MODULE I

INTRODUCTION TO C LANGUAGE

Introduction to computer Hardware and software: Computer generations, computer types, bits, bytes and words, CPU, Primary memory, Secondary memory, ports and connections, input devices, output devices, Computers in a network, Network hardware, Software basics, software types.

Overview of C: Basic structure of C program, executing a C program. Constant, variable and data types, Operators and expressions.

HISTORY OF C

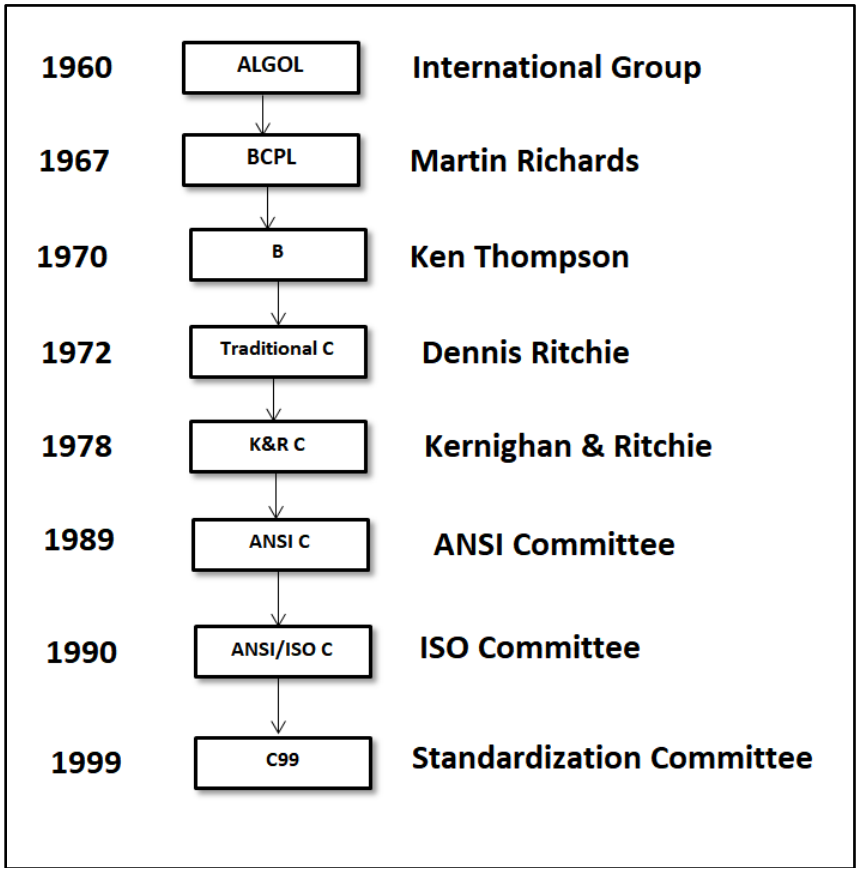


Fig 1.1 History of ANSI C

- 'C' programming language is a structured, high-level, machine independent language.
- The root of all modem languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure.
- In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software.
- In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories.
- C was developed by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features.
- During 1970s, C had evolved into what is now known as "traditional C". The language became more popular after publication of the book The C Programming Language' by Brian Kerningham and Dennis Ritchie in 1978.

- The book was so popular that the language came to be known as "K&R C" among the programming community.
- In 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C.
- It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.
- C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99.

IMPORTANCE OF C

- It is a robust language.
- Programs written in 'C' are efficient and fast. (Because of variety of data types and powerful operators)
- Highly Portable. (related to OS)
- Well suited for structured programming.
- Ability to extend itself

COMPUTER GENERATIONS

Generation in computer terminology is a change in technology of a computer.

Five generations of computer,

First Generation (1942-1954) uses vacuum tubes as basic components for memory and circuitry for CPU. In this generation mainly batch processing operating system were used. There were machine code and electric wired board languages used.

Advantages

- Vacuum tube technology made possible to make electronic digital computers.
- These computers could calculate data in millisecond.

Disadvantages

- The computers were very large in size.
- They consumed a large amount of energy.
- Air conditioning was required.

Second Generation (1952-1964): The second generation computers use transistors they were cheaper, consumed less power, more compact in size, more reliable and faster than first generation. In this generation assembly language and high level programming language like FORTRAN, COBOL were used. Batch Processing and Multiprogramming operating system were used.

Advantages

- Smaller in size as compared to the first generation computers.
- Used faster peripherals like tape drives, magnetic disks, printer etc.

Disadvantages

- Cooling system was required
- Costly and not versatile

Third Generation (1964-1972): The Third generation computers used the integrated circuits (IC). In this generation remote processing, time sharing, real time operating systems were used. High level programming languages like FORTRAN, COBOL, BASIC, ALGOL-68 were used.

Advantages

- Smaller in size as compared to previous generations.
- Better speed and could calculate data in nanoseconds.
- Used fan for heat discharge to prevent damage.

Disadvantages

- Air conditioning was required.
- Highly sophisticated technology required for the manufacturing of IC chips.

Fourth Generation (1972-1990): The fourth generation computers use Very Large Scale Integrated Circuits (VLSI). In this generation time sharing, real time, networks, distributed operating systems were used. High level programming languages like C, C++, DBASE etc were used.

Advantages

- More powerful and reliable than previous generations.
- Small in size
- No air conditioning required.

Disadvantages

- The latest technology is required for manufacturing of Microprocessors.

Fifth Generation (1990-Till Date): In fifth generation computers VLSI became ULSI (Ultra Large Scale Integration Technology). All the higher level languages like C , C++, Java, .NET etc are used in this generation. AI (Artificial Intelligence) is an emerging technology in this generation.

Advantages

- More user friendly.
- Availability of very powerful and compact computers at cheaper rates.

Computer Types

We can classify computers based on following criteria

1. According to Purpose
2. According to Technology
3. According to size and storage capacity

According to Purpose: It is classified into 2 types

a) General Purpose Computers: Almost all computers used in offices for commercial, educational and other applications are general purpose computers.

b) Special Purpose Computers: Computers that are designed to perform special tasks like scientific applications and research, weather forecasting etc are called special purpose computers.

According to Technology: It is classified into 3 types

a) **Analog Computers:** It is a special purpose computer that represents and stores the data continuously. Example: Thermometer, Speedometer

b) **Digital Computers:** It is a general purpose computer that represents and stores the data in discrete quantity and numbers. Example: PC, Desktop

c) **Hybrid computers:** Incorporate the technology of both analog and digital computers. These computer stores and process discrete numbers from digital to analog and from analog to digital. Example: Artificial Intelligence

According to size and storage capacity: it is classified into 4 types

a) **Super computer:** Biggest and fastest computers designed mainly for complex and scientific applications. It has many CPU's which can operate in parallel. Example: CRAY3, NEC-500
b) **Mainframe Computer:** It is an large and fast computers but smaller and slower than super computers. It has single CPU shared by many users. Example: IBM 3090, IBM 4300
c) **Mini computer:** are medium scale, smaller and generally slower than main frame computers. They have many terminals which are connected with one CPU and can support many users. Example PDD-1, IBM AS/400
d) **Micro computer:** it is an smallest digital computer which uses a micro-processor as its CPU. It can be used as standalone and in a multi-user environment. Example: Laptops, Notebook.

Bits, Bytes and Words

- **Bit:** It is a single digit in a binary number and it can be either 1 or 0. It can store 256 different combinations of ON and OFF states.
- **Byte:** A byte is 8 bit combinations of 1's and 0's.
- **Word:** A word is basically a number of bits CPU can deal with one go. Example 32bit machine, 64bit machine and so on.

BLOCK DIAGRAM OF COMPUTER

Computer is an electronic device and it performs the following 5 basic operations. It converts the raw input data into information, which is useful to the users.

- *Inputting*: It is the process of entering data & instructions to the computer system.
- *Storing*: The data & instructions are stored for either initial or additional processing, as & when required.
- *Processing*: It requires performing arithmetic or logical operation on the saved data to convert it into useful information.
- *Outputting*: It is the process of producing the output data to the end user.
- *Controlling*: The above operations have to be directed in a particular sequence to be completed.

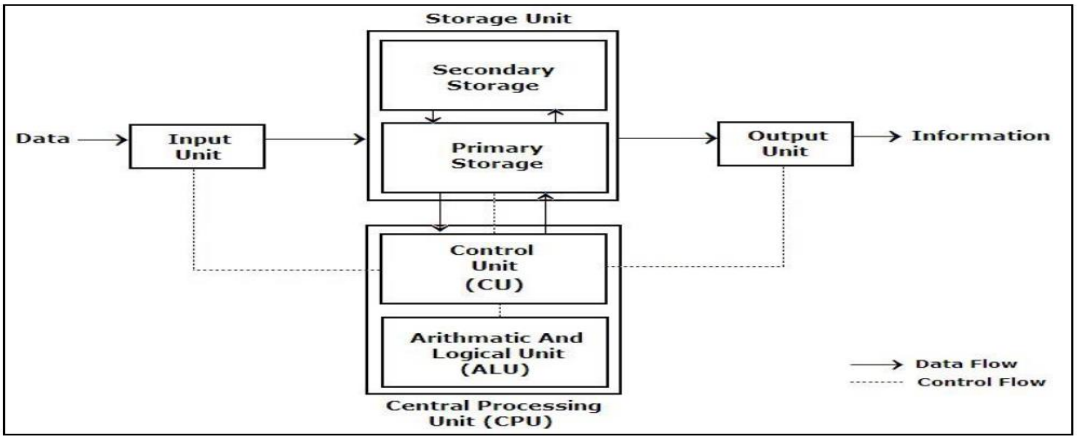


Fig 1: Block Diagram of a Computer

***Input Unit*:** We need to first enter the data & instruction in the computer system, before any computation begins. This task is accomplished by the input devices. (Eg: keyboard, mouse, scanner, digital camera etc).

***Storage Unit*:** The data & instruction that are entered have to be stored in the computer. This storage unit is designed to save the initial data, the intermediate result & the final result.

This storage unit has 2 units: *Primary storage* & *Secondary storage.*

- ***Primary Storage*:** The primary storage, also called as the *main memory*, holds the data when the computer is currently on. As soon as the system is switched off or restarted, the information held in primary storage disappears (i.e. it is volatile in nature). Moreover, the primary storage normally has a limited storage capacity, because it is very expensive as it is made up of semiconductor devices.
  Example: Random Access Memory (RAM), cache etc.
- ***Secondary Storage*:** The secondary storage, also called as the *auxiliary storage*, handles the storage limitation & the volatile nature of the primary memory. It can retain information even when the system is off.
  Examples: floppy disks, USB devices etc.

***Central Processing Unit*:** Together the Control Unit & the Arithmetic Logic Unit are called as the Central Processing Unit (CPU). The CPU is the brain of the computer. The CPU is responsible for activating & controlling the operation of other units of the computer system.

***Arithmetic Logic Unit*:** An arithmetic logic unit (**ALU**) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer.

***Control Unit*:** The **control unit** (CU) is a component of a computer's central processing **unit** (CPU) that directs the operation of the processor. It tells the computer's memory, arithmetic/logic **unit** and input and output devices how to respond to a program's instructions.

***Output Unit*:** It accepts the results produced by the computer in coded form. It converts these coded results to human readable form. Finally, it displays the converted results to the outside world with the help of output devices (Eg :monitors, printers, projectors etc..).

A **computer** system consists of two major elements:

1. *Hardware*- This hardware is responsible for all the physical work of the computer.
2. *Software*- This software commands the hardware what to do & how to do it.

MEMORY

- A memory is just like a human brain.
- It is used to store data and instructions.
- Memory is a storage space where data is processed and stored.

Memory is of 3 types,

1. **Cache Memory:** It is a very high speed semiconductor memory. It acts as an buffer between CPU and Main memory.
2. ***Primary Storage*:** The primary storage, also called as the *main memory*, holds the data when the computer is currently on. As soon as the system is switched off or restarted, the information held in primary storage disappears (i.e. it is volatile in nature). Moreover, the primary storage normally has a limited storage capacity, because it is very expensive as it is made up of semiconductor devices.
   Example: Random Access Memory (RAM), cache etc.
3. ***Secondary Storage*:** The secondary storage, also called as the *auxiliary storage*, handles the storage limitation & the volatile nature of the primary memory. It can retain information even when the system is off.
   Examples: floppy disks, USB devices etc.

Random Access Memory (RAM)

- RAM constitutes the internal memory of the CPU for storing data, program and program result.
- RAM is read/write Memory and it is volatile in nature
- RAM is of 2 types
    - **Static RAM:** The memory retains its contents as long as power remains applied.
    - **Dynamic RAM:** It must be continuously refreshed in order to maintain the data.

Read Only Memory (ROM)

- The memory which can only read but cannot write on it, this type of memory non-volatile in nature.
- Information's are stored permanently.
- ROM is of 4 types

    - **MROM (Masked ROM):** ROM's are hardware devices that contain a pre-programmed set of data, these kinds of ROM is called as MROM.
    - **PROM (Programmable ROM):** is a read only memory that can be modified only once by a user by adding the desired contents.
    - **EPROM (Erasable and programmable ROM):** EPROM can be erased by exposing it to ultra violet light for duration of upto 40minutes.
    - **EEPROM (Electrically Erasable and Programmable ROM):** EEPROM is erased electrically one byte at a time.

Flash Memory

- It is a form of EEPROM that allows multiple memory locations to be erased or written in one programming operation.

PORTS AND CONNECTIONS

- A PORT is a physical docking point using which an external device can be connected to the computer.
- Different types of PORTS are:
    - **Serial Port:** used for external modems and older computer mouse, data is transferred serially.
    - **Parallel Port:** used for scanners and printers, data is transferred parallel.
    - **PS/2 Port:** used for old computer keyboard and mouse.
    - **USB Port (Universal Serial Bus):** it can connect all kinds of external USB devices. Example Mouse, Keyboard, Printer etc
    - **VGA Port:** Connects monitor to the computer video card, VGA ports has holes.
    - **Power Connector:** Connects computers power cable that plugs into a power bar.
    - **Firewire Port:** connects camcorders and video equipment to the computer.
    - **Modem Port:** Connects a PC's Modem to the Telephone Network.
    - **Ethernet Port:** Connects the computer cable to the network.
    - **Game Port:** Connects a joystick to a PC, now replaced by USB.
    - **DVI Port (Digital Video Interface):** Connects flat panel LCD Monitor to the computers high- end video graphics Cards.
    - **Sockets:** Connects the Micro-phone and Speakers to the sound card of the computer.

INPUT DEVICES

- The devices which are used to give data and instructions to the computer are called Input Devices.
- Various types of input devices can be used with the computer depending upon the type of data you want to enter in the computer.
- Example: keyboard, mouse, joystick, light pen, etc.

1. **Keyboard:** It is the most commonly used input device. It is used to enter data and instructions directly into the computer. There are 104 buttons on the keyboard which are called keys.
2. **Mouse:** Mouse is another input device which is commonly found connected with the computers. It is basically a pointing device which works on the principle of Point and Click.
3. **Light Pen:** Light pen is another pointing type input device. It is a pen shaped device which can be used by directly pointing the objects on the screen. It can also be used for making drawings directly on the monitor screen.
4. **Scanner:** We can store pictures, photographs, diagrams into the computer with the help of scanner. The scanner reads the image and saves it in the computer as a file.
5. **Microphone:** This is an input device which is used to record sound or voice into the computer system. You can store voice data in the computer by speaking in front of this device.
6. **Game Pad:** A game-pad is a type of game controller held in two hands, where the fingers (especially thumbs)are used to provide input by pressing buttons on it. It is also known as Control Pad.

OUTPUT DEVICES

- The devices which are used to display the results or information are called **Output Devices**.
- You can view the output on the monitor or you can print it on a paper using a printer.
- Monitor and the printer are the commonly used output devices.

1. **Monitor:** This is the most common output device connected with the computer to display the processed information. It looks like a TV and is also know as **VDU(Visual Display Unit).**
2. P**rinter:** It gives a printed output of the results that appears on the monitor screen. Printed output is also called Hard Copy output because unlike monitor, this output can be preserved even if the computer is switched off.

COMPUTER IN A NETWORK

General Network Techniques:

- Computers communicate on a network, they send data packets to other computers connected to a network bus.
- To distinguish between the computers every computer have an unique ID called **MAC address** (Media Access Control).
- The MAC-addresses is primarily used by the computers to filter out incoming network traffic that is addressed to the individual computer.
- All computers hear all network traffic since they are connected to the same bus. This network structure is called multi-drop.
- If several computers communicate with each other at high speed they may not be able to utilize more than 25% of the total network bandwidth. Hence it causes bandwidth problem.

Characteristics of a Computer Network

- Share resources from one computer to another.
- Create files and store them in one computer, access those files from the other computer(s) connected over the network.
- Connect a printer, scanner, or a fax machine to one computer within the network and let other computers of the network use the machines available over the network.

NETWORK HARDWARE: Network hardware is the individual components of a network system that are responsible for transmitting data and facilitating the operations of a computer network.

Following is the list of hardware's required to set up a computer network.

- Network Cables
- Distributors
- Routers
- Internal Network Cards
- External Network Cards

Network Cables: Network cables are used to connect computers. Example: Category 5 cable RJ-45.

Distributors: A computer can be connected to another one via a serial port but if we need to connect many computers to produce a network, this serial connection will not work. The solution is to use a central body to which other computers, printers, scanners, etc. can be connected and then this body will manage or distribute network traffic.

Router:

- A router is a type of device which acts as the central point among computers and other devices that are a part of the network.
- It is equipped with holes called ports.
- Computers and other devices are connected to a router using network cables.
- Now-a-days router comes in wireless modes using which computers can be connected without any physical cable.
- Routers determine the path for the transfer of data as a processing unit for information packets.
- The router uses a specific **protocol** or set of rules to determine which information packets are to be routed.

Network Interface Card: Network interface cards are used to connect each computer to the network so they can communicate with the network router to receive information packets.

Network cards are of two types: Internal and External Network Cards.

1. **Internal Network Cards:** Motherboard has a slot for internal network card where it is to be inserted.
2. **External Network Cards:** External network cards are of two types: Wireless and USB based. Wireless network card needs to be inserted into the motherboard; however no network cable is required to connect to the network.

Network Switches: Network switches work similar to routers because they both copy information from one area of the network to the other. However, network switches contain multiple ports for copying frames of information from one port to the other.

Network Bridge: A network bridge divides traffic on a local area network by separating the LAN into several different segments. It is also responsible for filtering data by determining the data destination or discarding unnecessary data.

Universal Serial Bus (USB): USB card is easy to use and connects via USB port. Computers automatically detect USB card and can install drivers required to support the USB network card automatically.

SOFTWARE BASICS

A set of instructions that achieve a single outcome are called program. Many programs functioning together to do tasks make software**.** Example, word-processing, web browser etc**.** There are two categories of software — System Software, Application Software, Utility software.

- System software: system software acts as interface between hardware and user application.
- To run the hardware parts of the computer and other applications system software is required.
- System software are of 4 types: operating system, Language processor, Device drivers.

Operating system: System software that is responsible for functioning of all hardware parts and their interoperability to carry tasks is called operating system. OS is booted once computer is switched on. OS manages computers functions like storage and retrieval of data, scheduling etc..

Language Processor: Function of system software is to convert all instructions into machine understandable language.

The different computer languages are:

1. Machine level Language
2. Assembly level Language
3. High level Language

1. Machine level Language

- Machine Language is the only language that is directly understood by the computer.
- The program code is written in terms of 1's and 0's
- It does not need any translator program.
- We also call it machine code and it is written as strings of 1's (one) and 0's (zero).
- When this sequence of codes is fed to the computer, it recognizes the codes and converts it in to electrical signals needed to run it.
- For example, 0010101111000011

Advantages of Machine Language:

1. The advantage is that program of machine language run very fast.
2. No translator is required.

Disadvantages of Machine Language:

1. Programs written in this language are machine dependent
2. It is very difficult to program in machine language. The programmer has to know details of hardware to write program.
3. The programmer has to remember a lot of codes which are machine specific.
4. It is difficult to debug the program.
5. It is not an easy language for programmers to learn because it is difficult to understand
6. It is efficient for the computer but very inefficient for programmers.

2. Assembly level Language

Assembly level language contains Symbolic English like words to specify instructions .These symbols are known as ***mnemonics***

```
Ex:   MOV AL, 3
      ADD BL
```

Advantages of Assembly level Language

1. Programs written in this language are more readable than machine language

Disadvantages of Assembly level Language:

1. Programs cannot be directly executable they require ***assemble***r to translate programs to machine language
2. Programs are machine *dependent*. A program written for one computer might not run in other computers with different hardware configuration.

3. High level Language

It is a type of computer language that uses English words plus symbols and numbers to express the program code.

Advantages of High level Language

1. It is close to human language
2. It is easy to understand
3. It consists of English language like structure
4. It does not depend upon machine
5. It is easy to modify

Disadvantages of High level Language:

- It needs a language translator called ***compiler*** to translate program in to machine language.
- It does not execute directly on computer.

Device Drivers:

- System software that controls and monitors functioning of a specific device on computer is called device driver. Example printer, scanner, speaker etc..
- When a new device is attached you need to install a device driver so that the OS knows how it needs to be managed.

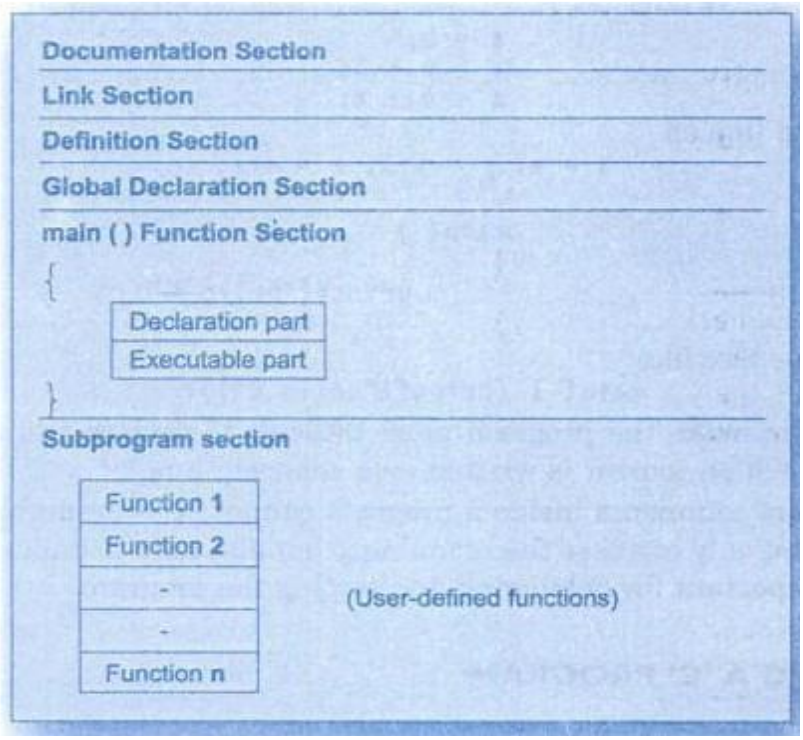
Application software:

Software that performs a single task and nothing else is application software, it is specialized in their function, example word processing, multimedia tools etc..

Utility software:

Application software that assists system software in doing work is called utility software. It is actually a cross between a system software and application software.

BASIC STRUCTURE OF C PROGRAM



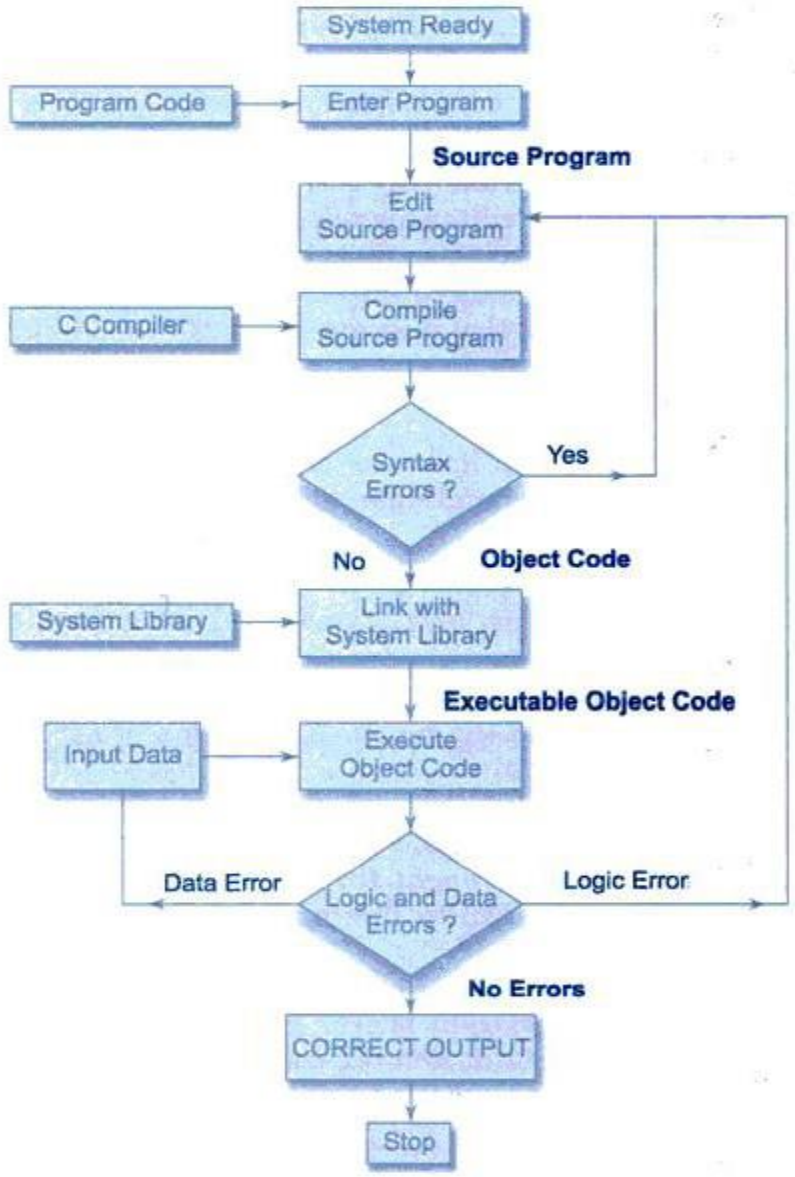
- The **documentation section** consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
- The **link section** provides instructions to the compiler to link functions from the system library.
- The **definition section** defines all symbolic constants.
- There are some variables that are used in more than one function. Such variables are called ***global* variables** and are declared in the *global* declaration section that is outside of all the functions. This section also declares all the **user-defined functions**.
- Every C program must have one **main () function section**.
    - This section contains two parts, **declaration part and executable part**.
    - The **declaration part** declares all the variables used in the executable part.
    - There is at least one statement in the **executable part**.
    - These two parts must appear between the opening and the closing braces.
    - The program execution begins at the opening brace and ends at the closing brace.
    - All statements in the declaration and executable parts end with a semicolon (;).

EXECUTING A C PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program;
2. Compiling the program;
3. Linking the program with functions that are needed from the C library; and
4. Executing the program.

The below figure illustrates the process of creating, compiling and executing a C program.



Creating the program

- Once we load the UNIX operating system into the memory, the computer is ready to receive program.
- The program must be entered into a file.
- The file name can consist of letters, digits and special characters, followed by a dot and a letter c.
- Examples of valid file names are: hello.c , program.c, ebgl.c.
- The file is created with the help of a text editor, either ed or vi. The command for calling the editor and creating the file is

  vi filename

Compiling and Linking

- The source program has been created in a file named program.c.
- The compilation command to achieve this task under UNIX is

  cc program.c
- The source program instructions are now translated into a form that is suitable for execution by the computer.

- The translation is done after examining each instruction for its correctness.
- The translated program is stored on another file with the name program.o. This program is known as **object code.**
- **Linking** is the process of putting together other program files and functions that are required by the program. For example, if the program is using exp() function, then the object code of this function should be brought from the math library of the system and linked to the main program.
- The linking is automatically done (if no errors are detected) when the cc command is used.
- If any mistakes in the syntax and semantics of the language are discovered, they are listed out and the compilation process ends right there.
- The errors should be corrected in the source program with the help of the editor and the compilation is done again.
- The compiled and linked program is called the **executable object code** and is stored automatically in another file named a.out.

Executing the program

- Execution is a simple task.
- The command

./a.out

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard.

Creating Your Own Executable File

The linker always assigns the same name a.out. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

mv a.out name

Multiple Source Files

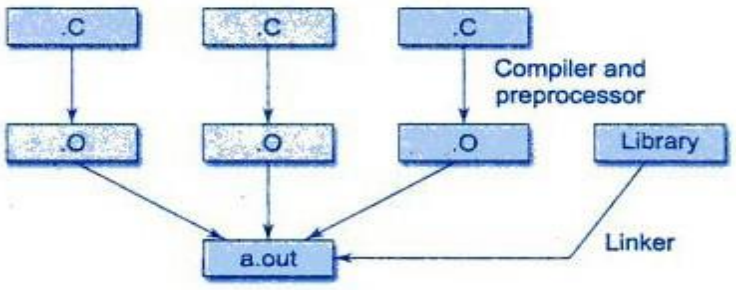
- To compile and link multiple source program files, we must append all the files names to the cc command.

cc filename-l.c,…… filename-n.c

These files will be separately compiled into object files called

filename-i.o

and then linked to produce an executable program file a.out as shown below figure.



CHARACTER SET

- Characters that can be used to form words, numbers and expressions depend on the computer on which the program is run.
- The characters in C are grouped in to the following categories :
  1. Letters 2. Digits 3. Special Characters 4. White Spaces
- Compiler ignores white spaces unless they are a part of a string constant.

- The entire character set is as shown in below table

Letters	*Digits*
Uppercase A.....Z Lowercase a.....z	All decimal digits 0 9

Special Characters

, comma	& ampersand
. period	^ caret
; semicolon	* asterisk
: colon	– minus sign
? question mark	+ plus sign
' apostrophe	< opening angle bracket
" quotation mark	(or less than sign)
! exclamation mark	> closing angle bracket
| vertical bar	(or greater than sign)
/ slash	(left parenthesis
\ backslash	) right parenthesis
~ tilde	[left bracket
_ under score	] right bracket
$ dollar sign	{ left brace
% percent sign	} right brace
	# number sign

White Spaces

Blank space
Horizontal tab
Carriage return
New line
Form feed

TRIGRAPH CHARACTERS

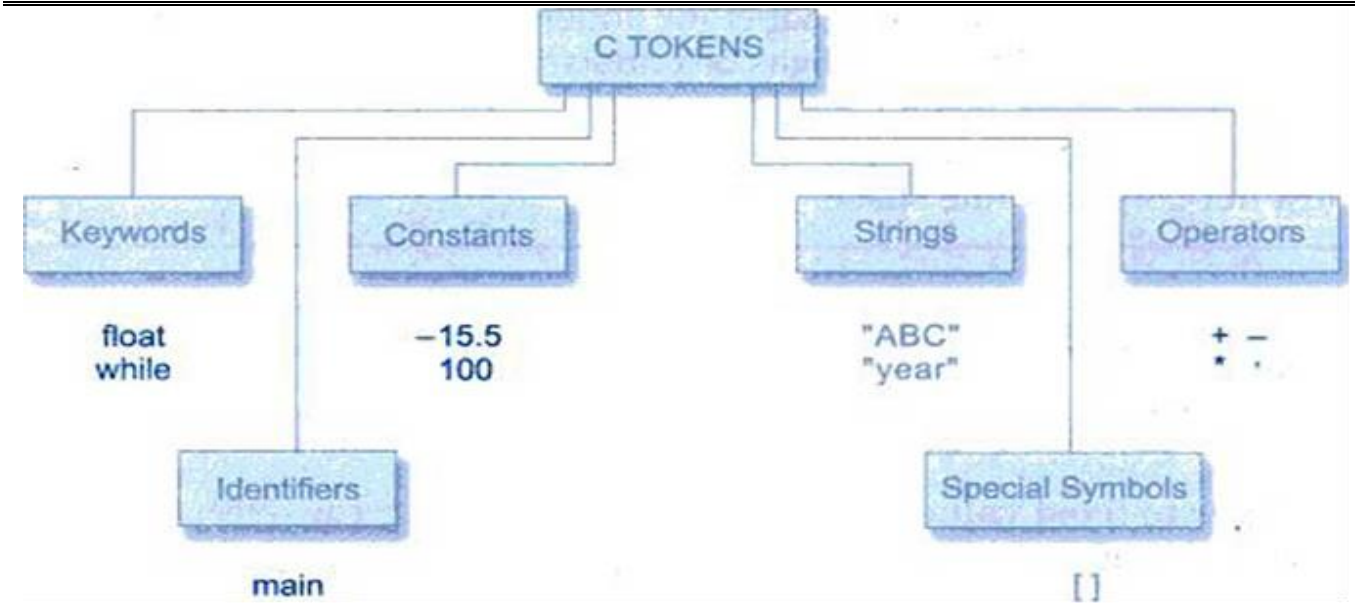
- ANSI C introduces the concept of "trigraph" sequences to provide a way to enter certain characters that are not available in some keyboards.
- Each trigraph consists of 3 characters (two question marks followed by another character).
- Ex: if a keyboard does not supports opening square brackets ([ ), we can still use them using trigraphs ??(and for closing square brackets ( ]) we can use ??).

Table 2.2 *ANSI C Trigraph Sequences*

Trigraph sequence	*Translation*
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vetical bar
??/	\ back slash
??/	^ caret
??-	~ tilde

C TOKENS

- In C program smallest individual units are known as C tokens.
- C has 6 types of tokens as shown in below figure.
- C programs are written using these tokens and the syntax of language.



KEYWORDS:

- Keywords are the words which has fixed meaning and that cannot be changed by user. Hence, they are also called ***reserved-words***.
- They should be written in ***small*** letters only
- There are totally ***32*** keywords

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

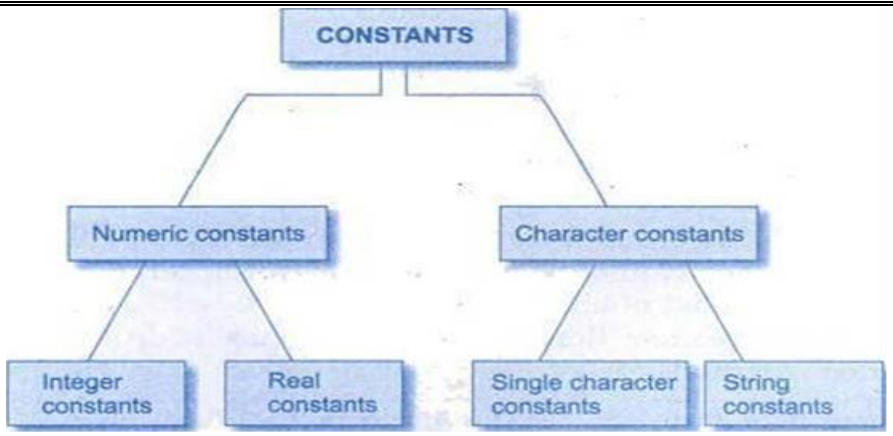
IDENTIFIERS: An identifier is a name given to variables, constants, functions, arrays etc. These are userdefined names. An identifier can be made up of alphabet, digits and underscores.

Rules:

- The first character in the identifier should be a letter or alphabet(uppercase or lower case) or an _ (underscore)
- No other symbols other than "_" can be used
- The length of an identifier can be up to a maximum of *32* characters
- *Keywords* cannot be used as identifiers
- White space is not allowed

Valid Identifiers	**Invalid identifiers**	**Reason for invalidity**
max	if	keyword
pi	3sum	Started with numeric digit
Reg_no	Sum-of-digits	Minus not allowed
SUM3		

CONSTANTS: Constants in C refer to the fixed values that do not change during the execution of the program. C supports several types of constants as shown in figure



Integer Constants:

- An Integer constant refers to a sequence of digits.
- There are 3 types of integers namely decimal integer,octal integer and hexa decimal integers.
    - **Decimal Integers:** It consist of set of digits,0 through 9,preceded by an optional – or + sign. Ex: 123,-123, 0, +78, 123456.
    - **Octal Integers**: It consists of any combination of digits from 0 through 7 , with a leading 0. Ex: 037,0,0435,0551.
    - **Hexadecimal Integers:** Sequence of digits preceded by 0x or 0X is considered as hexadecimal integers. They may include alphabet A through F or a through f. These alphabets reprints number 10 through 15. Ex : 0X2,0x9F,0XBCD etc.

Real Constants:

- The numbers containing fractional part are called real constants. Ex: 17.35,0.00086,-0.56,+247.0 etc. These numbers are shown in decimal notation, having a whole number followed by decimal point and the fractional part.
- Real numbers are also valid when digits before and after decimal points are omitted. Ex: 215. , .95, +.71, -.5 etc...
- Real number may also be represented in exponential form also. Ex: 215.65 may be written as 2.1565e2 in exponential notation.e2 means multiply by 10^2.
- General form is

  mantissa e exponent

  where mantissa is either a real number expressed as decimal or integer. Exponent is an integer with optional + or – sign. Letter 'e' can be written either in lower or upper case.

Examples of Numeric constants

Constant	*Valid ?*	*Remarks*
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Single Character Constant :

- A Single character constant contains a single character enclosed within a pair of single quote mark. Ex : '5','x' ':',' '
- Character constants have integer values known as ASCII values.
  Ex: printf("%d",a); will print the ASCII value of 'a'. printf("%c",97); will print the letter a, as 97 is the ASCII value of 'a'.

String Constants:

- A string constant is a sequence of characters enclosed in double quotes. Characters may be letters, numbers and blank space. Ex: "HELLO!","5+3","?...!" etc ..
- A character constant (ex:'X') is not equivalent to the single character string (ex: "x").

Backslash Character Constant:

- C supports some special backslash constants that are used in output functions
- Each backslash represents one character, although they consist of two.
- These combination of characters are called **escape sequences**.
- Ex :

Constant	*Meaning*
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

VARIABLE: A variable is an identifier. It can change its value for each execution of the program. All variables must be declared before their usage.

Ex. ***sum*** –is a variable stores the result of addition of 2 or more numbers.

A variable considered with its **type**, its **value** and its **address**.

Rules to be followed to use variables

1. The variable name must begin with a letter and can have digits and underscore
2. A variable name can be or 32 characters
3. keywords cannot be used as variable names
4. White spaces cannot be used.

Some examples are as given below: Value3, S1, ph_value, I_rate

Invalid variables names are 23rd, 355, (rate)

DECLARATION OF VARIABLES: assigning memory to a variable is known as declaration of variable.

Primary Type Declaration: All C variables must be *declared* as follows:-

datatype variable1, varaible2…;

Ex. int i ; Here ***i*** is a variable which can hold integer type data

Ex. char a, b, ch ; a, b, ch are character type data, they hold one character

User-Defined Type Declaration:

- C supports a feature known as type definition that allows users to define an identifier that would represent an existing data type.
- User defined data types can be used to declare variable
  typedef **type identifier** ;
  where type refers to existing data type and identifier refers to new name given to the data type
- Ex**:** **typedef int** units;
  typedef float marks;
  here units symbolizes int and marks symbolizes float.

Assigning values to Variables: The method of giving values to the variables is called assigning.

Syntax: - datatype **var-name = value;**

Ex:
```
char ch = 'a' ;          //a is stored in ch
double d = 12.2323 ;     // 12.2323 is stored in d
int j = 20 ;             //20 is stored in j
```

Declaring variable as constant:

- The values of certain variables to remain constant during execution of program.
- We can do this by declaring the variable with qualifier **const** during the time of initialization.
- Ex**: const int** class_size = 40 ; This tells the compiler that value of int variable **class_size** must not be modified in the program.

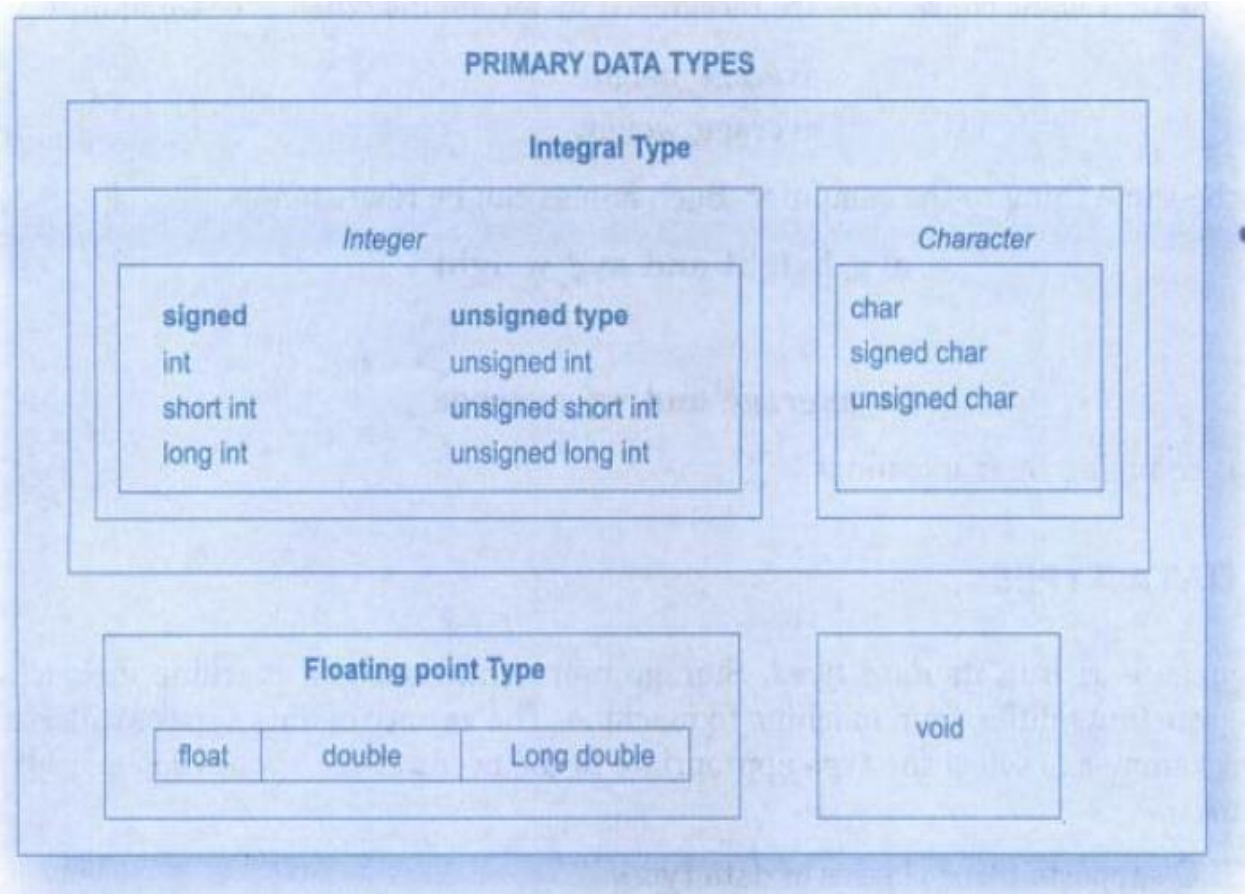
Declaring a variable as volatile:

- ANSI C defines another qualifier **volatile** that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources.
- Ex: **volatile int** date

DATA TYPES

- C Language is rich in its data types
- Storage representations and machine instructions to handle constants differ from machine to machine.
- Verity of data types allows programmers to select the type appropriate to the needs of application as well as the machine.
- ANSI C supports 3 classes of data types :
  1. Primary (or functional) data type
  2. Derived data types
  3. User-defined data types

- All C compilers support 5 fundamental data types, namely integer (int), character (char) , floating point (float) , double-precision floating point (double) and void. Many of them also offer extended data types such as long int and long double.
- Ranges of 4 basic type of data types are as shown in table.



Size and Range of Basic Data Types on 16-bit Machines

Data type	*Range of values*
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

Integer Types:

- Integers are whole numbers supported by a particular machine
- Usually integer occupies one word of storage, since word sizes varies from machine to machine(16 bi or 32 bit).size of integer is limed to -32768 to +32767.similarly for 32 bit word length integer range varies from -2.147,483,648 to 2,147,483,647.
- C has 3 classes of integer storage classes: short int , int , and long int both in signed and unsigned forms.

Type	Size (bits)	Range
char or signed char	8	–128 to 127
unsigned char	8	0 to 255
int or signed int	16	–32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	–128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	–2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E – 38 to 3.4E + 38
double	64	1.7E – 308 to 1.7E + 308
long double	80	3.4E – 4932 to 1.1E + 4932

Size and Range of Data types on a 16-bit Machine

Floating point types:

- Floating point numbers are defined in C by keyword **float** and are stored in 32 bits.
- When the accuracy provided by float is not sufficient, the type **double** is used and it uses 64 bits giving a precision of 14 digits.
- Long double can be used which uses 80 bits.

Void Types:

- The void type has no value and is usually used to specify the type of functions.
- The type of function is said to be void when it does not return any value to the calling function.

Character Types:

- A single character can be defined as **character (char)** type data and is usually stored in 8 bits of internal storage.
- Signed characters have values from -128 to 127 and unsigned from 0 to 255.

INPUT-OUTPUT FUNCTIONS:

These functions are used to ***input(read)*** data into variables or ***output (write)*** data of the variables.

1. Reading data from keyboard: The input functions that accepts data from the keyboard and stores data into variables.

scanf(): The only formatted input function is ***scanf***. It is used for formatted input. The general syntax of scanf function is

scanf("formatspecifier/s", &variable1,&variable2..)

- The scanf function takes format specifiers, this specifies the ***datatype*** of the variable , format sepcifiers are enclosed within " ". The different format specifiers are

  %c for character data type

  %d for integer data type

%f for float data type

- And a comma follows variable or variable list.**&** is used with all variables to indicate the reading value is stored at the address where the variable is stored in the memory.

```
Ex: int i, a ;
char b ;
float c ;
scanf( "%d", &i ) ;
scanf( "%d %c %f", &a, &b, &c
```

2.**Writing data on to the Monitor:** This function is similar to the scanf function except that it is used for formatted output.

The only formatted output function is printf. It is used for formatted output. The general syntax of scanf function is

printf("formatspecifier/s", variable1, variable2..)

- The printf function takes format specifiers, this specifies the ***datatype*** of the variable , format sepcifiers are enclosed within " ". The different specifiers are as specified with scanf and
- A comma follows variable or variable list. **&** is not used

```
For Example :-
int i = 15 ;
float f = 13.3576 ;
printf( "%3d", i ) ; /* prints    15 with 3 white spaces */
printf( "%6.2f", f ) ; /* prints "_13.36" which has a total width of 6 and displays 2 decimal
places */
```

Operators: An ***operators*** is a symbol that specifies the operates on various types of ***operands***. An expression has ***operands*** and ***operators***.

Ex. s=a+b; here s,a,b are ***operands*** and + and = are ***operators***.

The values upon which the operator operates are called ***operands***. The operators in C are classified into following types based on number of operands

1. Unary operators
2. Binary operators

1. Unary operators: These are the operators that operate on only one *operand.* The following are unary operators in C.

a. **+ unary plus operator** : indicates +ve value. Ex: +7,+23.
b. **- unary minus operator** : indicates -ve value. Ex: -7,-23.
c. **Increment operator**:
   - This operator makes the value of the variable to be increment by 1.

- It is indicated as ++
- Two types of increment are *pre increment* and *post increment*

Ex. int a=10;

a++; is equivalent to a=10
a=a+1(10+1) makes the value of a to be incremented by 1 and a becomes 11

++a; is equivalent to a=a+1 (10+1)
a=11 makes the value of a to be incremented by 1 and a becomes 11

d. **Decrement operator**:

- This operator makes the value of the variable to be decrement by 1.
- It is indicated as --
- Two types of decrement are *pre decrement* and *post decrement*

Ex. 1. int a=10;

a--; is equivalent to a=10
a=a-1 (10-1) makes the value of a to be decremented by 1 and a becomes 9

--a; to a=a+1 (10-1)
a=9 makes the value of a to be decremented by 1 and a becomes 9

e. **! logical complement operator:** Inverts the Boolean value(true to false and false to true).

2. Binary Operators:

These operators act on 2 *operands*. The different types of binary operators are :

1. Arithmetic Operators [+,-,*,/,%]
2. Relational Operators [>,<,>=,<=,==,!=]
3. Logical Operators [&&,||,!]
4. Assignment Operators [=]
5. Compound shorthand assignment operators [+=,-=,*=,/=,%=]
6. Bitwise Operators [&,|,^,~]
7. Conditional operators [? :]
8. Shift operators [>>,<<]
9. Special operators [(,),sizeof, . And ->, * and &]

1. Arithmetic operators:

- They used to perform arithmetic operations and are +,-, *, /, %
- Ex. sum=a+b, sub=a-b, mul=a*b, div=a/b

2. Relational Operators:

- The relational operators are used to test the ***relation between*** two entities.
- The relational operators return true or false value depending upon the condition
- They also return *zero or non zero* values. **0** values is taken as ***false*** and ***non-zero*** is taken as ***true***.

- These operators allow to compare two values, If they are equal, unequal, if one is greater to other.
- The different relational operators in C are <, >, <=, >=, !=, ==

Ex. 1. a=5, b=3 a==b is False

2. a=3, b=3 a==b is True

3. a=5, b=6 a>b is false

3. Logical Operators:

- We can connect the results of multiple expressions or logical operations by using logical operators.
- The logical operators are !(NOT), &&(AND), ||(OR)
- The **negation** takes only one operand and returns a false value if the operand is true and vice versa

Ex. a=2, b=3 !(a>b) result is **true,** because a>b ,2>3 results into **false** , the negation of false is **true**.

The **AND** operator returns a value 1 if both expressions are **true**
Ex. a=4, b=5, and c=7 (a>b) && (a>c) is false because both a>b and a>c are false

The OR operator returns 1 or true if either one of the expression is true.
Ex. (a>b) ||(c>b) is true because c>b is true

4. Assignment Operator: This operator is used to assign a value to a variable. Which is ' = '.The right side result or value will be assigned to the left side variable of the operator

- Ex. i=10; assigns the value 10 to i.

5. Compound shorthand assignment operators Operators

- Many C operators can be combined with the assignment operator as shorthand notation
  For Example :-
  x = x + 10 ; can be replaced by x += 10 ;
- x = x - 10 ; can be replaced by x -= 10 ;
- x = x * 10 ; can be replaced by x *= 10 ;
- x = x / 10 ; can be replaced by x /= 10 ;
- x = x % 10 ; can be replaced by x %= 10 ;

6. Bit-wise operators: The bit wise operator operates on a single bit. There are 4 bitwise operators in C

& (AND) operator: this *compares* each bit of the first operand with the corresponding bit of the second operand ,. If both bits are 1 , then the corresponding bit is set to 1 else it is set to 0.

Ex: **0 0 0 0 1 0 1 0**
0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 0

Op1	Op2	result
0	0	0
0	1	0
1	0	0
1	1	1

| (OR) operator: this operator compares each bit of the first operand with the corresponding bit of the second operand , if either of the bit is

Ex: **0 0 0 0 1 0 1 0**
0 0 0 0 0 1 1 0
0 0 0 0 1 1 1 0

Op1	Op2	result
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise X-OR (^) operator : If the corresponding bit positions in both the operands are different , then ex-OR results in 1 else 0.

Ex: **0 0 0 0 1 0 1 0**
0 0 0 0 0 1 1 0
0 0 0 0 1 1 0 0

Op1	Op2	result
0	0	0
0	1	1
1	0	1
1	1	0

The complement operator ~: The complement operator inverts the value of each bit of the operand.if the operand bit is 1 the result is 0 and if the operand bit is 0 the result is 1.

Ex: **0 0 0 0 0 1 1 0**
1 1 1 1 1 0 0 1

Op1	result
0	0
1	1

7. Conditional operator: The combination of ?and : is called as Conditional operator denoted by (? :). It is also called as ternary operator since it takes 3 operands.

Syntax : expression1? expression2: expression 3

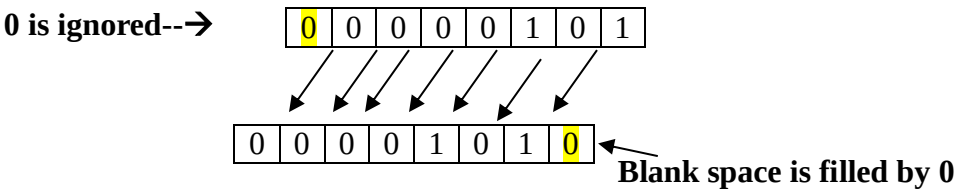
first expression1 is evaluated, if it is true then expression2 is evaluated, expression1 is the result of the expression and if expression1 is false then expression3 is evaluated and expression3 is the result of the expression.

Ex: a=30, b=50,result;

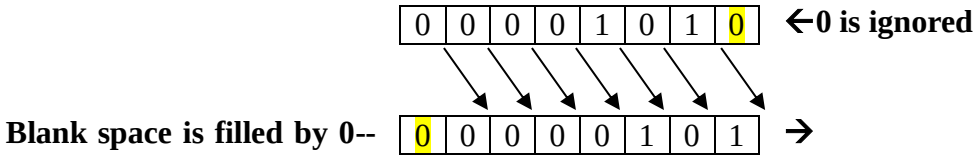
result=a>b?a:b; greatest of a and b and assigns it to result*/

8. Shift operators: A shift operator performs bit manipulation on data by shifting the bits of its operands right or left.

Left Shift operator(<<): This operator is used to shift each bit positions towards left. The blanks that are created on the left will be filled by zero



Right Shift operator: This operator is used to shift each bit positions towards right. The blanks that are created on the left will be filled by zero.



9. Special Operators:

- **Comma operator:** It is used to combine multiple expressions into a single expression.
- Ex. x=(a=12,b=22,a+b); the value of the rightmost expression 34 is assigned to x.
- **Sizeof()-**operator-it is used to determine the size of the operand
- **Ex.** int x=2000; sizeof(x)); /*x=2 as it is integer and takes 2 bytes */
- Member selection operators(. and ->)
- Pointer operators(* and &)

TYPE CONVERSION:

The process of converting data from one data type to another data type is called type conversion.
Two types of conversion are
1. Implicit conversion
2. Explicit conversion

1. Implicit conversion: The compiler itself converts data from one type to another and the conversion takes from lower datatype to higher datatype. This conversion is shown below

char ⟶ ***int*** ⟶ ***long int*** ⟶ ***float*** ⟶ ***double***

Ex. 1. int y=5;
float x;
x=y; gives the result x=5.0, integer 5 will be automatically converted to float

2. Explicit conversion: The user forcefully converts one data type to another data type is called explicit conversion. The data conversion from ***higher data type*** to ***lower data type*** is possible by using ***explicit conversion***.

Ex.
float y=5.1;
int x;
x=(int)y; gives the result x=5, float 5 .1 will be forcefully converted to int

EXPRESSIONS:

- An *expression* is combination of operands and operator that reduces to a single value.
- Ex. C=A+B is a mathematical expression with A,B and C as operands and + and = are operators
- The expressions are evaluated according to the ***priority*** and ***associativity.***

Conversion of mathematical expressions to C –expressions: The mathematical expressions are to be written in C-program notation while programming

MATHEMATICAL EXPRESSION	C-REPRESENTAITON
A= $\frac{5x+3x}{a+b}$	A=(5*x+3*y)/(a+b)
B = $\frac{\sqrt{s(s-a)(s-b)(s-c)}}{2a}$	B=sqrt(s*(s-a)*(s-b)*(s-c))/(2*a)
C = $e^{\lvert x+y-10\rvert}$	C=exp(abs(x+y-10))
D = $X^{25} + Y^{35}$	D=pow(x,25)+pow(x,35)
E = $\frac{e^{\sqrt{x}} + e^{\sqrt{y}}}{x\sin\sqrt{y}}$	E=(exp(sqrt(x))+exp(sqrt(y)))/(x*sin(sqrt(y)))
$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	X=(-b+sqrt(b*b-4*a*c))/(2*a)

Precedence and Associativity: In C programming the expressions are evaluated according to precedence and associativity.

Precedence: The priority in which the operator is evaluated in an expression are called ***precedence*** **or** ***hierarchy of operators***.
The below table shows the operator's ***precedence***.

Operators	Order of precedence	Associativity
Brackets	[ ] or ( )	L⟶R
Unary	++, -- ! sizeof	R⟶L
Arithmetic	* /%	L⟶R
Arithmetic	– +	L⟶R
Shift	<< >>	L⟶R
Relational	< >= <= >	L⟶R
Bitwise	^ |	L⟶R
Logical	&& ||	L⟶R
Conditional	? :	R⟶L
Assignment	=, +=,-=,*=	R⟶L
Comma operator	,	L⟶R

Associatively: associativity of an operator is a property that determines the order of evaluation of a operator in an expression.

Two types of associativity are:

- Left Associativity: meaning the operators are evaluated from the left to right.
  Ex: a+b;
- Right Associativity: meaning the operators are evaluated from the right to left.
  Ex: x=a+b;

ASSIGNMENT QUESTIONS:

1. Explain in brief
   a. "History of C".
   b. Importance of C
   c. Generation types
   d. Computer Types
2. What is a Computer? Write a Block diagram of a computer and explain in brief.
3. What is Memory? Explain different types of memory.
4. Explain the steps involved for executing a C program.
5. Explain the structure of C program with an example.
6. What are data types? Mention the different data types supported by C Language, giving an example to each.
7. Define & explain with an example
   a. Variable
   b. Constant
   c. Associativity
   d. Precedence
   e. Bit-Wise Operator
   f. Conditional Operator
8. Write a program in C and draw flowchart to find
   a. The area and perimeter of a circle.
   b. Addition of 3 numbers and average of it
   c. Simple Interest
9. What is a token? What are different types of tokens available in C language? Explain.
10. Explain different types of ports in a computer.
11. List and explain different types of input and output devices.
12. Write a note on
    a. Computer in a network
    b. Network hardware
    c. Router
13. Explain in brief language processor and character set.
14. What is a variable? Explain the declaration and initialization of variables.Write C equivalent expression for the following.

    a. $C = e^{|x+y-10|}$ b. $E = \frac{e^{\sqrt{x}}+e^{\sqrt{y}}}{x\sin\sqrt{y}}$ c. $D = X^{25} + Y^{35}$

    d. $B = \frac{\sqrt{s(s-a)(s-b)(s-c)}}{2a}$

15. What is type-conversions? Explain two types of type conversion along with an example.
16. Explain I/O functions.
17. What is an operator? Explain the arithmetic, relational, logical, and assignment operators in C language.
18. What is the value of x in the following code segments? Justify your answer.

    i.
    ```
    int a,b;
    float x;
    a=4;
    b=5;
    x=b/a;
    ```

    ii.
    ```
    int a,b;
    float x;
    a=4;
    b=5;
    x=(float)b/a;
    ```

****

Module 2

Branching and Looping

Managing Input and output operations. Conditional Branching and Loops. Example programs, Finding roots of a quadratic equation, computation of binomial coefficients, plotting of Pascal's triangle.

MANAGING INPUT-OUTPUT OPERATIONS:

These functions are used to ***input (read) data*** into variables or ***output (write)*** data of the variables. The different types of I/O statements are

1. Formatted I/O: The functions that use format specifiers are called as formatted I/O function.
2. Unformatted I/O: The functions that does not use format specifiers are called as unformatted I/O function.

1. Formatted Input functions: The input functions that accepts data from the keyboard and stores data into variables.

scanf(): The only formatted input function is ***scanf***. It is used for formatted input. The general syntax of scanf function is

scanf ("format specifier/s", &variable1,&variable2..)

- The scanf function takes format specifiers, this specifies the ***datatype*** of the variable , format specifiers are enclosed within " ". The different format specifiers are

  %c for character data type
  %d for integer data type
  %f for float data type
  %e (%E) float or double exponential format
  %o int unsigned octal value
  %s array of char sequence or string
  %u int unsigned decimal
  %x (%X) int unsigned ***hex*** value

- And a comma follows variable or variable list.**&** is used with all variables to indicate the reading value is stored at the address where the variable is stored in the memory.

```
Ex: int i, a ;
char b ;
float c ;
scanf( "%d", &i ) ;
scanf( "%d %c %f", &a, &b, &c );/* e.g. typing   10  x 1.234  stores 10 in a, x in b
```

```
and 1.234 in c*/
scanf( "%d:%c", &i, &c ) ; /*  10 : x   stores 10 in i after : x is stored in c  */
```

printf(): This function is similar to the scanf function except that it is used for formatted output. The format specifiers have the same meaning as for printf() and the *space* or the *newline*(\n or \t) characters can be used as delimiters between different inputs.The general syntax of printf function is

printf("formatspecifier/s", variable1, variable2..)

- The printf function takes format specifiers, this specifies the ***datatype*** of the variable , format sepcifiers are enclosed within " ". The different specifiers are as specified with scanf and
- A comma follows variable or variable list. **&** is not used

```
For Example :-
int i = 15 ;
float f = 13.3576 ;
printf( "%3d", i ) ; /* prints    15 with 3 white spaces */
printf( "%6.2f", f ) ; /* prints "_13.36" which has a total width of 6 and displays 2
decimal places */
```

2.Unformatted I/O

1.Getch(),getche(),getchar() and **putchar():** Getch(),getche(),getchar() functions are used to input single ***character***. They require no format specifier

- **Getch ():** reads a single character from the keyboard and stores into a character variables. It doesn't display the character on the screen and return the key pressed.
  Syntax: <character variable> = getch() ;

  Ex: char ch1 ;
  ch1 = getch() ;

- **Getche ():**reads a single character from the keyboard and stores into a character variables. It displays the character on the screen and return the key pressed.
  Syntax: <character variable> = getche() ;

  Ex: char ch1;
  ch1 = getche ();

- ***getchar*():** reads a single character from the keyboard and stores into a character variables. It accepts the input until the carriage returns is entered.
  syntax: <character variable> = getchar() ;

  Ex: char ch1 ;
  ch1 = getchar() ;

- ***putchar*()** displays a character on the standard output device i.e. on to the screen.

  Ex :-ch2 = 'a' ;
  putchar(ch1) ; or printf("%c", ch1);
  putchar(ch2);

2. gets () and **puts():**gets() accepts string including spaces from the standard Input device (keyboard). gets() stops reading character from keyboard only when the *enter key* is pressed

Syntax: gets(variable_name);
Ex: gets(str);

puts() displays a single / paragraph of text to the standard output device.

Syntax: puts(variable_name);
Ex: puts(str);

CONDITIONAL BRANCHING AND LOOPS

Statements: single C- language instruction is called a statement.

Control statements: statements normally execute sequentially. "The control statements will define the order in which individual statements or instructions of a program are executed or evaluated.

The control structures are divided into the following categories

- Conditional statements or selection.
- Looping or Iterative statements
- Jumping statements

Conditional Statements: The conditional statement in a program allows the user to select one or more execution paths in a program. If the conditional expression results ***true*** then the specified statements will be executed otherwise(if the result is ***false***) the statements are bypassed

The different types of conditional statements supported by C are

- If –statement (single selection)
- If-else statement (two way selection)
- Nested –if statement
- If-else ladder (multiway selection)
- Switch statement (Multiway selection)
- Nested switch statements

The if-statement:

Syntax: if(expression)
false
{

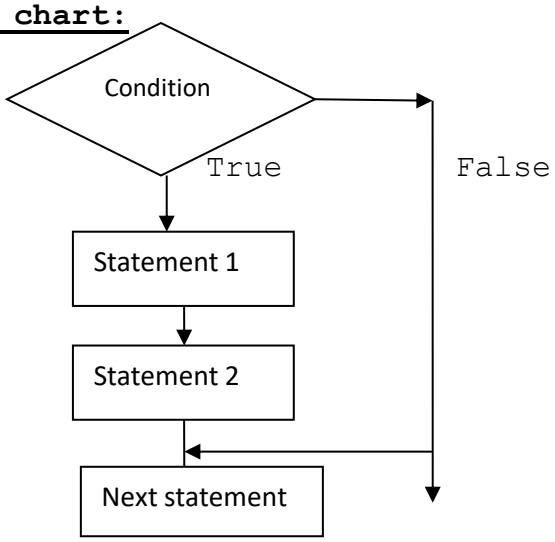
Statement 1;

Statement 2;

}

Next statement;

flow chart:



- If statement is a single selection/decision statement .
- When the expression inside if is evaluated to true then the statement1, statement 2 will be executed. If it is false then the control moves to next statement.

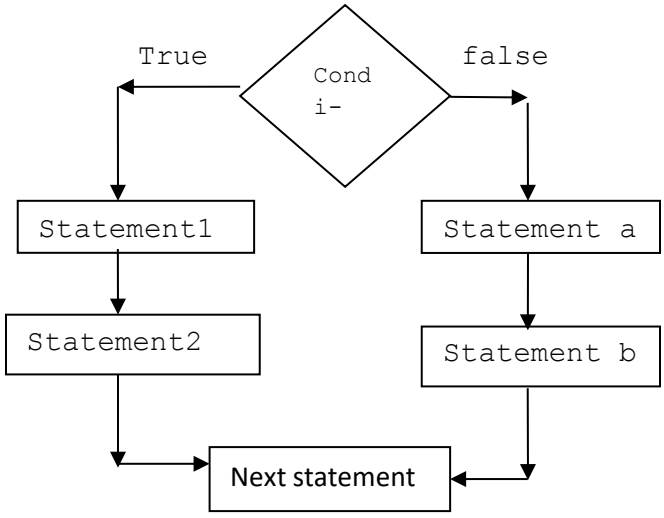
Ex. main( )
{
int age;
printf("enter the age");
scanf("%d",&age);
if(age<18)
printf("You are eligible to vote");
}

If-else statement:

- If –else *statement* is used to choose either one alternative among two set of alternatives. i.e when the expression is evaluated to false, then if –else statement is used.
- If the expression is evaluated to **true** then statements 1, 2 will be executed and statement a, b will not execute
- If the expression is evaluated to **false**, and the statements a, b will be executed. And statements 1, 2 will not execute
- It is also called two way decision statement.. the syntax is as shown,

Syntax:

```
  if(condition)

{ Statement1;
 Statement 2;
}
else
{ Statement a;
 Statement   b;
}
```



Ex**. 1.Program to find greatest of two numbers**

```
 main()
{
int a, b;
printf("Enter two numbers");
scanf("%d%d",&a,&b);
if(a>b)
printf("a is greater");
else
printf("b is greater");
}
```

2.Program to check whether the given number is odd or even

```
main()
{
int n;
printf("Enter the number");
scanf("%d",&n);
if(n%2==0)
printf("n is even number");
else
printf("n is odd number");
}
```

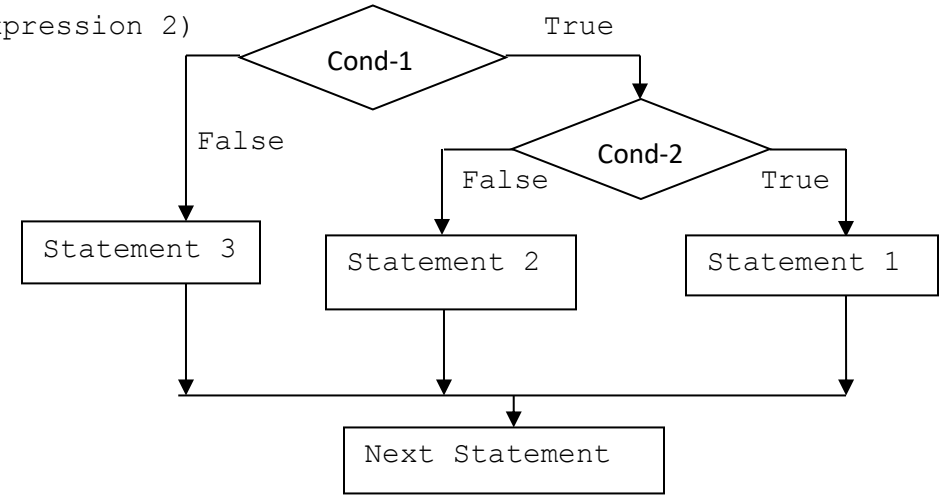
Nested-if statement

- Nested `if` means one if statement contains another `if` statement.
- When the outer `if` statement is evaluated to be `true` the control of a program moves into the inner `if` statement.
- The syntax of the `if` statement is as described below,

Syntax:

```
if (conditional expression 1)
       if(conditional expression 2)
            Statement 1;
       else
            Statement 2;
 else
       statement 3;
```

flow chart:



Ex. if(income>20000)
if(sex='m')
tax=income*10.0;
else
tax=0;

in the above code, tax is evaluated only for male employees,

The if -else ladder

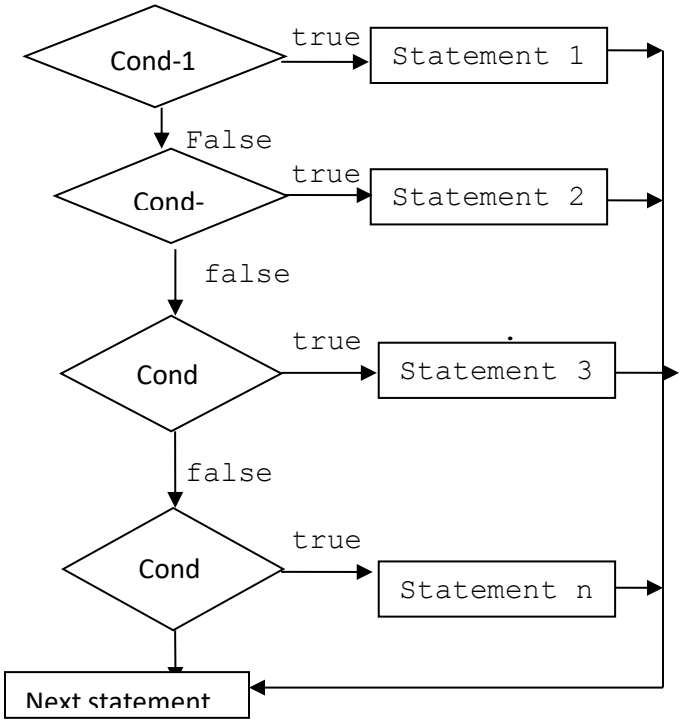
- An if –else ladder is a special case of nested if statements where nesting take place only in the else part.
- It is used when an action has to be selected based on range of values, this statement is used. So it is called *multiway selection statement.*

Ex. Quadratic equation

Syntax:

```
if(condition)
{ Statement1;
  }
else if(condition-2)
{statement 2;
}
 else if(Condition-3)
{ Statement 3;

}

else if(condition n)
{
statement n;
}

Next statement;
```

flow chart:



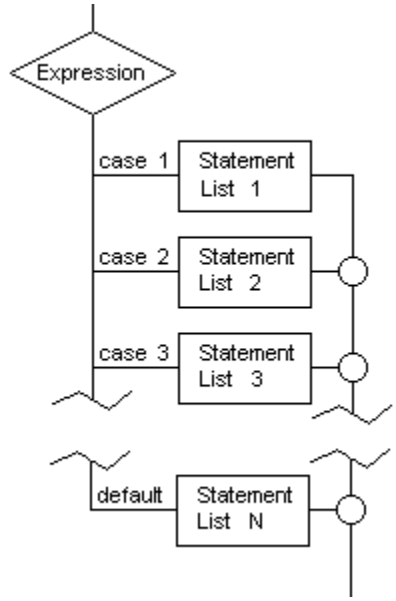
Program to find quadratic equation

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

void main()
{
        int a,b,c;
        float root1, root2, realp, imgp, disc;
        printf("Enter three co-efficients of the quadratic equation\n");
        scanf("%d%d%d",&a,&b,&c);
        if(a*b*c==0)
        {
                printf("\nRoots can not be found");
                getch();
                exit(0);
        }
        disc = b*b-4*a*c;
        if(disc==0)

        {
                printf("\nRoots are equal\n");
                root1 = root2 =-b/(2.0*a);
                printf("\nRoot1 = Root2 = %.2f", root1);
        }
        else  if(disc>0)
        {
                printf("\nRoots are real and distinct");
                root1 = (-b + sqrt(disc))/(2*a);
                root2 = (-b - sqrt(disc))/(2*a);
                printf("\nRoot1 = %.2f", root1);
                printf("\nRoot2 = %.2f", root2);
        }
        else
        {
                printf("\nRoots are imaginary");
                realp = -b/(2.0*a);
                imgp = sqrt(fabs(disc))/(2*a);
                printf("\nRoot1 = %.2f + i %.2f",realp, imgp);
                printf("\nRoot2 = %.2f - i %.2f",realp, imgp);
        }
}
```

The switch statement

- The switch statement is a control statement used to make a selection between many alternatives.
- Based on the choice (integer or character value) the control is transferred to a particular case value.
- *Switch cannot be used when the selection is based on a range of values.*
- The syntax of switch is as described below,

Flow chart



Syntax:

switch(choice/expression)
{
case value1:
Block1;
break;
case value 2:
Block2;
break;
case value 3:block3;
break;
case value n:
default:
Block d;
}

Program to simulate the functions of a simple calculator

```
#include <stdio.h>
int main()
{
int num1,num2;
float result;
char choice; //to store operator choice
printf("Enter first number: ");
scanf("%d",&num1);
printf("Enter second number: ");
scanf("%d",&num2);
printf("Choose operation to perform (+,-,*,/,%): ");
scanf(" %c",&choice);
result=0;
switch(choice)
        {
            case '+':
                        result=num1+num2;
                        break;
             case '-':
```

```
                result=num1-num2;
                break;
        case '*':
                result=num1*num2;
                break;
        case '/':
                result=(float)num1/(float)num2;
                break;
        case '%':
                result=num1%num2;
                break;
        default:
                printf("Invalid operation.\n");
    }
  printf("Result: %d %c %d = %f\n",num1,ch,num2,result);
  return 0;
}
```

Looping or Repetitive statements (Counter control Loops)

- These are the statements that helps to execute a set of statements repeatedly.
- The statements within a loop may be executed for a fixed number of times or until condition fails.

The different types of looping statements in C language are, *while, do-while* and *for*

while loop:

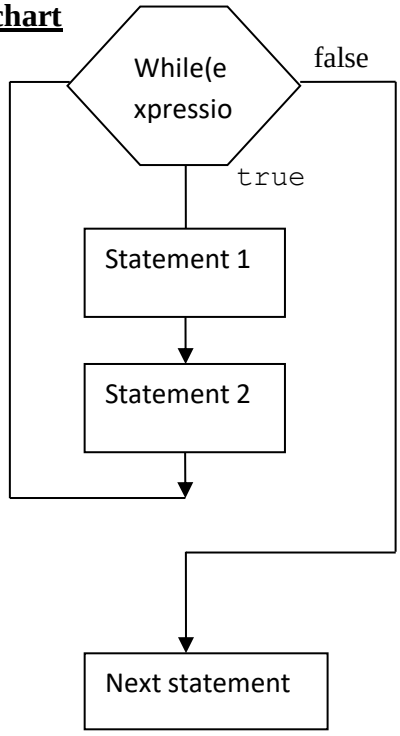
- It executes set of statements repeatedly based on some condition untill the condition fails.
- It is called *condition controlled* or *event controlled* loop.
- It is a ***pre*** tested loop statement, where the condition is checked before entering into the loop.
- The syntax for while loop is as shown below, it starts with a keyword **while**, the expression after the keyword **while** must be enclosed within parentheses. It evaluates to ***true*** or ***false.***
- If the expression evaluates to true the body of the loop is executed.
- After executing the body of the loop, control goes back to beginning of the while – statement and expression is again evaluated to true of false.
- The body of the while loop is repeatedly executed as long as the expression evaluates to true. Once this expression becomes false the control comes out of the loop.

Syntax:

while(expression)
{
statement 1;
statement 2; **//body of the loop**

}
next statement;

flow chart



Program to reverse a given integer number.

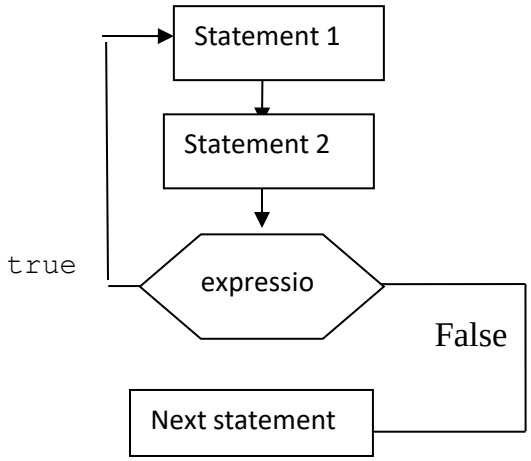
```
#include<stdio.h>
#include<conio.h>

void main()
{
        int temp, n, rev=0;
        clrscr();
        printf("\nEnter the value of n:");
        scanf("%d", &n);
        temp=n;
        while (n>0)
        {
                rev = rev*10+n%10;
                n = n/10;
        }
        printf("\nThe Reverse of the number %d = %d", temp, rev);
        if ( temp == rev)
                printf("\n It is a Palindrome");
        else
                printf("\n It is not a Palindrome");

}
```

do-while loop:

- do-while is similar to while loop, this is used when we do not know exactly how many times a set of statements have to be repeated it is called ***post tested*** loop.
- Here the body of the loop executed first then the expression is evaluated.
- If the expression evaluates to true, then the loop repeats, otherwise, it terminates.

```
do
{
statement 1;ststemetn 2;
body of the loop

}while(expression);
```

flow chart



Difference between while-loop and do-while loop:

while loop	**do-while loop**
syntax	syntax
1.while() 1.	do
{	{
Stat-1;	stat-1;
Stat-2	stat-2;
…	….
…	…
Stat-3;	stat-3
}	}while(exp);
2. It is *top*-testing	2. It is *bottom* testing
3. It is *pre* tested loop	3. It is *post* tested loop
4. if, *while* evaluates to false no Statements of loop will execute	4. Even if,*while* evaluates to false the loop statements evaluates *at least once*
5.*while* loop ends with no **;**	5. *do while* ends with **;**
6.*while* has **no** *do* word	6. *do while* loop has the reserve word *do*
7.It is *entry* controlled loop	7. It is *exit* controlled loop
8. Ex. to print 1, 2,3….n the code is as follows	8. Ex. to print 1, 2,3….n the code is as follows
while(i<=n)	do
{	{
printf(“\n%d”,i);	printf(“\n%d”,i);
i++;	i++;
}	}while(i<=n);

for loop:

- *for* loop is also called *counter controlled* loop .since it is required to specify how many times a set of statements have to be executed at the beginning .
- Once the specified number of times the loop executed the program control comes out of the loop. The syntax of for loop is,

syntax:

```
for(initialization; conditional expression; post expression)
{
        simple or compound statement;
}
```

- The initialization part initializes the variable with a value. It executes only once.
- This is followed by conditional expression. If the conditional expression evaluates to *false*, the program control comes to the statement next to the for loop.
- If the conditional expression evaluates to *true* then the body of the for loop is evaluates.
- The post expression is used to increment or decrement a counter variable controlling a loop
- The for loop continues to execute until the conditional expression evaluates to false.
- The body of for loop may contain a single statement or compound statement.

Program to find factorial of a given number

```
void main()
{
      int n, f=1,i;
      printf("enter the  number  \n");
      scanf("%d",&n);
      for(i=1;i<=n;i++)
      f=f*i;
      printf("\n the factorial of %d is %d\n",n, f);
}
```

Program to print 1, 2,3….n

```
void main()
{
      int n,i;
      printf("Enter the number");
      scanf("%d",&n);
      for(i=1;i<=n;i++)
       {
            printf("\n%d",i);
       }
   }
```

Un-conditional branch statements (jump statements):

- These are the statements used to transfer the control from one statement to other statement in the program without any condition
- They are ***goto, break, continue***

Jumping out of a loop:

Goto statement : using this statement control can be transferred from one statement to the specified statement without any condition,

The syntax of goto is

goto label; label is any identifier

Ex.
```
void main ()
{
Printf (“1”);
Printf (“2”);
Printf (“3”); goto end;
Printf (“4”);
Printf (“5”);
end: printf(“End”);
}
```

break statements:

- A break statement is jump statement which can be used in *switch* statement and loops.
- The break statement in switch causes the control to terminate switch statement.
- If break appears in loop. The control comes out of the loop.
- The statements following *break* are skipped and the execution continues from that point

Ex.

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

or

```
for(i=1;i<5;i++)
{
printf(“%d” i);
break;
printf(“skipped statement”);
}
```

The output of this program is 1,2

The output of this program is 1

<u>Skipping Part of a loop</u>

Continue statement:

- This statement is used only in the loop to terminate the current iteration.
- When the continue statement is executed, the statements following *continue* are skipped and but the execution of the iteration still continues.

Ex.

```
for(i=1;i<=5;i++)
{
if(i==3)
continue;
printf("%d",i);
}
```

Output: The output of this program is 1,2,4,5

or

```
for(i=1;i<5;i++){
printf("%d" i);
continue;
printf("skipped statement");
}
```

Output: The output of this program is 1,2,3,4

COMPUTATION OF BINOMIAL COEFFICIENTS

Problem: Binomial coefficients are used in the study of binomial distributions and reliability of multicomponent redundant systems. It is given by,

$$B(m,x) = \binom{m}{x} = \frac{m!}{x!(m-x)!}, \quad m >= x$$

A table of binomial coefficients is required to determine the binomial coefficient for any set of m and x.

Problem Analysis: The binomial coefficient can be recursively calculated as follows:

$$B(m,o) = 1$$

$$B(m,x) = B(m,x-1)\left[\frac{m-x+1}{x}\right], \quad x = 1,2,3,\ldots,m$$

Further,

$$B(o,o) = 1$$

That is the binomial coefficient is one when either x **is zero or m is zero. The** program below prints the table of binomial coefficients for **m** = **10. The program employs one do loop** and **one while loop.**

Program

```
#define MAX 10
main()
{
        int m, x, binom;
        printf ("m x").
        for (m=0;m<=10;++m)
         printf("%4d", m);
```

```
        printf("\n----------------------\n");
        m = 0;
        do
        {
                printf("%2d ", m);
                x = 0; binom = l;
                while (x <= m) '
                {
                        if(m == 0 || x == 0)
                        printf("%4d", binom);
                        else
                        {
                                binom = binom * (m - x + 1)/x;
                                printf("%4d", binom);
                        }
                        x=x+1;
                }
                printf("\n");
                m = m + 1;
        }
        while (m <= MAX);
        printf("----------------------------------------");
}
```

Output

mx	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

Program to print binomial coefficient table

PLOTTING OF PASCALS TRIANGLE

There 2 different source codes in C program for Pascal's triangle, one utilizing function and the other without using function. Both of these program codes generate Pascal's Triangle as per the number of row entered by the user.

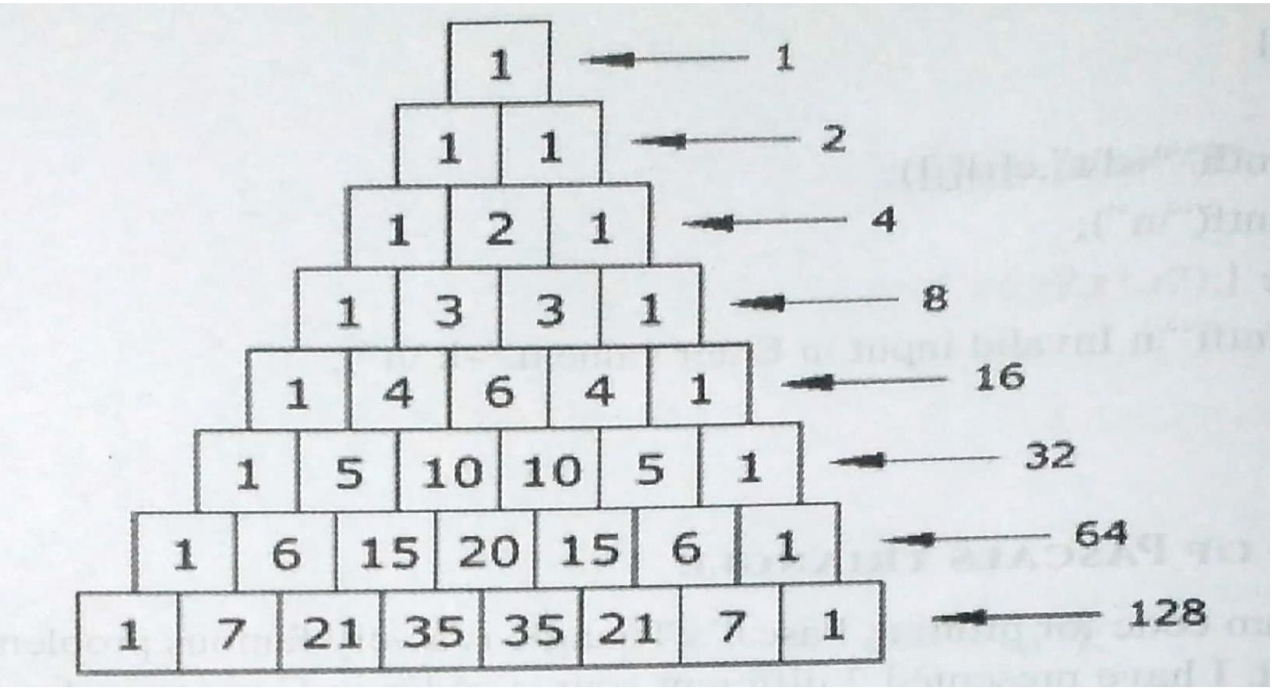
The construction of the triangular array in Pascals triangle is related to the binomial coefficients by Pascals rule.

How to Build Pascal's Triangle?

- In Pascals triangle all the numbers outside the triangle are '0's' , to build the triangle start with a '1' at the top, continue putting numbers below in a triangular pattern so as to form a triangular array.
- So, each new number added below the top '1' is just the sum of two numbers above, except for the edge which are all '1's'

This can be summarized as:
0 row =1
1 row = (0+1), (1+0) = 1, 1
2 row = (0+1), (l + 1), (1+0) = 1, 2, 1
3 row = (0+1), (1+2), (2+1), (1+0) = 1, 3,3, 1
4 row = (0+1), (1+3), (3+3), (3+1), (1+0) = 1, 4, 6, 4, 1

Properties of Pascal's Triangle



- The sum of all the elements of a row is twice the sum of all the elements of its preceding row. For example, sum of second row is 1+1= 2, and that of first is 1. Again, the sum of third row is 1+2+1 =4, and that of second row is 1+1 =2, and so on. This major property is utilized to write the code in C program for Pascal s triangle.
- The sequence of the product of each element is related to the base of the natural logarithm e.
- The left and the right edges of Pascal's triangle are filled with "1" s only.
- All the numbers outside the triangle are "0"s.
- The diagonals next to the edge diagonals contain natural numbers (1,2,3,4,….) in order.
- The sum of the squares of the numbers of row "n" equals the middle number of row "2n".

```
#include<stdio.h>
int main()
{
        int num, rows, cols,spaces,ans;
        printf("Enter number of rows ");
        scanf("%d", &num);
        for(rows=0; rows<num; rows++)
        {
                for(spaces=1 ;spaces<=num-rows;spaces++)
                printf(" "); //              printf("**");
                for(cols=0;cols<=rows; cols++)
                {
                        if(cols=0 || rows=0)
                                ans=l;
                        else
                                ans= ans*(rows-cols+l)/cols; printf("%4d",ans);
                }
                printf("\n");
        }
        return 0;
}
```

ASSIGNMENT QUESTIONS

1. What is two way selection statements? Explain ***if, if-else, nested if, if-else if*** and ***cascaded if else(if else ladder)*** with Examples and syntax

2. Write a program that takes 3 co-efficient a, b ,c of a quadratic equation $ax^2+bx+c=0$ and compute all possible roots and print with the appropriate messages.

3. Explain ***Switch*** statement with example

4. Explain ***jumping statements with example***.

5. What is a loop? Explain the different looping statements in C language.

6. Write a program,

- To calculate square root of a number without using library function.
- To read year as input and to find whether it is a leap year or not.
- To reverse a given number and to check whether it is a palindrome or not.
- C program find factorial of a number.
- To develop a simple calculator for 2 operands using switch
- C program to find greatest of three numbers.

7. Differentiate between ***while*** and ***do-while*** statements.

8. Explain in brief binomial co-efficient with a C program.

9. Explain in brief Plotting of Pascals Triangle with a C program.

8. Write a program to calculate sin(x) value using the Taylor's series

$$\sin(x) = x - \frac{x3}{3!} + \frac{x^5}{5!} - \cdots$$

****

Arrays: Arrays (1-D, 2-D), Character arrays and Strings, Basic Algorithms: Searching and Sorting Algorithms (Linear search, Binary search, Bubble sort and Selection sort).

Arrays:

- An array is a collection of *similar* set of elements.
- An array is a collection of *same* type of values stored in *consecutive* memory locations.
- The values in the array are referred by a ***name***,(which is the name of the array) and ***index***, this indicates the position of the value in the array.
- The different types of arrays are
    1. ***Single –dimensional array.***
    2. ***Two-dimensional array***
    3. ***Multidimensional arrays***.

SINGLE DIMENSIONAL ARRAYS:

A ***single dimensional array*** is a linear list of related data items with a single subscript. In memory all the data items are stored in contiguous memory locations one after the other.

Declaring one dimensional array: Declaration of array is similar as declaring ordinary variables in program. An array is to be declare before using it. Array is declared by using the following syntax,

Syntax: Data_type array_name[size];

In the above line, datatype represents the data type of the array to be declared, array_name represents the name of the array and size represents the number of values to be sorted in the array.

Ex: **int a [10];**
in this example, a is the name of the array contains 10 elements, which are indexed 0 to 9. Which contain 10 integer values

The elements in one dimensional array are stored starting at the index zero to one less than the size of the array. The total amount of memory that can be allotted to the array can be calculated using the formula

Total memory=size*[sizeof[data-type)]
= 10*2 bytes=20 bytes.

Initializing one dimensional array: initializing array is similar to initializing ordinary variables.
The various types of array initialization are,

- One by one initialization
- Initializing in a single statement.
- Initializing using for loop and scanf ().

1. **One by one initialization:**
   We can initialize the elements of the array one by one.
   Ex: Int a[5];
   a [0]=10; a [1]=20; a [2]=30; a [3]=40; a [4]=50;

2. **Initializing in a single statement:** We can initialize all the elements of the array at the time of declaration of the array. The syntax to initialize array is as follows,

   Array_name [index] =value;

Array_name indicated the name of the array, and *index* represents the position of the array element in the array and the *value* represents the value being assigned to the array.

Ex. int a[5]={10,20,30,40,50}; this specifies initializing all the array elements at the same time.

Initialization

int a[5]={10,20,30}; only 3 elements are initialized. Even though the compiler reserves 5 spaces only 3 are initialized the next two will automatically initialized to **0.**

Initialization using for loop and scanf():

The elements of the array can be read by input device(keyboard) by using for loop and scanf().

```
for(i=0;i<n;i++)
scanf("%d",&a[i]);
```

Limitations of arrays:

- The size of the array specified at the time of creation cannot be altered later. Or we *cannot add more elements* to an array than specified in its size.
- It is difficult to insert an element between two existing elements.
- The amount of memory taken by an array depends on the size specified for the array and not on the number of elements stored in the array.

Program to find the *largest or maximum* element in the array.

```
void main()
{
int i,n,big=0,a[10];
printf("enter the size of the array");
scanf("%d",&n);
printf("Enter the elements of the array :\n");
for(i=0;i<n;i++)
Scanf("%d",&a[i]);              /* reads the array elements*/
for(i=0;i<n;i++)
  {
        if(a [i]>big)           /* checks each element of array with big element*/
     big=a [i];
   }
printf("The biggest element in the array is=%d",big);
}
```

TWO DIMENSIONAL ARRAYS:

Array with two subscript is called two-dimensional array. It is used to store a table of values of the same data type. It has two subscripts rows and columns.

Declaring a Two dimensional array:

data_type array_name [size1][size2];

In the above syntax data_type represents the data type of the array to be declared, array_name represents the name of the array, size1 represents the number of *rows* and size2 represents the number of *columns*.

Ex. int a[4][5];

Here, the array a is a two dimensional array containing 4 rows and 5 columns

Initializing a Two–dimensional Array:

Initializing all specified memory locations:
The elements of a two dimensional array can be initialized similar to a one –dimensional array. The syntax for this is as follows.

data_type array_name[size1][size2]={value1,value2,……value n};

Ex. int a[2][3]={2,3,4,5,6,7}; //This is an array with 2 rows and 3 columns,

The above initialization is similar to ,
a[0][0]=2; a[0][1]=3; a[0][2]=4;
a[1][0]=5; a[1][1]=6; a[1][2]=7;

OR

```
int a[2][3]={{2,3},
             {4,5},
             {6,7}};
```

Initialization using for loop and scanf():

The elements of the array can be read by input device(keyboard) by using for loop and scanf().

```
for(i=0;i<m;i++)
        for(j=0;j<n;j++)
scanf("%d",&a[i][j]);
```

Partial array initialization

```
int a[3][3]={{2,3},
             {4,5},
             {6,7}};
```

This array has the size 3X3 with only 6 elements, the remaining locations will be filled with 0s

2	3	0
4	5	0
6	7	0

Program to find the transpose of a given matrix

```
void main()
{
        int i,j,m,n;
        int a[4][4],b[4][4];
        printf("Enter the number of rows :\n");
        scanf("%d",&m);
        printf("Enter the number of columns :\n");
        scanf("%d",&n);

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                scanf("%d",&a[i][j]);
                printf("the transpose of the  matrix  is\n");
                for(i=0;i<m;i++)
                {
```

```
                    for(j=0;j<n;j++)
                    {
                        b[i][j]=a[j][i];
                        printf("%d\t",b[i][j]);
                    }
                    printf("\n");
                }
        }
}
```

Develop, implement and execute a C program that reads two matrices A (m x n) and B (p x q) and Compute product of matrices A and B.

```
#include<stdio.h>
void main()
{
        int A[5][5], B[5][5], C[5][5], m,n,p,q,i,j,k;
        printf("Enter the order of matrix A");
        scanf("%d%d",&m,&n);
        printf("Enter the order of matrix B");
        scanf("%d%d",&p,&q);
        if(n!=p)
        {
                printf("\n Matrix multiplication is not possible");
                getch(); exit(0);
        }
        printf("\n Matrix multiplication is possible\n");
        printf("\n Enter %d elements of matrix A \n",m*n);
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                        scanf("%d",&A[i][j]);
        }
        printf("\n Enter %d elements of matrix B\n", p*q);
        for(i=0;i<q;i++)
        {
                for(j=0;j<p;j++)
                        scanf("%d",&B[i][j]);
        }
        for(i=0;i<m;i++)
        {
                for(j=0;j<q;j++)
                {
                        c[i][j]=0;
                        for(k=0;k<n;k++)
                                C[i][j]+= A[i][k]*B[k][j];
                }
        }
        printf("The Product Matrix C is\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<q;j++)
```

```
                printf("%d\t",C[i][j]);
            printf("\n");
        }

}
```

Develop, implement and execute a C program that reads two matrices A (m x n) and B (p x q) and Compute addition of matrices A and B.

```
#include<stdio.h>
void main()
{
        int A[5][5], B[5][5], C[5][5], m,n,p,q,i,j,k;
        printf("Enter the order of matrix A");
        scanf("%d%d",&m,&n);
        printf("Enter the order of matrix B");
        scanf("%d%d",&p,&q);

        printf("\n Enter %d elements of matrix A \n");
        for(i=0;i<m;i++)
        {
            for(j=0;j<n;j++)
                scanf("%d",&A[i][j]);
        }
        printf("\n Enter %d elements of matrix B\n");
        for(i=0;i<q;i++)
        {
            for(j=0;j<p;j++)
                scanf("%d",&B[i][j]);
        }
        for(i=0;i<m;i++)
        {
            for(j=0;j<q;j++)
            {
                c[i][j]=0;
                    C[i][j]= A[i][j]+B[i][j];
            }
        }
        printf("The Addition of Matrix C is\n");
        for(i=0;i<m;i++)
        {
            for(j=0;j<q;j++)
                printf("%d\t",C[i][j]);
            printf("\n");
        }

}
```

Note: Perform subtraction and division same as above.

Multi-Dimensional Array

C allows arrays of 3 or more dimensions. The exact limit is determined by the compiler. The general form of a mult**i-dimensional array is**

data_type array_name [s1] [s2] [s3]…….[sm]

where s_i is the size of the i^{th} dimension.

Some examples are

int **survey [3]** [5] [12];
float table [5] [4] [5] [3];

Dynamic Arrays:

- **An array** created **at compile** time **by specifying si**ze in the source code has a **fixed** size and cannot **be** modified at run-time.
- The process of a allocating memory at compile time is known as **static memory** allocation.
- In C it is possible to allocate memory to arrays at run time are called dynamic memory allocation and the arrays created at run time are called dynamic arrays.
- Dynamic arrays are created using hat are known as pointer variables and memory management functions malloc, calloc and realloc.

1. **malloc( ):**this function allocates specified amount of memory . The syntax is as shown

Syntax: datatype pointer_vatriable;
Pointer variable=(casting*)malloc(n*sizeof(datatype));

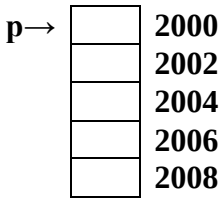
- *pointer_ variable* is of any data type
- **casting** specifies the allocated memory is to cleared to store the specified type of data
- **malloc()** function allocates the memory
- `sizeof()`- Returns size in bytes
- `n`-specifies the number of elements to which memory to be allocated

Ex.

```
int *p;
p=(int*)malloc(5*sizeof(int));
```

Here **p** is pointer variable holds the address of first memory location returned by ***malloc ()*** to hold ***5*** integer numbers.

Ex.



2. **calloc( ):** this function is same as **malloc() ,it allocates a block of memory.** but it initializes memory locations with 0 and returns pointer to first memory location

syntax :datatype pointer_vatriable;pointer variable=(casting*)calloc(n*sizeof(datatype));

- *pointer_ variable* is of any data type
- **casting** specifies the allocated memory is to cleared to store the specified type of data

- **calloc()** function allocates the memory
- `sizeof()`- Returns size in bytes
- `n`-specifies the number of elements to which memory to be allocated

Ex.

```
int *p;
p=(int*)calloc(5,sizeof(int));
```

Here **p** is pointer variable holds the address of first memory location returned by ***calloc ()*** to hold ***5*** integer numbers.

p→	**0**	**2000**
	0	**2002**
	0	**2004**
	0	**2006**
	0	**2008**

3. **realloc( )** : this function reallocates the memory that is already allocated either by **malloc** or **calloc.**This function helps to ***alter*** the memory size by *extending* or *shrinking* the allocated memory.

```
int *p;
p=(int*)malloc(5*sizeof(int));
```

p→		**2000**
		2002
		2004
		2006
		2008

when the memory to be increased so another 6 numbers can be stored, then ***realloc*** is used as shown below

```
p=(int*)realloc(11*sizeof(int));
```

and allocated memory is as shown,

p→		**2000**
		2002
		2004
		2006
		2008
		2010
		2012
		2014
		2016
		2018
		2020

4. free() : function is used to deallocate the allocated memory either by ***malloc*** or ***calloc*** or ***realloc***

syntax: `free(pointer_variable);`

if the above example is considered then

`free(p);` releases the memory

CHARACTER ARRAYS AND STRINGS

Strings:

String: An array of characters is called a *string.*
In C language string is an array of characters and terminated by NULL character which is denoted by the escape sequence '\0'

Declaring strings: There is no separate data type for strings in C . They treated as arrays of type character**.**
Ex. char a[80]; here the variable **a** can hold maximum of 80 characters including a Null (\0) character.

Initializing strings: arrays can be initialized in the following ways

1. **char** a[9]={'c','o','m','p','u','t','e','r'};- the compiler allocates 9 memory locations ranging from 0 to 8 and these locations are initialized with the characters in the order specified. The remaining locations are automatically initialized to null-characters as shown below

   0 1 2 3 4 5 6 7 8

   b-

c	o	m	p	u	t	e	r	\0

2. **char** a[10]={'r','a','m','a'};

r	a	m	a	\0	\0	\0	\0	\0

3. **char** b[ ]="computer";

4. **char** s[10];
   scanf("%s", s);

String Handling Functions: The built in string handling functions are

1. strlen(s)- returns length of the string s.
2. strcpy(d,s)- copies the source string s to destination string d
3. strcat(s1,s2)-append s2 to s1
4. strrev(s)-reverses the string s
5. strcmp(s1,s2)-compares two strings s1 and s2 , returns **0** if both strings are equal

All these string functions are defined in header file *"string.h"*

1 . strlen(str)- This function returns length of the string *s* . It takes a string syntax of this is

int strlen(char s[]);

Ex. program to find length of a string

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s[]="RAMA";
        printf("Length =%d",strlen(s));
}
```

Ex. program to find length of a string without using string function.

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

```
        char s[]=”RAMA”;
        int i=0;
        while(s[i]!=’\0’) i++;
        printf(“Length =%d”,i);
}
```

2. strcpy(d,s)- This function copies the source string s to destination string d

Ex.
```
#include<stdio.h>
#include<string.h>
void main()
{
        char s[]=”RAMA”, d[];
        strcpy(d,s)
        printf(“Destinatin string =%s”,d);
        d[]=RAMA;
}
```

Destination string d gets the value RAMA

Ex. program to copy a string to another without using string function.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s[20],d[10];
        int i=0,j;
        printf(“Enter the first string”);
        gets(s);
        while(s[i]!=’\0’)
        {
                d[i]=s[i];
                i++; j++:
        }
        d[i]=’\0’;
        printf(“The destination  string is =%s”,d);
}
```

If entered string is RAMA then ***d*** copied with RAMA

3.strcat(s1,s2)- This function joins two strings into one string or append s2 to s1. This function copies all the characters of strong s2 to the end of string s1.

Ex.

V	I	V	E	K	A			S1

R	A	M	A					S2

After strcat(S1,S2) and length of S1 is increased

S1	V	I	V	E	K	A	R	A	M	A	\0

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[10];
        printf("Enter the first string");
        gets(s1);
        printf("Enter the second string");
        gets(s2);
        strcat(s1,s2);
        printf("The concatenated string is =%s",s1);
}
```

Program to implement string concatenation without using strcat function

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[10];
        int i=0,j;
        printf("Enter the first string");
        gets(s1);
        printf("Enter the second string");
        gets(s2);
        while(s1[i]!='\0')
        i++;
        j=0;
        while(s2[j]!='\0')
        {
                s1[i]=s2[j];
                i++; j++;
        }
        s1[i]='\0';
        printf("The concatenated string is =%s",s1);
}
```

4.strrev(s)-reverses the string s. This function reverse the given string s

Ex.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s[]="RAMA";
        printf("reverse of string is =%s",strrev(s));
        }
```

S is

I	N	D	I	A	\0		

After strrev(s)

S is

A	I	D	N	I	\0		

C program to check Palindrome or not using string

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20], s2[20];;
        int i,n;
        printf("Enter the string");
        gets(s1);
        strcpy(s2,s1);
        strrev(s2);
        if(strcmp(s1,s2)==0)
        printf("the given string  %s    is palindrome",s1);
        else
        printf("the given string  %s   is not palindrome",s1);
}
```

Reversing of string without using strrev

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20], s2[20];;
        int i,n;
        printf("Enter the first string");
        gets(s1);
        n=strlen(s1);
        for(i=0;i<n;i++)
        s2[n-1-i]=s1[i];
        s2[n]='\0';
}
```

SEARCHING AND SORTING ALGORITHMS

Searching: Searching is the process of finding the ***key*** value in the array, If the value is found we display its position or else we display the searching element or key is not found in the array

Two types of searching are

- **Linear search** :In this type of searching key being compared with each element in the array with starting to ending position
  It is a simple method

  Ex: in this array if key=5 , it will first check a[0] i.e 8==5 no match so next it compares with a[1] i.e 2==5 No match it proceeds to compare with a[2], a[3], a[4], a[5]etc., as the match found at a[7] the position is displayed

8	0
2	1
4	2
6	3
11	4
3	5
9	6
5	7

Program to find the *given key* element in the array using *LINEAR search*.

```
void main()
{
int i,n,key,a[10];
printf("enter the size of the array");
scanf("%d",&n);
printf("Enter the elements of the array  :\n");
for(i=0;i<n;i++)
Scanf("%d",&a[i]);
printf("Enter the element to be search  :\n");
Scanf("%d",&key);
for(i=0;i<n;i++)
if(a[i]==key)    /*checks each element of array with key element*/
        {
                printf("the given element is found at  %d", i+1);
               break;
        }
        if(i==n)
        printf("%d is not found in the array", key);
}
```

ALGORITHM

PURPOSE: Implement Linear search
INPUT: n, a[50], key
OUTPUT: key is present at the position or key not found

START

STEP 1: [Read no of elements]
Read n

STEP 2: [Read the elements of the array]
For←i=0 to n
Read a[i]
End i^{th} for loop

STEP 3: [Read the elements to be searched]
Read key

STEP 4: [checks each element of array with key element)
if ←(a[i]==key)

STEP 6: [Check if key is not found]
If not found
Print key not found

STEP 11:[Finished]
STOP

- **Binary search**:
  - Binary Search is applied on the sorted array or list.
  - In binary search, we first compare the value with the elements in the middle position of the array.
  - If the value is matched, then we return the value.
  - If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array.
  - We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.
  - If key is 35 first it is compared with middle value 11 , as 35 is greater to 11, the low part of the array will be checked for 35 and middle will be calculated by using mid=(low+high)/2

2	0
4	1
6	2
8	3
11	4
23	5
29	6
35	7

Program to find the *given key* element in the array using *BINARY search*.

```
void main()
{
        int i,n,key,a[10],m,l,h;
        printf("enter the size of the array");
        scanf("%d",&n);
        printf("Enter the elements of the array in ascending order  :\n");
        for(i=0;i<n;i++)
        Scanf("%d",&a[i]);
        printf("Enter the element to be search  :\n");
        Scanf("%d",&key);
        low=0;
        high=n-1;
        while (l<=h)
        {
          mid=(low+high)/2;
          if (key ==a[mid])
                    {
                              printf("%d found at location %d.\n", mid+1);
                             break;
                    }
                     if (key <a[mid]) high=mid-1;
                    if (key>a[mid]) high=mid+1;

         }

                    printf("%d is not found in the array", key);
}
```

ALGORITHM

PURPOSE: Implement Binary search
INPUT: n, a[50], key
OUTPUT: key is present at the position or key not found

START

STEP 1: [Read no of elements]
Read n

STEP 2: [Read the elements of the array]
For←i=0 to n
Read a[i]
End i^{th} for loop

STEP 3: [Read the elements to be searched]
Read key

STEP 4: [Initiliazation]
low = 0;
high = n-1;

STEP 5: [Check whether low is less than of equal to high and calculate mid)
while ←low <=high
mid= (low+ high)/2;

STEP 6: [Check if mid is equal to key)
If a[mid] is equal to key

STEP 7: [print the key position]
Print key is present at the position mid+1

STEP 8: [check if mid is greater than key and calculate]
If mid is greater than key
high = mid-1;

STEP 9: [Else]
low = mid+1;

STEP 10: [Check if key is not found]
If not found
Print key not found

STEP 11:[Finished]
STOP

Sorting: Sorting is a process of arranging elements in *ascending* or *descending* order.

Bubble Sort: Bubble sort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.

```
#include<stdio.h>
int main()
{
        int n,temp,i,j,a[20];
        printf("Enter total numbers of elements: ");
        scanf("%d",&n);
        printf("Enter %d elements: ",s);
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
                                //Bubble sorting algorithm
                for(i=0;i<=n-1;i++)
                for(j=0;j<=n-1-i;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }
                }
 printf("After sorting: ");
 for(i=0;i<n;i++)
 printf(" %d",a[i]);
 return 0;
}
```

ALGORITHM

PURPOSE: Arranging the numbers in ascending order using bubble sort technique
INPUT: N, interger numbers in arrary a[i]
OUTPUT: Numbers are arranged in ascending order

START
STEP 1: [Input number of elements]
Read n

STEP 2: [Input the elements/numbers into array]
For i←0 to n
Read a[i]
End for

STEP 3 : [Sorting the elements in ascending order]
For i ←0 to n-1
For j←0 to n-1

[Compare the adjacent elements]
If(a[j]>a[j+1]) then

[Swap these elements]
Temp←a[j]
a[j]←a[j+1]
a[j+1]←temp
end if
end for
end for

STEP 4: [Display the sorted elements of array]
For i←0 to n
Print a[i]
End for

STEP 5: [Finished]
STOP

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

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

void main()
{
   int a[100],n,i,j,min,temp;
   clrscr();

   printf("\n Enter the Number of Elements: ");
   scanf("%d",&n);

   printf("\n Enter %d Elements: ",n);
   for(i=0;i<n;i++)
   {
      scanf("%d",&a[i]);
   }

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

```
    printf("\n The Sorted array in ascending order: ");
   for(i=0;i<n;i++)
   {
      printf("%d ",a[i]);
   }
   getch();
}
```

ALGORITHM

PURPOSE: Arranging the numbers in ascending order using bubble sort technique
INPUT: N, interger numbers in arrary a[i]
OUTPUT: Numbers are arranged in ascending order

START
STEP 1: [Input number of elements]
Read n

STEP 2: [Input the elements/numbers into array]
For i←0 to n
Read a[i]
End for

STEP 3 : [Sorting the elements in ascending order]
For i ←0 to n-1
For j←i+1 to n

[Compare the elements]
If(a[min]>a[j]) then
min = j;

STEP 4:
If(min!=i)
[Swap these elements]
Temp←a[i]
a[i]←a[min]
a[min]←temp
end if
end for
end for

STEP 4: [Display the sorted elements of array]
For i←0 to n
Print a[i]
End for

STEP 5: [Finished]
STOP

IMPORTANT QUESTIONS

1. What is an array? Explain declaration and initialization of one dimension and 2 dimension array.
2. Explain bubble sort with pseudo code and give example.
3. Write a program to calculate multiplications of 2 matrices.
4. Write a c program to find (one dimension array)
   i. Even and odd number in array elements.
   ii. Sum of even and odd number in the array.
   iii. Sum and average of array elements.
5. Write a c program to find (two dimension array)
   i. Even and odd number in array elements.
   ii. Sum of even and odd number in the array.
   iii. Sum and average of array elements.
6. Explain any 5 string handling functions with example.
7. Write a c program to copy the content of one string into another using function.
8. What is string? Explain the declaration and initialization of string.
9. Write a C program for searching techniques
   i. Linear Search
   ii. Binary Search
10. Write a C program for sorting techniques
    a. Bubble sort
    b. Selection Sort

****

Module 4

User Defined Functions and Recursion

Example programs, Finding Factorial of a positive integers and Fibonacci series.

Functions:

- A function is a sub program or group of statements that perform a specific task.
- A large program can be divided into manageable pieces called modules, where each module does a specific work. And each module is considered as a function.

Advantages of functions:

- Functions are used to avoid the repetition of codes in a program.
- A function reduces the length of the programs.
- Debugging functions is easier.
- Programs can be written easily.
- Functions can be shared by many programmers.

The repetitive code of a program can be written within a function and wherever necessary this function is used to do that work.

Types of functions:

Different types of functions are

- Built in Functions(Library functions)
- User-defined functions

Built in functions or **built-in library functions** are the functions provided in the library of the C compiler. These functions are created and stored in the library of the C compiler by C developers.

Ex. Standard *i/o* functions like, *scanf(), printf()*

Standard *string* functions like- *strlen(), strcat()*

User-defined functions:

Users or programs create their own functions in C programs. These functions are called *user defined functions.*

Elements of user-defined functions:

- ***Function definitions***
- ***Function call***
- ***Function declaration***

1. **Function definition**: The program module that is written to achieve a specific task is called function definition. The following code shoes the general syntax for function definition

```
return-type function-name(parameter list)
{
        local variable declaration;
        executable statement 1;
        executable statement 2;
        .
        .
        .
        executable statement n;
        return(expression);
}
```

In the above code, *return-type* indicates the *data type* of the *information* that the function *returns* to it's' calling program. A function can return *int, float, double, char* or any other data type of information.

function-name:

- refers to the name of the function and list of parameters passed into the function. The parameters enclosed within parentheses are called *formal parameters*.
- A function can contain *local variables* that are declared inside the function and they are *not recognized outside* the function.
- The function contains the executable code in many lines.
- The last statement is the *return* statement that *returns result or information* to the calling program. A function may contain or may not contain a return statement.
- Once the function is defined it can be *invoked* or *called* from anywhere in the program

For examples consider the below program,

```
void main()
{
        int m,n,res;
        printf("Enter two numbers");
        scanf("%d%d", &m, &n);
        res=ADD(m,n);          /* function call or function invocation*/
        printf("Addtion of m and n is %d", res);
}
int ADD(int a, int b) //function definition
{
        return(a+b);
}
```

The above function ADD( ), accepts two numbers of type **int**, adds these two numbers and return the result which is of **int** type to it's' calling program.

2. **Function Call or Function invocation**:

The general syntax for function invocation is.

variable=function-name(actual-parameter-list);

variable is the variable that stores the result returned by the function. It should be same to the return type of function.

Function –name is the name of the function

We know that execution of programs always starts from function *main()*. If we want to add two numbers then function ADD() is to be invoked. *Invoking* a function is called *function call*.

For Ex. ADD() can be invoked as,

```
void main()
{
        int             m,n,res;
        printf("Enter two numbers");
        scanf("%d%d", &m, &n);
        res=ADD(m,n);          /* function call or function invocation*/
        printf("Addtion of m and n is %d", res);
}
```

- A function can be called by specifying it's name followed by a list of arguments enclosed in the parentheses and separated by commas.
- If a function call does not require any arguments, then an empty pair of parentheses follows the function name.
- The arguments appearing in the **function call** are referred to as *actual parameters*

3. **Function declaration or function prototyping**

The process of declaring the functions before they are used is called *function declaration* or *function prototyping*

Declaration of functions can be done before main or after include files

Ex.

```
int ADD(int , int );   this is function declaration
void main()
{
        int             m,n,res;
        printf("Enter two numbers");
        scanf("%d%d", &m, &n);
        res=ADD(m,n);          /* function call or function invocation*/
        printf("Addtion of m and n is %d", res);
}
```

```
int ADD(int a, int b) //function definition
{
        return(a+b);
}
```

Formal Parameters and Actual parameters:

The variables defined in the function definition are called *formal parameters*

The variables used in the function or function invocation are called *actual parameters or argument.*

Ex.

```
void main()
{
        int m,n,res;
        printf("Enter two numbers");
        scanf("%d%d", &m, &n);
        res=ADD(m,n);          /* m and n are actual parameters or arguments*/
        printf("Addtion of m and n is %d", res);
}

int ADD(int a, int b) //a and b are formal parameters
{
        return(a+b);
}
```

Categories of functions:

Based on the parameters and return value, the functions are categorized as,

- Functions with no parameter and no return values
- Functions with no parameter and with return values
- Functions with parameter and no return values
- Functions with parameter and with return values

Functions with no parameter and no return values

In this type of functions there is ***no*** data transfer between the ***calling function*** and ***called function***. So calling function cannot send values and hence called function cannot receive the data.

Ex.

```
void main()
{
        display();       /* only function is called */
}
```

```
int display() //a and b are formal parameters
{
        printf("hello world\n");

}
```

- display( ) is called with no arguments so no parameters are defined in function heading
- display() in main transfers the control to called function and print "hello world"
- When last statement is executed control is transferred to the calling function

Functions with no parameter and with return values

In this type of function the data is not passed from calling function to called function but the result is returned to calling function

Ex.

```
void main()
{
        int res;
        res=ADD();     /* only function is called */
        printf("Addtion of m and n is %d",res);

}

int ADD() //function with no  parameters
{
        int       m=10,n=20,sum;
        sum=m+n;
        return(sum);
}
```

Function with parameter and with no return values

In this type of function the data is transferred from ***calling function*** to ***called functi***on
But there is no data transfer from ***called function*** to ***calling function***.

```
main()
{
        int a,b;
        a=10,b=20;
        exchange(a,b);
        printf("a=%d and b=%d", a,b);
}

void exchange(int m, int n)
{
        int temp;
```

```
        temp=m;
        m=n;
        n=temp;
}
```

Functions with parameter and with return values

In this type of function the data is transferred from ***calling function*** to ***called functi***on
And the result is transferred from ***called function*** to ***calling function***.

```
void main()
{
        int             m,n,res;
        printf("Enter two numbers");
        scanf("%d%d", &m, &n);
        res=ADD(m,n);       /* m and n are actual parameters or arguments*/
        printf("Addtion of m and n is %d", res);
}

int ADD(int a, int b) //a and b are formal parameters
{
        int sum;
        sum=a+b;  //function with no return value
        return(sum);
}
```

Parameter passing mechanism:

During the declaration of functions and during invocation of function the parameters are passed into the function, passing parameters to function is done in

- Call by Value
- Call by Reference

Call by Value:

Calling or invoking a function by passing *values* or *variables* to the function is called ***call by value***.

i.e in pass value , the values of *actual parameters* are *copied* into *formal parameters*

In this method any changes to the values or variables are done in the function, cannot be visible outside the function

Ex.

```
main()
{
        int a,b;
        a=10,b=20;
```

```
        exchange(a,b);
        printf("a=%d and b=%d", a,b); /* no exchange a=10, b=20 will  be printed*/
}

void exchange(int m, int n)  /* m=10 amd n=20/*
{
        int temp;
        temp=m;
        m=n;
        n=temp;                                 /* after exchange      m=20, n=10*/
}
```

Working:

- Execution starts from function *main()* and the variables **a** and **b** are assigned the values 10 and 20 respectively
- The function *exchange()* is called with actual parameters **a** and **b** whose values are 10 and 20
- In the function *exchange()* the formal parameters **m** and **n** receive the values 10 and 20
- In the function *exchange()* the values **m** and **n** are exchanged
- But the values of the actual parameters in *main ()* have not been exchanged.

In *pass by value* (call by value) any changes done on formal parameters will *not have any effect* on actual parameters.

Call by Reference:

If we want to view the changes made to the variables , outside the function, we can pass the *address* of the variables to the function, So any change to the formal parameters imply there is a change in actual parameters. This technique is called *call by address* or *pass by address*.

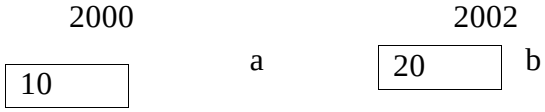
In this technique the address of actual parameters are sent to function. In the called function the formal parameters should be declared as **pointers**.

Ex.

```
main()
{
        int a,b;
        a=10,b=20;
        exchange(&a,&b);                        /* address of a and b are sent*/
        printf("a=%d and b=%d", a,b); /* exchange a=20, b=10 will be printed*/
}

void exchange(int *m, int *n)  /* m=2000 and n=2002*/
{
        int temp;
        temp=*m;
        *m=*n;
        *n=temp;                                /* after exchange      *m=20,* n=10*/
}
```

Working:

- Assume address of variable a is 2000, and b is 2002



- The function *exchange( )* is invoked by sending the address of **a** and **b** to the function.
- In the function exchange( ) the formal parameters **m** and **n** are declared as **pointers** as they have to hold the address of **a** and **b** respectively.
- In the function exchange( ) ***m** refers to the *value* stored in the address held by m and ***n** refers to the *value* stored in the address held by n . so values of a and b will be exchanged.

The changes made to formal parameters imply changes to actual parameters

Recursive functions:

A function that call's by *itself* is called a recursive function.

This type of functions call themselves during their execution until certain conditions are not satisfied.

Ex. .

```
main()
{
        long int n,f;
        printf("Etner a number");
        scanf("%ld",&n);
        f=factorial(n);                 /* call the function */
        printf("The factorial of %d  is = %d", n,f);
}

long int factoiral(long int n)
{
         if(n==1)
                return 1;
         else
             return(n*factorial(n-1)); /*calling factorial itself*/
}
```

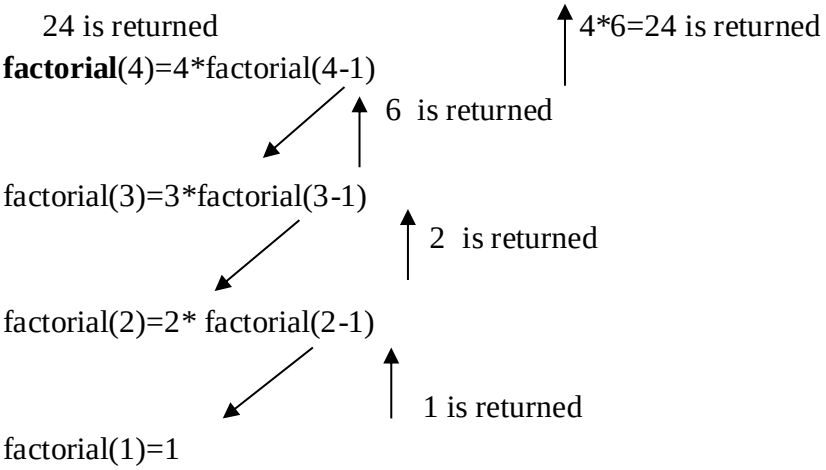
Working:

- In the *main ()* function **factorial** is called by passing n to it
- **n** is checked if it is 1, if not the program control moves to return(n***factorial**(n-1))
- now the factorial function is called with **factorial**(n-1) and this will be repeatedly calls up
- when the function **factorial**(1)calls with n=1 the result is returned as 1 through return 1 to it's calling function
- again the intermediate result will be calculated and returned to its calling function and so on.

Ex. if n=5, for the first time factorial is called with *factorial(5)* as **n** is not equal to 1 , the line

return(5***factorial**(5-1)) is executed.

return(5***factorial**(5-1))= **5*factorial**(4)at this stage 5*24=120 is returned to *factorial (5)*



Program to get nth term of Fibonacci Series

```
void main()
{
        int i,n;
        printf("enter the term number");
        scanf("%d",&n);
        i=fib(n);
        printf("the %d fibonacci number is %d", n,i);
}
int fib(m)
{
        if(m==1 || m==2)
        return 1;
        else return(fib(m-1)+fib(m-2));
}
```

Storage classes/Scope availability and life time of variables:

The different storage classes in C language are

- Global variables/ External variables
- Local variables
- Static variables
- Register variables
- Automatic variables

Global variables: these are the variables defined after preprocessor directive and before void main() and which can be accessed throughout the program.

Ex.

```
int display( );
int a = 10;                    A Global variable
void main()
{
int b=20;                      B is Local variable in main function
printf("a=%d",a);
printf("b=%d",b);
display();
}

int display ()
{
int c=30;                      C is Local variable in display function
printf("a=%d",a);
printf("c=%d",c);
}
```

Local variables: Local variables are the variables which are defined within a function. It is defined are defined inside a block have local scope, it can be accessed between limited boundary that is block.

In the above program c is used in function add( ) is a local variable as it is defined only in that function

Static Variable:

The variables that are declared using the keyword static are called static variables. They have the characteristics of both local and global variables.

Ex.

```
void main()
{
        display();          /* prints 1 */
        display();          /* prints 2 */
        display();          /* prints 3 */
        display();          /* prints 4 */
        display();          /* prints 5 */
}
 void display()
{
        static int i=0;
        i++;
        printf ("%d\n", i)
}
```

Each time control goes into the function, the previous value of ***i*** is used and it is updated. The updated value is used in the next call of the function.

Register Variable:

Any variable declared with the *qualifier* **register** is called a register variable. The variable value is stored not in memory but it is stored in one of the register of CPU, Registers are accessed much faster compared to memory. So this leads to *faster execution of the program*.

This declaration is allowed is allowed only for the local variables and to the formal parameters.

Ex. register int x; /* x value is stored in one of the CPU register*/

Automatic variable:

A variable declared inside a function without storage class specification is by default an automatic variable. The storage class of an variable ***number*** in the example below is automatic

Ex:
```
main()
{
        int number;
        ……....
        ……..
}
```

We may also use the keyword auto to declare automatic variables explicitly

```
main()
{
        auto int number;
        ……..........
        ……..
}
```

IMPORTANT QUESTIONS

1. Explain different types of functions and elements of user defined functions along with examples.

OR

2. Explain ***function definition, function call*** and ***function declaration*** with examples.

3. What are ***actual*** and ***formal parameters***? Illustrate with example

4. What is *recursion*? Write a program to compute the factorial of a given n number using recursion.

OR

5. Write recursive function to calculate factorial of a number & write a C program to calculate the ncr(Binomial coefficient).

6. Explain the two categories of arguments passing techniques with examples

7. Explain different types of functions based on ***return value*** and ***parameters*** with examples.

OR

8. Explain different categories of functions

9. Explain different types of Storage classes along with examples

OR

10. Explain Scope availability and life time variables along with examples.

11. Write a C program to get nth term of Fibonacci Series

****

Module 5

Structure and Pointers, Preprocessor Directives

STRUCTURES:

Structure is a collection of one or more variables of same data type or different data types that are grouped together under a *single name* .It is user defined data type. It is also a record about particular entity.

Declaration of structure: Structure is declared using key word ***struct*** followed by structures name. The structures can be declared in three types

1. Tagged structure
2. Structure without tag
3. Type defined structure

1. **Tagged structure:** A structure is declared using key word **struct** followed by identifier called tag_name. structure with tag name is called tagged structure

Syntax:

```
struct < tag_name >
{
datatype variable1;
datatype variable2;
…….. ……
};
```

Ex.

```
struct student
{
        int rollno;
char name[10];
int marks;
        char grade;
        };
```

```
struct < tag_name > varaible1,variable2;
struct student s1,s2;
```

- Where struct is the keyword specifies structure is defined
- tag_name is used as an identifier to define the structure variables, structure name is any valid variable name.
- The variables inside the structure are defined with data type. They called as ***members*** or ***fields*** of structure. They can be of ***int float*** or ***char*** data type
- Each member is ended with **;**
- The definition of structure should end with ***;***
- Structure variables are declared with data type as shown struct < tag_name > varaible1,variable2;

In this structure definition student is structure name with 4 data members rollno is integer type indicates students rollnumber, name indicates student name, marks indicates marks scored by student grade indicates grade scored by student.

s1,s2 are variables of ***stuct student***, s1 indicates first student s2 indicates second studnet.

Space allocated for s1 is as shown below,

s1.rollno	integer	2 bytes
s1 Name	Array of 10 characters	10 bytes
s1.marks	Integer	2 bytes
s1.grade	character	1 byte
	Total memory=	15 bytes

2. Structure without tag: A structure is defined without tag_name is called structure without tag.

Syntax:

```
struct
{
datatype variable1;
datatype variable2;
…….. ……
} varaible1,variable2;
```

Ex.

```
struct
{
    int rollno;
    char name[10];
    int marks;
    char grade;
} s1,s2;
```

3. **Type defined structure:** The structure defined with keyword typedef is called type defined structure.

Syntax:

```
Typedef  struct
{
datatype variable1;
datatype variable2;
…….. ……
}< tag_id >;

< tag_id > varaible1,variable2;
```

Ex.

```
typedef  struct
    {
    int rollno;
    char name[10];
    int marks;
        char grade;
} student;

student s1,s2;
```

Initialization of structure:

1. structure variables can be initialized at declaration time, as shown,
struct student s1={111,"Raju",430,'A'};

This initializes s1.rollno =111, s1.name = Raju, s1.marks = 430,s1.grade = 'A'

2. Through key board variables can be initialized with **.** (dot or access) operator.
Ex. struct student

```
{
        int rollno;
        char name[10];
        int marks;
        char grade;
};
main()
{
        struct student s1,s2;
        printf("enter the details of first student");
        scnaf("%d%s%d%c",&s1. rollno,s1. Name,&s1.marks,&s1.grade);
        printf("enter the details of second student");
        scnaf("%d%s%d%c",&s2. rollno,s2. name,&s2.marks,&s2.grade);
}
```

Values Entered through keyboard are assigned to s1 variable are as shown,

s1.rollno	111
s1 Name	"Raju"
s1.marks	430
s1.grade	'A'

Values Entered through keyboard are assigned to s2 variable are as shown,

S2.rollno	222
S2 Name	"Ali"
S2.marks	400
S2.grade	'B'

Array of` structures:

Similar to array of integers , we can define an array of structure .For example, to store the details of 10 students we can define an array of students as shown,

```
struct student
    {
        int rollno;
        char name[10];
        int marks;
        char grade;
    };
main()
{
        struct student s[10]; //s is an array of 10 elements of student type.
        printf(”enter the details of students”);
        for(i=0;i<10;i++)
        scnaf(“%d%s%d%c”,&s[i].s[i].rollno,&s[i].Name,s[i].marks,&s[i].grade);
        printf(”Details of  students ”);
        printf(“%d%s%d%c”, s[i]. rollno,s[i]. name, s[i].marks, s[i].grade);
}
```

Structure to contain the details of employees

```
struct employee
    {
        int id;
        char name[10];
        int salary;
    };
main()
{
        struct employee e[10]; //e is an array of 10 elements of employee type.
        int n;
        printf(”enter the number of employees”);
        scanf(“%d”,&n);
        printf(”enter the details of employees”);
```

```
        for(i=0;i<n;i++)
        scnaf("%d%s%d",&e[i].id,e[i].name,&e[i].salary);
        printf("Details of employees ");
        for(i=0;i<n;i++)
        printf("%d%s%d",e[i].id,e[i].name,e[i].salary);
}
```

Program to read student details using an array of structure with 4 fields (Roll number, Name, Marks and Grade) . Printing the marks of the student, whose name is given

```
struct student
        {
                int rollno;
                char name[10];
                int marks;
                char grade;
        };
main()
{
        int n,i,found=0;
        char sname[10];
        struct student s[10];
        printf("enter how many students");
        scanf("%d",&n);
        printf("enter the details of 10 students");
        for(i=0;i<n;i++)
        scnaf("%d%s%d%c",&s[i].s[i].rollno,&s[i].Name,s[i].marks,&s[i].grade);
        printf("Details of students are ");
        printf("%d%s%d%c", s[i]. rollno,s[i]. name, s[i].marks, s[i].grade);
        printf("enter name of student whose marks to be displayed");
        scanf("%s",sname);
        for(i=0;i<n;i++)
        {
                if(strcmp(s[i],sname)==0)
                {
                        printf("Marks of student is %d"s[i].marks);
```

```
            printf("name=");
            puts(s[i].name);
            exit(0);
            }
        printf("given student name not found");
        }
```

UNIONS: Union is same as structure. But it assigns memory to only a single member which is having large space. The memory in union is shared by all the variables. Only one variable can be accessed at a time.

Ex. student is defined as , here struct is replaced by the keyword union

Syntax :

```
union < tag_name >          union student
    {                               {
    datatype variable1;         int rollno;
    datatype variable2;         char name[10];
    ……..    …….                    int marks;
                                    char grade;
};                                  };
```

union student s1;

If s1 is a variable of union student type,

s1.Name	Array of 10 characters	10 bytes

As name field is the largest field, memory of 10 bytes will be allocated. This will be shared by all other members.

Difference between structure and union

slno	Structure	Union
1	Keyword ***struct*** is used to define a structure	Keyword ***union*** is used to define a structure
2	Memory is allocated for each member	Memory is allocated only
3	Altering values of one member will not affect other members	Altering values of one member will 4affect other members
4	Individual members can be accessed at a time	Only one member can be accessed at a time

5	All members can be initialized at one	Only first member can be initialized

6.	Syntax:
	struct < tag_name >
	{
	datatype variable1;
	datatype variable2;
	…….. ……
	};
	
	Ex. struct student
	{
	int rollno;
	char name[10];
	int marks;
	char grade;
	};
		Syntax:
		union < tag_name >
		{
		datatype variable1;
		datatype variable2;
		…….. ……
		};
		
		Ex. union student
		{
		int rollno;
		char name[10];
		int marks;
		char grade;
		};

POINTERS: Pointer is a variable which contains the address of another variable of same type.

Declaration of Pointers:
General Syntax:

```
Datatype  * pointer_variable ;
```

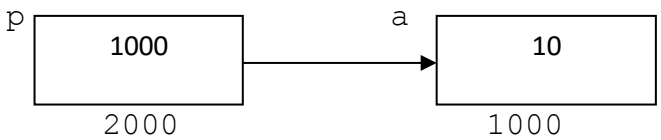
`Ex. int *p;` p is an integer pointer variable that is used to point to another integer variable. * is called dereferencing operator.

Initialization of pointer:
Pointers can be initialized with address of other variables as shown,

```
int *p,a;
p=&a;       //p is assigned with the address of a
```

Program to demonstrate the use pointers;

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=10,*p;
p=&a;
printf("the value of p = %u",p);--→1000
printf("the value of a = %d",a);--→10
printf("the value of *p = %d",*p);---→10
getch();
}
```



Pointer to Array

We can use a pointer to point to an Array, and then we can use that pointer to access the array.

Syntax: datatype * ptr_name;
Ptr_name=&array_name or ptr_name=array_name;

```
Ex: int a[5]={ 1, 2, 3, 4, 5 };
```

We can declare a pointer of type `int` to point to the array **a**.

```
int *p;
p = a;  or p = &a[0]; //both the statements are same meaning.
```

Now we can access every element of array **a** using **p++** to move from one element to another.

NOTE : You cannot decrement a pointer once incremented. p-- Won't work.

EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a;  // same as int*p = &a[0]
for (i=0; i<5; i++)
{
 printf("%d", *p);
 p++;
}
getch();
}
```

In the above program, the pointer ***p** will print all the values stored in the array one by one. We can also use the Base address (**a** in above case) to act as pointer and print all the values.

1	**2000**
2	**2002**
3	**2004**
4	**2006**
5	**2008**

p | 2000 |

`int *p = a;` this statement assigns p to contain the address 2000
when `p++;` is executed p points to `2002`

Character pointer:

Strings are array of characters. We can use pointer to that pointers to the characters present in the string (array of characters).

Declaration Syntax: datatype * ptr_name;
ptr_name=string_name;

```
Char str[10]="nanjesh"
```

We can declare a pointer of type char to point to the array of character **str**.

```
char *p;
```

```
p = str;
```

The pointer points to the starting character of the string str i.e 'n' and later the pointer can be incremented to get the other elements.

EXAMPLE:

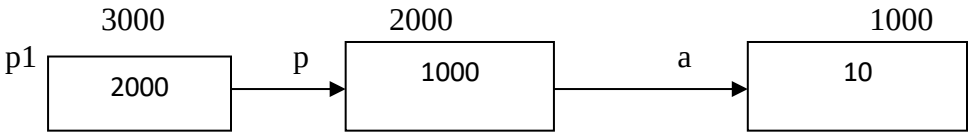
```
#include<stdio.h>
#include<conio.h>
Void main()
{
int i;
char str[10] = "nanjesh";
char *p = str;
for (i=0;str[i]!='\0'; i++)
{
 printf("%c", *p);
 p++;
}
getch();
}
```

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

Declaration: Datatype ** pointer_variable ;

Ex. int *p; –→p is an integer pointer.
int **p1; --→p1 is pointer to pointer.
int a;

p=&a; ---→p points to a integer variable a.
p1=&p; ------→p1 points to the pointer variable p.



EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=10,*p,**p1;
p=&a;
p1=&p;
printf("the value of a = %d",a);---→ 10
printf("the value of p = %u",p);---→1000
printf("the value of *p = %d",*p);--→10
printf("the value of p1 = %u",p1)---→2000
printf("the value of p1 = %u",*p1)---→1000
printf("the value of **p1 = %d",**p1);----→10
```

```
getch();
}
```

Pointers as function arguments: The address of the variables can be passed as an argument to a function, when we pass addresses to a function, the parameters receiving the addresses should be pointers(call by reference).

Ex:

```
Ex.     main()
        {
                int a,b;
                a=10,b=20;
                exchange(&a,&b);                    /* address of a and b are sent*/
                printf("a=%d and b=%d", a,b);       /* exchange a=20, b=10 will be printed*/
        }

void exchange(int *m, int *n)  /* m=2000 and n=2002*/
{
                int temp;
                temp=*m;
                *m=*n;
                *n=temp;                             /* after exchange      *m=20,* n=10*/
}
```

Functions returning pointers:

A function can return a single value by its name or return multiple values through pointer parameters. We can also force a function to return a pointer to the calling function.

Ex:

```
int *larger(int *,int *);
main()
{
        int a=10;
        int b=20;
        int *p;
        p=larger(&a,&b);                    //Function call
        Printf("%d",*p);
}

int *larger(int *x,int *y)
{
        if(*x>*y)
        return x;                           //address of a
        else
        return y;                           //address of b
}
```

Pointers to Functions:

A function, like a variable, has a type and an address location in the memory. It is therefore possible to declare a pointer to a function, which can then be used as an argument in another function.

Ex:

```
double mul(int , int);
double (*p1);
p1=mul;
```

declare p1 as a pointer to a function and mul as a function and then make p1 to point 2 to the function mul that is equivalent to

```
mul(x,y);
```

Pointers and structures

The name of an array stands for the address of the zeroth element the same thing is true of the names of arrays of structure variables.

Ex: Suppose product is an array variable of struct type. The name product represents the address of its zeroth element.

```
struct inventory
{
        char name[30];
        int number;
        float price;
}product[2], *ptr;

ptr=product;
ptr→name;
ptr->num;
ptr->price;
```

Dynamic Memory allocation functions:

Allocation of memory to variables is known as memory allocationTwo types of memory allocation are

- ***Static memory allocation:*** Allocating memory at compile time is known as ***static*** memory allocation. Memory allocated at this time is ***fixed*** , and it cannot be ***altered,*** Memory as allocated at declaration
- ***Dynamic memory allocation***: Allocation of memory during runtime is known as dynamic memory allocation. Memory once fixed can be altered at later stages in the program.Memory is allocated as and when required. Using pointers memory can be allocated dynamically

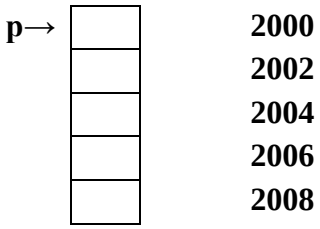
Dynamic memory allocation functions: The various dynamic memory allocation and deallocation functions are

1. `malloc()`
2. `calloc()`
3. `realloc()`
4. `free()`

1. **malloc( ):**this function allocates specified amount of memory . The syntax is as shown
   syntax : datatype pointer_vatriable;
   pointer variable=(casting*)malloc(n*sizeof(datatype));
   - *pointer_ variable* is of any data type
   - **casting** specifies the allocated memory is to cleared to store the specified type of data
   - **malloc()** function allocates the memory
   - `sizeof()`- Returns size in bytes
   - n-specifies the number of elements to which memory to be allocated

   Ex.
   ```
   int *p;
   p=(int*)malloc(5*sizeof(int));
   ```
   Here **p** is pointer variable holds the address of first memory location returned by ***malloc ()*** to hold ***5*** integer numbers.

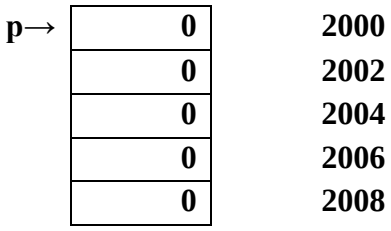
   Ex.



2. **calloc( ):** this function is same as **malloc() ,it allocates a block of memory.** but it initializes memory locations with 0 and returns pointer to first memory location
   syntax :datatype pointer_vatriable;
   pointer variable=(casting*)calloc(n,sizeof(datatype));

   - *pointer_ variable* is of any data type
   - **casting** specifies the allocated memory is to cleared to store the specified type of data
   - **calloc()** function allocates the memory
   - `sizeof()`- Returns size in bytes
   - n-specifies the number of elements to which memory to be allocated

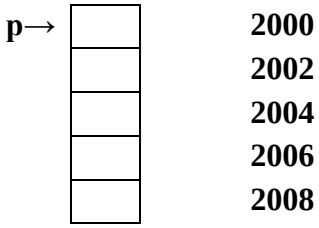
   Ex.
   ```
   int *p;
   p=(int*)calloc(5,sizeof(int));
   ```
   Here **p** is pointer variable holds the address of first memory location returned by ***calloc ()*** to hold ***5*** integer numbers.



3. **realloc( )** : this function reallocates the memory that is already allocated either by **malloc** or **calloc.**This function helps to ***alter*** the memory size by *extending* or *shrinking* the allocated memory.

   Assume

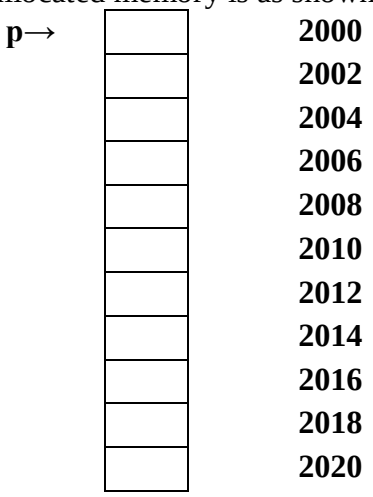
```
int *p;
p=(int*)malloc(5*sizeof(int));
```



   when the memory to be increased so another 6 numbers can be stored, then ***realloc*** is used as shown below

```
p=(int*)realloc(11*sizeof(int));
```

   and allocated memory is as shown,



4. free() : function is used to deallocate the allocated memory either by ***malloc*** or ***calloc*** or ***realloc***

   syntax: `free(pointer_variable);`

   if the above example is considered then

   `free(p);` releases the memory

Preprocessor Directives:

- The set of statements which are processed before compilation are called preprocessor.
- Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.
- The different preprocessor directives in C language are,

1.#define: Defines a macro substitution. This macro defines *constant value* and can be any of the basic data types.

Syntax : #define identifier string

Ex: #define pi 3.14

Program to find square and cube of a number using macro.

```
#include<stdio.h>
#include<conio.h>
#define square(a) a*a
#define cube(a) a*a*a
Void main()
{
        int n=2;
        printf("square=%d\n",square(n));
        printf("cube=%d\n",cube(n));
        getch();
}
```

Program to find area of a circle using macro.

```
#include<stdio.h>
#include<conio.h>
#define area(r) 3.14*r*r
void main()
{
        int r=2;
        printf("area=%d\n",area(r));
        getch();
}
```

2. #include: specifies the files to be included. The source code of the file "file_name" is included in the main program at the specified place.

Syntax: #include <file_name>
Ex: #include<stdio.h>,
#include<conio.h>,
#include<math.h>

3. #undef: undefines a macro which is defined using #define.

Syntax: undef identifier
Ex: #define pi 3.14
#undef pi

4.#ifdef : Checks a macro is defined or not. If defined block 1 is executed else block 2
Ex:

```
#define pi 3.14
#ifdef  pi
        Printf("pi is defined\n");
        Else
        Printf("pi is not defined\n");
```

5.#ifndef: This is reverse of #ifdef.

```
#define pi 3.14
#ifndef  pi
        Printf("pi is not defined\n");
        Else
        Printf("pi is defined\n");
```

6.#if and else : #if works using relation operator. Else works as an alternative to #if.

7. #error: prints the error message on stdrr

Ex:

```
#define pi 3.14
#ifndef  pi
        Printf("pi is not defined\n");
        Else
        #error "pi not found"
```

IMPORTANT QUESTIONS

1. Define Structures. Explain declaration of Structures and accessed using dot operator with examples.
2. Show how structure variables are passed as a parameter to a function with examples.
3. Write a C program to maintain record of n students detail using using array of structures with 4 fields (Roll No., name, marks, grade). Each field is an appropriate data type. Print the marks of student if student name is given.
4. Explain with an example, how to create a structure using typedef.
5. Explain array of structures with suitable example program.
6. Explain structures within structure with suitable example program.
7. Write a C program to maintain a record of n employee detail using an array of Structures with 3 fields (id, name, salary) and print the details of employees whose salary is above 5000.
8. Define Pointer variable. Explain with an example, the declaration and initialization and accessing of pointer variable
9. What is DMA? Explain following C functions along with syntax and example to each.
    - malloc()
    - calloc()
    - realloc()
    - free()

10. Develop a C program to read 2 numbers and functions to swap these numbers using pointers
11. Write short notes on following:
    a. Pointers and arrays
    b. Pointers and character strings
    c. Pointers as function arguments
    d. Function returning pointers
    e. Pointers to functions
    f. Pointers and structures
12. Write a C program to find the sum, mean and standard deviation of all elements in an array using pointers
13. Explain any 5 preprocessor directives in C.

****