two numbers using a user defined function

i) Add two numbers using pointers.

```
//Do the following using pointers i) add two numbers
#include <stdio.h>
int main()
{
int x, y, *a, *b, sum;
printf("Enter two integers to add: ");
scanf("%d%d", &x, &y);
a= &x;
b= &y;
sum = *a + *b;
```

```
printf("Sum of the numbers = %d \n", sum);
return 0;
}
```

OUTPUT

```
Enter two integers to add: 23 34
```

```
Sum of the numbers = 57
```

ii) Swap two numbers using a user defined function.

```
// swap two numbers using user defined function
#include <stdio.h>
void swap(int *x,int *y);
int main()
{
int a, b;
printf("Enter Value of A: ");
scanf("%d", &a);
printf("Enter Value of B: ");
scanf("%d", &b);
printf("The value of A and B Before swapping is %d and %d . \n", a,b );
swap(&a, &b); // call by reference
}
void swap(int *x, int *y)
{
int temp = 0;
temp = *x;
*x = *y;
*y = temp;
printf("The value of A and B after swapping is %d and %d . \n", *x,*y );
}
```

OUTPUT

```
Enter Value of A: 20
```

```
Enter Value of B: 30
```

```
The value of A and B Before swapping is 20 and 30 .
```

```
The value of A and B after swapping is 30 and 20 .
```

Experiment 20:

Input and print the elements of an array using pointers.

```
//Input and Print the elements of an array using pointers.  
#include <stdio.h>  
int main()  
{  
int arr[20]; int n, i;  
int * ptr = arr;  
printf("Enter size of an array: ");  
scanf("%d", &n);  
printf("Enter the elements in the array:\n");  
for (i = 0; i < n; i++)  
{  
    scanf("%d", (ptr + i));  
}  
printf("Array elements are: \n");  
for (i = 0; i < n; i++)  
{  
    printf("%d\t", *(ptr + i));  
}  
return 0;  
}
```

OUTPUT

```
Enter size of an array: 5  
Enter the elements in the array:  
23  
34  
45  
67  
78  
Array elements are:  
23    34    45    67    78
```

Experiment 21:	Compute sum of the elements stored in an array using pointers and user defined function.
-----------------------	--

```
//Compute sum of the elements stored in an array using pointers and user defined function.

#include <stdio.h>

void sumofarray(int *ptr,int n);

int main()
{
int arr[10];
int n, i;
int * ptr = arr;
printf("Enter size of an array: ");
scanf("%d", &n);
printf("Enter the elements in the array:\n");
for (i = 0; i < n; i++)
{
    scanf("%d", (ptr + i));
}
sumofarray(ptr,n);
return 0;
}

//Function to sum the elements of an array
void sumofarray(int *ptr,int n)
{
int i, sum=0;
for (i = 0; i < n; i++)
{
    sum= sum+ *(ptr + i);
}
printf("The sum of the array elements is %d .\n",sum);
}
```

OUTPUT

```
Enter size of an array: 5
Enter the elements in the array:
```

12
12
13
14
15
The sum of the array elements is 66 .

Experiment 22:	Create a file and perform the following i) Write data to the file ii) Read the data in a given file & display the file content on Console iii) append new data and display on console
-----------------------	---

```
/* Create a file and perform the following i) Write data to the file ii) Read the data in a given
file & display the file content on Console iii) append new data and display on console */
#include <stdio.h>
int main()
{
    FILE *fp;
    char ch[100];
    //opening the file before appending
    fp = fopen("ece1.txt", "w");
    // Write content to file
    fprintf(fp,"Welcome to SBCE.");
    fclose(fp);
    fp = fopen("ece1.txt", "r");
    printf("The file Before appending: ");
    printf("%os",fgets(ch,100,fp));
    printf("\n");
    fclose(fp);
    // Append mode is been called
    fp = fopen("ece1.txt", "a");
    // Write content to file
    fprintf(fp,"A NACC accredited institution.");
    fclose(fp);
```

```

//opening the file after appending
fp = fopen("ece1.txt", "r");
printf("The file After appending: ");
printf("%os",fgets(ch,100,fp));
printf("\n");
fclose(fp);
return 0;
}

```

OUTPUT

```

The file Before appending: Welcome to SBCE.
The file After appending: Welcome to SBCE.A NACC accredited institution.

```

Experiment 23:	Open a text input file and count number of characters, words and lines in it; and store the results in an output file.
-----------------------	--

```

/* Open a text input file and count number of characters, words and lines in it; and store the
results in an output file.*/
#include <stdio.h>
int main()
{
    FILE * ptr;
    char text[100];
    char ch;
    int characters=0, words=0, lines=0,space=0;
    ptr = fopen("ece1.txt", "r");
    printf("\nThe File Contents is:");
    printf("\n-----\n");
    while ((ch=fgetc(ptr))!= EOF)
    {
        printf("%c",ch);
        characters++;
        if (ch == '\n' || ch == '\0')
        {
            lines++;
        }
    }
}

```

```

    }
    if (ch == ' ' || ch == '\t' || ch == '\n' || ch == '\0')
    {
        words++;
    }
    if (ch == '\n')
    {
        space++;
    }
}

// Increment words and lines for last word

if (characters > 0)
{
    lines++;
    words++;
}

printf("\n-----\n");
printf("Total number of characters = %d\n", characters - space);
printf("Total number of words    = %d\n", words);
printf("Total number of lines    = %d\n", lines);
fclose(ptr);
return 0;
}

```

OUTPUT

The File Contents is:

Welcome to SBCE.

A NACC accredited institution.

Located in Pandalam.

Total number of characters = 61

Total number of words = 10

Total number of lines = 3

Bibliography

- 1) [Learn C Programming \(programiz.com\)](https://www.programiz.com/c-programming)
- 2) [Learn C Programming Language Tutorial - javatpoint](https://www.javatpoint.com/c-programming-tutorial)
- 3) [C Tutorial \(tutorialspoint.com\)](https://www.tutorialspoint.com/c_programming/index.htm)
- 4) [C Programming Language - GeeksforGeeks](https://www.geeksforgeeks.org/c-programming-language-tutorial/)
- 5) [C Tutorial - Learn C Programming \(w3schools.in\)](https://www.w3schools.com/c/)