PROGRAMMING FOR PROBLEM SOLVING

NOTE:

MAKAUT course structure and syllabus of 2nd Sem has been changed from 2019. **PROGRAMMING FOR PROBLEM SOLVING** has been introduced as a new subject in the present curriculum. We are providing chapterwise model questions & answers along with the complete solutions of new university papers, so that students can get an idea about university questions patterns.

INTRODUCTION TO PROGRAMMING

☞ Chapter at a Glance

Anatomy of Computer

A computer can be defined as a digital ectronic machine that can be programmed to perform large set of mathematical, logical calculations. It is also used to processhuge sets of data in accordance with a predetermined set of instructions. The computer itself is referred to as hardware, whereas the various programs are referred to as software.

Computer Harware Assembly

Computer hardware can be defined as a unit of all the physical components that are inter-connected by a circuit and the software that provides instructions for the hardware to accomplish tasks. It includes

Motherboard: It represents the main circuit controller of a computer that in turn connects the various peripherals attached to a computer.

RAM (random access memory): It represents the primary physical storage.

HDD (Hard disk drive): It represents the secondary physical storage

Processors: It represents the main central processing unit, the nucleus of a computer system responsible for all the complex calculations in a computer.

Keyboards/Mouse/Scanner: They represent the input device to a computer.

Monitors: It represents the VDU (visual display unit) of a computer.

Secondary storage devices: Floppies (magnetic storage device), CD ROM/DVD (optical storage device), USB storage etc.

Printers: Typically output deivce to print.

Computer Software

A computer is capable of unerstanding digitals signals in the form of a binary bit either '0' or '1'. For example, if '0' represents the absence of a pulse and 1 its presence, a sequence such as 1001 has some specific meaning for the computer. 1001 represents 9. Similarly a character for example A is represented by 1000001. Such a binary representation is used in computers and the programs written using these binary numbers are said to be **machine language.**

A computer runs on top of a software known as operating system. It is the main system program to run other software programs.

Other software programs most commonly used are Microsoft Office, Browser, text editors etc. These are mostly termed as utility software programs readily available from the market.

Applications softwares are programming languages like C, C++, Java, Python which are used to write system programs, utility programs and many other customized softwares as per requirement. So computer programming language is the fundamental requirement of a software.

In order to write, complie and execute programming languages we need specific softwares like assemblers, compliers and interpreters.

An **assembler** is a systems program that translates the source language (assembly language) into a machine equivalent language (bits) known as the intermediate object code understandable by the computer hardware. In short an assembler is a translator for a low level language.

A **compiler** is a systems program that translates the source language (high-level language like C, C++, C# and Java) into a machine equivalent language (bits) known as the intermediate object code understandable by the computer hardware. In short a compiler is a translator for a high level language.

Executing a program written in a high-level programming language is basically a two-step process:

- The source program must first be compiled that is translated into object program.
- . Then the object program is loaded into memory and executed.

An **Interpreter** is a high-level language translator that converts individual high-level computer language program instructions (source code) into machine instructions. It translates and executes each statement line-by-line during the execution of a program. BASIC and Java are the high level languages that are sometimes interpreted and sometimes compiled.

Algorithm, Flowchart, decision tables and Programming

Algorithm, flowchart, decision tables and programming are the most fundamental aspect of software design and implementation.

A software design can be conceptualized as logical steps to understand a problem given for a solution. Initially the problem definition will appear to be vey abstract in nature. Hence, the first step in the problem- solving process is to review the problem carefully in order to determine its

- Input: what information is given and which items are important in solving the problem
- Output: what information must be produced to determine that the problem was solved?

Algorithm

Once a problem is specified, a procedure or process to produce the required output from the given input must be identified. Since the computer is a machine possessing no inherent problem-solving capabilities, this procedure must be formulated as a detailed sequence of simple steps. Such a procedure or a logical interpretation is called an **algorithm**.

The steps that comprise an algorithm must be organized in a logical, clear manner so that it can be easily understood to translate that into a program that implements this algorithm is similarly well structured. Algorithms and programs are designed using three basic methods of control:

Sequential: Steps are performed in a strictly sequential manner, each step being executed exactly once.

Selection: One of several alternative actions is selected and executed.

Repetition: One or more steps are performed repeatedly.

*An **algorithm** can be defined as a finite set of step-by-step instructions for a problem-solving or computation procedure, especially one that can be implemented by a computer programming language.*

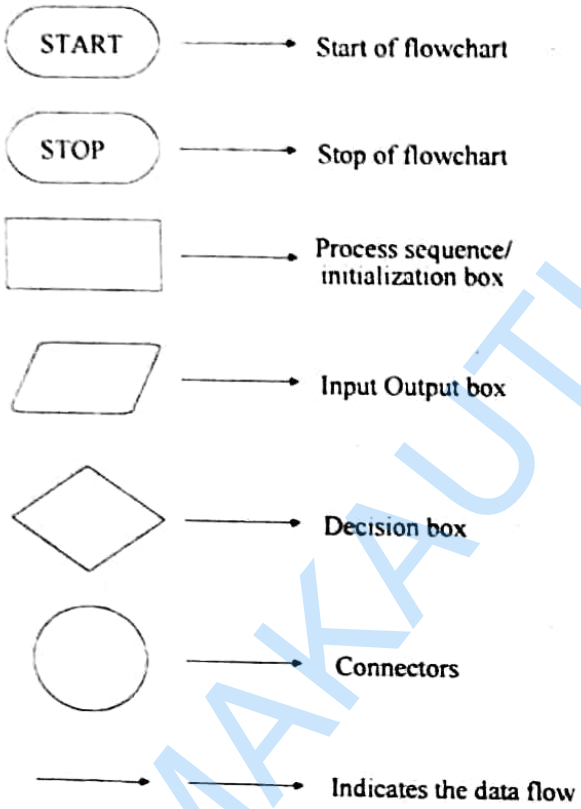
It can also be defined as mathematical procedure that can usually be explicitly encoded in a set of computer language instructions that manipulates data.

Flow chart and Decsision tables

Algorithms can be represented using graphical or tabular form. The graphical representation with specific notations is called a flow-chart, whereas, the tabular representation, almost similar to a mathematical matrix is called decision tables.

Flow chart

The graphical representation with specific notations is called a flow-chart. The various components of a flow-chart are as follows:



Decsision tables

The tabular representation, almost similar to a mathematical matrix to test different combination of inputs to a certain output is called decision table. A certain algortihmic behaviour that can be represented by a generic decision table is as follows

Title: Author: Comments: Date:	**Rules**		
Conditions	**Rule 1 (Action)**	**Rule 2 (Action)**	**Rule 3 (Action)**
x>y	**True**	**False**	**False**
X<y	**False**	**True**	**False**
X == y	**False**	**False**	**True**

Psuedo codes

Unlike high-level programming languages such as Pascal or C, there is no set of rules that defines precisely what is and what is not in pseudocode. It varies from one programmer to another. Pseudocode is a mixture of natural language, such as English, and symbols, terms, and other features commonly used in one or more high-level languages. The following features are common to most pseudocodes:

- The usual computer symbols are used for arithmetic operations: + for addition, - for subtraction, * for multiplication, and / for division.

- Symbolic names (identifiers) are used to represent the quantities being processed by the algorithm.
- Some provision is made for including comments. This is usually done by enclosing each comment between a pair of special symbols such as /* and */.
- Certain key words that are common in high-level languages may be used: for example, read or enter to indicate an input operation; display, print, or write for output operations.
- Indentation is used to set off certain key blocks of instructions.

Very Short Answer Type Questions

1. Operating system is a [WBUT 2019]
   a) application software b) system software
   c) both (a) and (b) d) none of these

Answer: (b)

2. ALU is a part of [WBUT 2019]
   a) Memory b) input device c) CPU d) output device

Answer: (c)

3. A computer consists of the following parts [MODEL QUESTION]
   a) A Central Processing Unit b) A monitor for display
   c) A keyboard and mouse for input d) All of the above

Answer: (c)

4. The main circuit board of a computer is called [MODEL QUESTION]
   a) CPU b) RAM
   c) Secondary Storage d) Motherboard

Answer: (d)

5. An algorithm can be sequential, selective, iterative [MODEL QUESTION]
   a) True b) False

Answer: (a)

6. Which component is used to translate high level language into it's equivalent machine language? [MODEL QUESTION]
   a) Assembler b) Operating System
   c) Compiler d) Motherboard

Answer: (c)

7. Which algorithmic representation gives us the flexibility to evaluate multiple combination of conditions to produce an output? [MODEL QUESTION]
   a) Class Diagram b) Decision Functions
   c) Programming language d) Rules
   e) Decision Tables

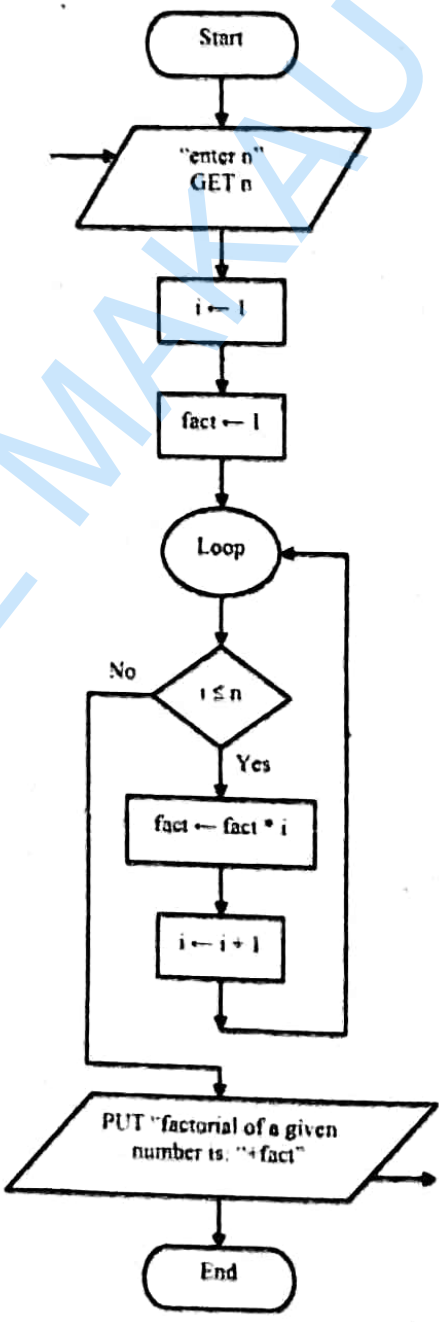
Answer: (e)

8. Flow chart is a one-way algorithmic representation. [MODEL QUESTION]
a) True b) False
Answer: (b)

9. Which component in a computer is primarily responsible in processing inputs? [MODEL QUESTION]
a) CPU b) RAM c) Hard Disk d) USB
Answer: (a)

Short Answer Type Questions

1. Draw flowchart to evaluate factorial of a numbers. [WBUT 2023]
Answer:



2. What are the different generations of computers? **[MODEL QUESTION]**

Answer:

There are five computer generations:

1) First generation (1942-1955): The first generation of computer used vacuum tubes as hardware components.

2) Second generation (1955-1964): The second generation of computer used transistor as a hardware component. The transistor, a smaller and more reliable successor to the vacuum tube, was invented in 1947.

3) Third generation (1964-1975): The third generation of computer used integrated circuit (1C) as a hardware instead of transistor, resistor, capacitor across one single chip of silicon.

4) Fourth generation (1975 onwards): The fourth generation of computer used Small Scale Integration (SSI), Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) as a hardware component.

5) Fifth generation (Yet to come): Scientists are now working on the fifth generation of computer. They aim to bring us machines with genuine I.Q., the ability to reason logically and with real knowledge of the world. Fifth generation will be totally different, totally novel, and totally new.

3. Explain the different types of storage devices used in a computer in terms of usage, advantages and disadvantages. **[MODEL QUESTION]**

Answer:

Storage devices are broadly classified into two categories:

- Primary Storage: RAM
- Secondary storage: Hard Disks

Primary Storage: In some data processing, all of the instructions and data are entered in primary storage (RAM). The computer completes its processing, results are presented, and the application ends. The computer may have sufficient primary storage to hold all the instructions and data that are needed during processing. However, in some applications, a computer's primary-storage capabilities are insufficient and unable to handle the instructions and data needed for processing. Another limitation in primary storage is its volatility. In many cases, a particular set of data is used more than once. When power is turned off, the data in primary storage are lost and require re-entry for each new use. This is time-consuming and costly way to process data. Often it is desirable to save the results of processing so that they can be used as input for further processing or printed as hard or soft copy later. So, because primary storage may not be large enough to hold all the required instructions and data, and because RAM is volatile and doesn't provide long-term memory, supplemental storage is necessary. The solution to these limitations is found in the use of secondary storage.

Secondary Memory: *Secondary storage* is the nonvolatile memory that is stored externally to the computer. A second storage medium is usually used for the storage of large amounts of data or for permanent or long-t storage of data or programs. Secondary

storage is also used for storing backups, or copies of data programs so that they are not permanently lost if primary-storage power is interrupted.

Different secondary storage media can be ranked according to the following criteria:
1. Retrieval speed: The access time of a storage device is the time it takes to locate and retrieve stored data. A fast access time is preferable for any storage media.
2. Storage capacity: A device's storage capacity is the ability to store data. A large storage capacity is desired.
3. Cost per bit of capacity: Low cost is preferred.

The secondary storage device can be again classified into three types:

- Magnetic tapes
- Magnetic disks
- Optical disks

Magnetic Tape
Magnetic tape is a one-half or one-fourth-inch ribbon of Mylar (a plastic like material) coated with a thin layer of iron-oxide material. It has been used as an input/storage medium for batch processing, and it remains a choice today for high speed, large volume applications. It is also a medium that is often selected to store large files that are sequentially accessed and processed. It's data density (the number of characters that can be stored in a given physical space) is high and its transfer rate (the speed with which data can be copied into processor storage) is fast.
Tape drive has an electromagnetic Read/Write head through which the tape passes and data is read and written onto it. Magnetic tape is usually divided into nine separate strips or tracks. Eight of these tracks are used for the codes, which represent data characters. And the ninth track is reserved for an error-checking bit.
Magnetic tape stores data sequentially, i.e., one after another. Between each record stored on the tape, there is a space called inter record gap (IRG), These gaps are automatically created when data are written on the tape. When record data are read from a moving tape into the processor, the movement stops when a gap is reached. The tape remains motionless until the present record is processed and then starts moving again lo enter the next record into the computer. This procedure is repeated until the file is processed. But what if there are a large number of very short records. To avoid such an inefficient situation, several short records are commonly combined into a block, tape block, and read into the processor as a single unit. This saves tape space and speeds data input.

Advantages of Magnetic Tape
1. High data density
2. Low cost and ease of handling

Limitations of Magnetic Tape
1. Lack of direct access to records
2. Environmental problems—sensitivity to dust, humidity and temperature levels

Magnetic Disk

A magnetic disk is a Mylar or metallic platter on which electronic data are stored. Unlike magnetic tapes, the data on magnetic disks can also be read randomly. The data are recorded as tiny invisible magnetic spots. Read/write heads are tiny electromagnets that can read, write, or erase the polarized spots that represent data on magnetic media. These heads are fastened to an arm in a disk storage device so; that they can be moved quickly and directly to any disk location to store or retrieve data.

The access time for data stored on a magnetic disk is determined by two factors:

1. The *seek-time,* i.e., the time required for positioning read/write head over the proper track
2. *The search (latency) time,* i.e., the time required for spinning the required data under the head.

Once the data have been accessed, they are copied from the disk to the processor for processing. The transfer rate depends on the density of the stored data and the rotational speed of the disk.

Data is represented on a disk using a 9-bit code similar to the EBCDIC code discussed for tapes. Each byte or character is represented longitudinally along a disk track by a 9-bit configuration. Magnetic disks come in various sizes. They can be portable or permanently mounted in their stc devices, disk drives. They can be made of rigid metal (Hard Disks) or flexible plastic (Floppy Diskettes). They can be divided into two broad categories:

1. Fixed Disks or Hard Disks
2. Floppy Diskettes

Optical disks

The optical storage device that most of us are familiar with is the compact disc (CD). A CD can store huge amounts of digital information (783 MB) on a very small surface that is incredibly inexpensive to manufacture. The design that makes this possible is a simple one: The CD surface is a mirror covered with billions of tiny holes that are arranged in a long, tightly wound spiral. The CD player reads the holes with a laser beam and interprets the information as bits of data.

4. What are the different computer languages? **[MODEL QUESTION]**

Answer:

Computer languages can be broadly classified into two types:

a) Low level language b) High level language

a) Low-level language

These are the languages, which directly interacts with computer memory to store and retrieve data. One such language is assembly language. It uses alphabetic mnemonics to indicate instructions and decimal numbers to represent data. Programs written in these languages are read into the computer and converted into binary machine code by a translator program permanently stored within the computer. Assembly languages are very

much dependent on the specific machine, requiring the programmer to have detailed knowledge of that machine. Such languages are still reasonably efficient in terms of machine operation, and became collectively known as *low-level languages*.

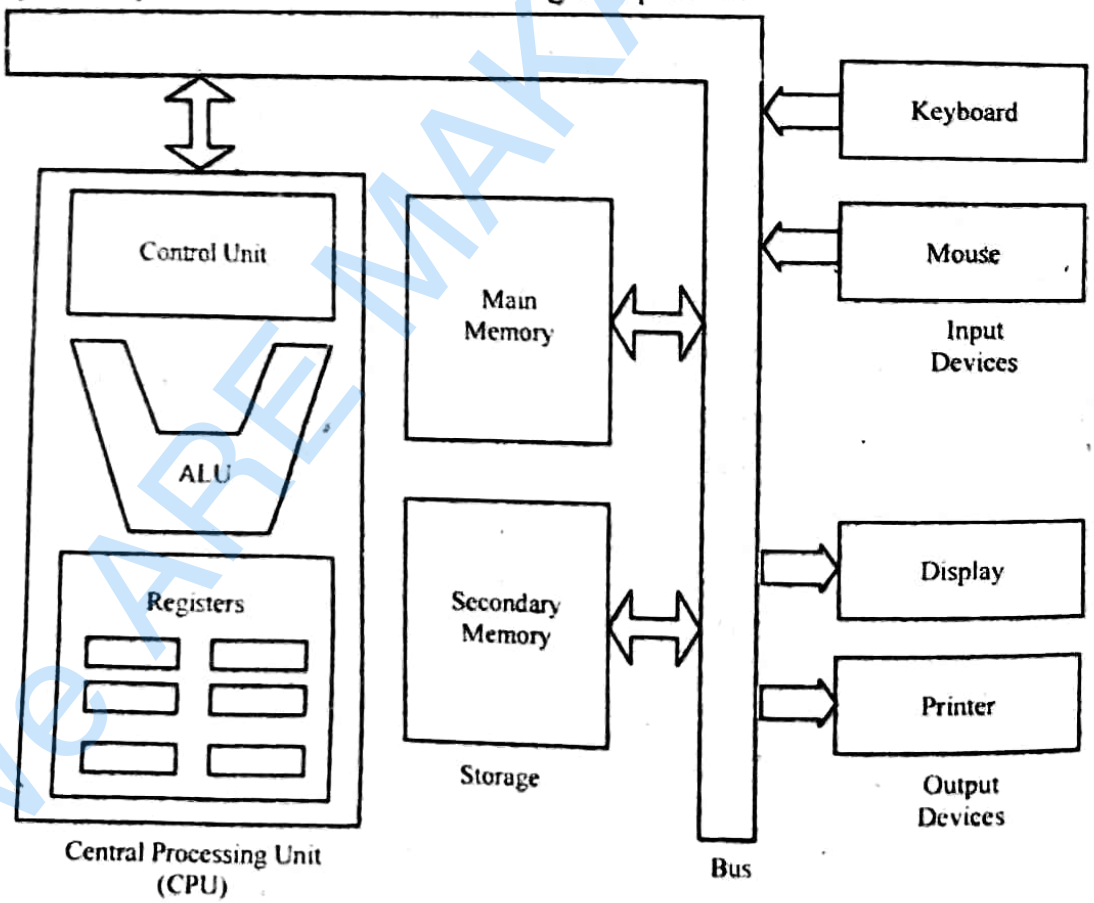
b) High-level language

These are the user-friendly languages targeted for easier development without any detailed knowledge of the computer hardware thereby reducing the bulk of programming complexity. Programs written in these languages are read into the computer and translated into machine code by in built translation programs. Usually, this translation takes place in more than one stage, producing a compiled-coded version of the program as an intermediate stage. Languages like BASIC, PASCAL, C, JAVA etc. are all examples of high level languages.

5. Describe the function of different components of a conventional digital computer with a suitable diagram. **[MODEL QUESTION]**

Answer:

Typically, a computer consists of the following components



An anatomy of a computer can be categorized into three primary units viz.

- Central Processing Unit
- Storage
- Input/Output

Central Processing Unit (CPU)

The Central Processing Unit (CPU) is the electronic circuit responsible for executing the instructions of a computer program.It is sometimes referred to as the microprocessor or processor.

The CPU consists of ALU, CU and a variety of registers.

Registers

Registers are high speed storage areas in the CPU. All data must be stored in a register before it can be processed.

MAR	Memory Address Register	Holds the memory location of data that needs to be accessed
MDR	Memory Data Register	Holds data that is being transferred to or from memory
AC	Accumulator	Where intermediate arithmetic and logic results are stored
PC	Program Counter	Contains the address of the next instruction to be executed
CIR	Current Instruction Register	Contains the current instruction during processing

Arithmetic and Logic Unit (ALU)

The ALU allows arithmetic (add, subtract etc) and logic (AND, OR, NOT etc) operations to be carried out.

Control Unit (CU)

The control unit controls the operation of the computer's ALU, memory and input/output devices, telling them how to respond to the program instructions it has just read and interpreted from the memory unit. The control unit also provides the timing and control signals required by other computer components.

Buses

Buses are the wires that connects all major internal components to the CPU and memory. The data is transmitted from one part of a computer to another through the bus.

A standard CPU system bus is comprised of a control bus, data bus and address bus.

Address Bus	Carries the addresses of data (but not the data) between the processor and memory
Data Bus	Carries data between the processor, the memory unit and the input/output devices
Control Bus	Carries control signals/commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer

Memory Unit

The memory unit consists of RAM, sometimes referred to as primary or main memory. Unlike a hard drive (secondary memory), this memory is fast and also directly accessible by the CPU.

RAM is split into partitions. Each partition consists of an address and its contents (both in binary form).

The address will uniquely identify every location in the memory.

Loading data from permanent memory (hard drive), into the faster and directly accessible temporary memory (RAM), allows the CPU to operate much quicker.

Cache (pronounced as cash) is a volatile memory i.e. contents are erased on each power off. It stands between main memory and microprocessor. It is a small fast memory, which is used to store active portions of program and data.

Long Answer Type Questions

1. a) What is an Operating system?
b) What are the functions of operation system?
c) What is booting?
d) What are the differences between compiler and interpreter? **[WBUT 2023]**

Answer:

a) An *operating system* is a system program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs.

b) The functions of an operating system is as summarized below:

- **Resource allocator:** A computer system has many resources (hardware and software) that may be required to solve a problem such as CPU time, memory space, the file storage, I/O devices and so on. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for tasks.
- **Control program:** A control program controls the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices such as printer, floppy disks etc.

c) During the boot process, the computer goes through multiple steps to ensure the computer hardware works correctly and the necessary software can be loaded. When booting, the computer performs the following tasks, mostly in the background.
Here are the 6 steps in the booting process that take place during the booting process in OS.

Step 1: BIOS is loaded
The first step involves turning on the power. It powers the essential parts, including the processor and BIOS, with electricity.

Step 2: BIOS: Power on Self-Test
It is the first test run by the BIOS. Additionally, this test runs a preliminary examination of the computer's main memory, disk drives, input/output devices, etc. Moreover, the system emits a beep sound in case of any errors.

Step 3: Loading of OS
The bootable sequence stored in the CMOS (Common Metal Oxide Semiconductor) is read by BIOS following the successful completion of POST. Based on the bootable sequence, it will look for the Master Boot Record (MBR) on bootable devices such as floppy disks, CD-ROMs, and hard disks.
A message saying "No Boot Device Found" will be displayed, and the system will crash if MBR is not found in any of them. If MBR is discovered, the BIOS will launch

a unique application software called the Boot Loader, which will ultimately launch the operating system.

Step 4: System Configuration

Device drivers are put into the memory after the OS is loaded to ensure the proper operation of our gadgets.

Step 5: Loading System Utilities

In this step, system utilities like antivirus and volume control are loaded into the memory.

Step 6: User Authentication

The system will prompt the user to input their credentials if any user authentication is configured. Once the system has received valid credentials, it will typically launch the GUI shell or the CLI shell.

d)

Compiler	Interpreter
A compiler translate the compiler source program in a single line	An interpreter translate the source program line by line
It is faster	It is slower
It consumes less time	It consumes more time them compiler
It is more efficient	It is less efficient
Compilers are larger in size.	Interpreters are smaller than compilers

2. a) Convert Binary to Decimal of $(1101.01)_2$.

b) What is 2's complement? Show by example.

c) Using 2's Complement do $10101_2 - 00111_2$.

d) Convert Decimal to Octal $(5673)_{10}$. **[WBUT 2023]**

Answer:

a) $(1101.01)_2 = (1\times 2^3)+(1\times 2^2)+(0\times 2^1)+(1\times 2^0)+(0\times 2^{-1})+(1\times 2^{-2}) = (13.25)_{10}$

b) In number system, the **2s complement** is an operation that is frequently used to encode the negative and positive numbers in the form of binary variables. It is usually used in logic gates such as AND gate, NOT gate, OR gate, etc.

While the working of 1s complement is almost similar to twos complement but it is only applicable for the positive binary, decimal, & hexadecimal numbers. The reason of creating a new operation is just for the representation of the negative terms.

The 2s complement is a perfect example of true complement (radix complement). Twos complement can be evaluated by adding 1 to the least significant bit of the 1s complement. While the one's complement is determined by taking the transpose of the binary numbers.

To transform the binary variables into 2s complement, keep an eye on the following steps.

1. Firstly, take a number in the form of 0 & 1.

2. Invert all the binary variables such as transform all the 1s into 0s and all the 0s into 1s. the inversion of the binary variable is said to be the transpose. The process gives you the result of the one's complement.
3. Add one to the LSB of the transpose of the binary variables.
4. The final result will be the 2s complement of the binary variable.

Example:
Transform 110011001010 into twos complement.
Solution:
Step - 1: Firstly, take a number in the form of 0 & 1.
110011001010
Step - 2: Invert all the binary variables such as transform all the 1s into 0s and all the 0s into 1s.
001100110101
Step - 3: Add one to the LSB of the transpose of the binary variables to get the result in twos complement.
Therefore,
001100110110 is the twos complement of the given binary number.

c)

$$\begin{array}{rr} 10101 & 10101 \\ -00111 & +11001 \\ \hline 01110 \leftarrow \boxed{X} & 01110 \end{array}$$

d)

	Remainders
8 | 5,673	1
8 | 709	5
8 | 88	0
8 | 11	3
8 | 1	1
0	

3. Write short note on the following: **[WBUT 2019]**
a) Compiler
b) Operating System
Answer:
a) Compiler:
A **compiler** is a systems program that translates the source language (high-level language like C, C++, C# and Java) into a machine equivalent language (bits) known as the intermediate object code understandable by the computer hardware. In short a compiler is a translator for a high level language.

Executing a program written in a high-level programming language is basically a two-step process:

- The source program must first be compiled that is translated into object program.
- Then the object program is loaded into memory and executed.

b) Operating System:

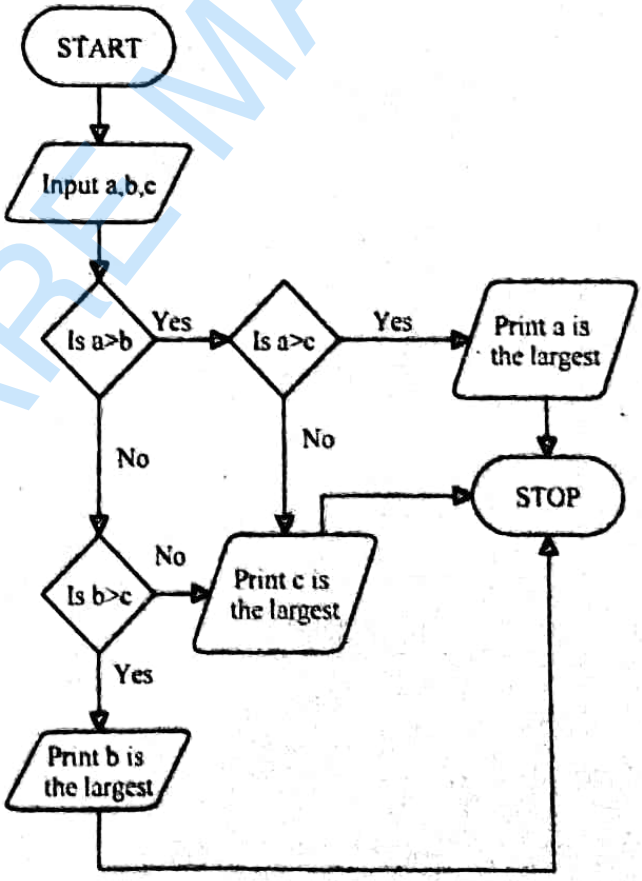
An *operating system* is a system program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs.

The role of an operating system is as summarized below:

- **Resource allocator:** A computer system has many resources (hardware and software) that may be required to solve a problem such as CPU time, memory space, the file storage, I/O devices and so on. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for tasks.
- **Control program:** A control program controls the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices such as printer, floppy disks etc.

4. Draw a flowchart to find the largest among three numbers taken as input.
[MODEL QUESTION]

Answer:



Algorithm in simple English

Step 1: Initialize three variables.
Step 2: Take input and assign them to these variables
Step 3: Compare each variable to the other and find the variable holding the largest value
Step 4: Print the variable holding the largest value

5. Draw a flowchart for a Fibonacci sequence.
0 1 1 2 3 5 8 13 21 34 ..... n terms. **[MODEL QUESTION]**

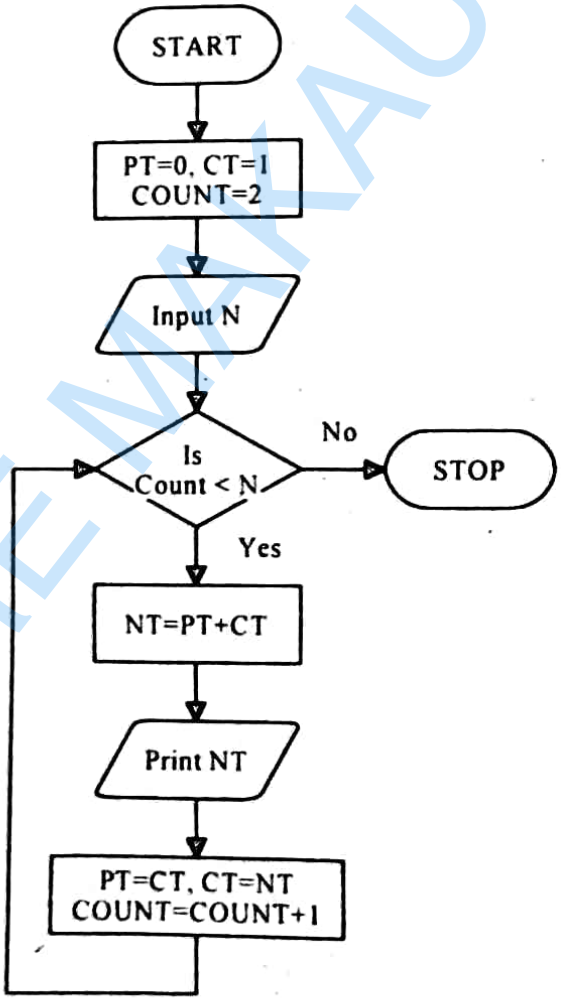
Answer:

1 — Previous term (PT)

1 — Common term (CT)

2 — New term (NT)

NT= PT+CT



ARITHMETIC EXPRESSIONS AND PRECEDENCE

Chapter at a Glance

Operators in programming language

In most programming languages, operators can be classified into the following categories

Types of Operators	Description
Arithmetic_operators	These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus
Assignment_operators	These are used to assign the values for the variables in C programs.
Relational operators	These operators are used to compare the value of two variables.
Logical operators	These operators are used to perform logical operations on the given two variables.
Bit wise operators	These operators are used to perform bit operations on given two variables.
Conditional (ternary) operators	Conditional operators return one value if condition is true and returns another value is condition is false.
Increment/decrement operators	These operators are used to either increase or decrease the value of the variable by one.
Special operators	&, *, sizeof( ) and ternary operators.

Arithmetic Operators

The basic operators for performing arithmetic operations are:

+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	modulus (remainder after division)

Assignment Operators

For example

```
int c = 2, d = 3, e = 4, f = 6, g = 8
```

Assignment Operator	Sample expression	Explanation	Assigns
+=	c += 2	c = c + 2	4 to c
-=	d -= 1	d = d – 1	1 to d
*=	e *= 2	e = e * 2	8 to e
/=	f /= 3	f = f / 3	2 to f
%=	g %= 4	g = g % 4	2 to g

Relational Operators

Relational operators are used to compare two operators depending on their relation.

For example, for comparing the price of two things. The value of a relational operator is either 1 or 0. If the specified relation is true then 1 else 0.

For example ifx>y, then > is a relational operator and it will return 1 if x greater than y else it will return 0.

EXPRESSION	TRUE IF
x = = y	x is equal to y
x ! = y	x is not equal to y
x<y	x is less than y
x>y	x is greater than y
x<= y	x is less than or equal to y
x >= y	x is greater than or equal to y

Conditional Operators

The conditional operator in C is the *Ternary Operator*. The general syntax for such an operator is

expression1 ? expression2 : expression3

It means that, if expression 1 is true (i.e., if its value is non zero), then the value returned will be expression 2, otherwise the value returned will be expression 3.

Let us consider the following example

y = (x>5 ?3 : 4)

This statement will store 3 in y, if x is greater than 5, otherwise it will store 4 in y.

Bitwise Operators

The different types of Bitwise operators available in C language are:

OPERATORS	MEANING
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	shift left
>>	shift right
~	bitwise complement

A bitwise operator operates on each bit of data. These operators are used for testing and complementing or shifting bits to the right or left. Usually bitwise operators are not useful in cases of float or double variables.

One of the most widely used representations of numerical data is the binary coded decimal (BCD) form in which a 4-bit binary number represents each integer of a decimal number. Let us consider the following example to represent 0 to 15 in binary through a truth table.

Binary Bits				Decimal
8	4	2	1	
0	0	0	0	0
0	0	0	0	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

The Bitwise 'And' Operation

Truth Table

X	OPERATOR	Y	OUTPUT
0	&	0	0
0	&	1	0
1	&	0	0
1	&	1	1

The BCD equivalent of x =10 and y = 6 is

	8	4	2	1
x = 10	1	0	1	0
y = 6	0	1	1	0
x & y	0	0	1	0

Therefore, x & y = 2.

The Bitwise 'OR' Operation

Truth Table

X	OPERATOR	Y	OUTPUT
0	|	0	0
0	|	1	1
1	|	0	1
1	|	1	1

The BCD equivalent of x =10 and y = 6 is

	8	4	2	1
x=10	1	0	1	0
y = 6	0	1	1	0
x | y	1	1	1	0

Therefore, x | y = 14.

The Bitwise 'XOR' Operation

Truth Table

X	OPERATOR	Y	OUTPUT
0	^	0	0
0	^	1	1
1	^	0	1
1	^	1	0

The BCD equivalent of x =10 and y = 6 is

	8	4	2	1
x =10	1	0	1	0
y = 6	0	1	1	0
x ^ y	1	1	0	0

Therefore, x ^ y = 12.

The Bitwise Left-Shift (<<) Operator

Let us consider the expression z = x << 2

Before Left shift x is

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

After Left shift x is

1024	512	128	64	32	16	8	4	2	1
0	0	0	0	1	0	1	0	0	0

All the bits are shifted 2 positions to the left. The original leftmost bits 0 and 0 before left shift are dropped. Hence z = x <<2 is 40

The Bitwise Right-Shift (>>) Operator

Let us consider the expression z = x >> 2

Before Right shift x is

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

After Right shift x is

32	16	8	4	2	1
0	0	0	0	1	0

All the bits are shifted 2 positions to the right. The original rightmost bits are dropped. The new position of bits is shown above
Hence z = x >> 2 is 2

Comma Operators

The comma operator allows us to write a set of expressions using comma. For example, the expression
int x = 10, y = 6, z= 4; is valid
and we can avoid writing int x=10; int y = 6; int z =4. However, initializing variables with a comma separator is not a good programming practice.

sizeOf operator

The sizeof operator when used with an operand returns the number of bytes the operand occupies. This operand may be a variable, a constant or a data type qualifier.
The size of operator is normally used to determine the length of arrays and structures when programmer does not know their sizes. It is also used to allocate memory space dynamically to variables during execution of program.

Precedence and Order of Evaluation

Operators specify an order of evaluation to be performed on one of the following:

- One operand (unary operator)
- Two operands (binary operator)
- Three operands (ternary operator)

Operator precedence determines which operation will be performed first in a group of operators with different precedence. For instance 5 + 3 * 2 is calculated as 5 + (3 * 2), giving 11, and not as (5 + 3) * 2, giving 16.

Operator associativity rules define the order in which adjacent operators with the same precedence level are evaluated. For instance the expression 8 - 3 - 2 is calculated as (8 - 3) - 2, giving 3, and not as 8 - (3 - 2), giving 7. In this case we say that subtraction is left associative. It means that the left most subtraction must be done first.

Tabular Representation

The Table below summarizes the rules for precedence and associativity of all operators.

- Operators on the same line have the same precedence.
- Blocks are in order of decreasing precedence.

Rows in the same block have the same precedence

Description	Operator	Associativity
Increment/decrement	+ +/ - -	Right to left
Negation (NOT)	!	Right to left
Unary Minus	-	Right to left
Size in bytes	sizeof	Right to left
Multiplication	*	Left to right
Division	/	Left to right
Mod	%	Left to right
Addition	+	Left to right
Subtraction	-	Left to right
Less than	<	Left to right
Less than or equal to	< =	Left to right
Greater than	>	Left to right
Greater than or equal to	> =	Left to right
Equal to	= =	Left to right
Not equal to	! =	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional (Ternary)	? :	Right to left
Assignment	= % = + = - = * = / =	Right to left
Comma	,	Left to right

Very Short Answer Type Questions

1. What is the value of x, if $x = (3 \wedge 6) + (2 \wedge 2)$? **[WBUT 2019]**

a) 733 **b) 5** **c) 18** **d) None of these**

Answer: (b)

2. Output of the following code is **[WBUT 2019]**

```
int i=5;
printf("%d%d%d%d", ++i, i--, --i, i++);
```

a) 5555 **b) 6645** **c) 6545** **d) 5566**

Answer: (a)

3. Which of the following is a bitwise operator? **[WBUT 2019]**

a) < **b) <<** **c) !** **d) | |**

Answer: (b)

4. What is the value of the following if $a = 2$? **[WBUT 2023]**

```
printf("%d",-a);
```

Answer: -2

5. Which of the following is not a valid arithmetic operation? **[MODEL QUESTION]**

a) `a *= 20;` **b)** `a /= 30;` **c)** `a %= 40;` **d)** `a! = 50;`

Answer: (d)

6. What is the value of x in the program below? **[MODEL QUESTION]**

```
void main()
{
        int x = 4 *5 / 2 + 9;
}
```

a) 6.75 b) 1.85 c) 19 d) 3

Answer: (c)

7. What is the output of the following code? **[MODEL QUESTION]**

```
void main()
{
        int a = 5;
        int b = ++a + a++ + --a;
        printf("Value of b is %d", b);
}
```

a) Value of b is 16 b) Value of b is 19
c) Value of b is 15 d) Value of b is 18

Answer: (b)

8. What is the output of the following code? **[MODEL QUESTION]**

```
void main()
{
int x = 0;
      if (x = 0)
        printf("Its zero\n");
      else
        printf("Its not zero\n");
}
```

a) Its not zero b) Its zero c) Run time error d) None

Answer: (a)

9. What is the output of the following code? **[MODEL QUESTION]**

```
int main()
{
      int i = -5;
      int k = i %4;
      printf("%d\n", k);
}
```

a) Compile time error b) -1 c) 1 d) None

Answer: (b)

10. The precedence of arithmetic operators is (from highest to lowest)? **[MODEL QUESTION]**

a) %, *, /, +, -
b) %, +, /, *, -
c) +, -, %, *, /
d) %, +, -, *, /

Answer: (a)

Short Answer Type Questions

1. Write a program to find roots of a quadratic equation. [WBUT 2023]

Answer:

```
#include <math.h>
#include <stdio.h>
int main() {
    double a, b, c, discriminant, root1, root2, realPart,
imagPart;
    printf("Enter coefficients a, b and c: ");
    scanf("%lf %lf %lf", &a, &b, &c);

    discriminant = b * b - 4 * a * c;

    // condition for real and different roots
    if (discriminant > 0) {
        root1 = (-b + sqrt(discriminant)) / (2 * a);
        root2 = (-b - sqrt(discriminant)) / (2 * a);
        printf("root1 = %.2lf and root2 = %.2lf", root1, root2);
    }

    // condition for real and equal roots
    else if (discriminant == 0) {
        root1 = root2 = -b / (2 * a);
        printf("root1 = root2 = %.2lf;", root1);
    }

    // if roots are not real
    else {
        realPart = -b / (2 * a);
        imagPart = sqrt(-discriminant) / (2 * a);
        printf("root1 = %.2lf+%.2lfi and root2 = %.2f-%.2fi",
realPart, imagPart, realPart, imagPart);
    }

    return 0;
}
```

2. What is conditional operator? [WBUT 2023]

Answer:

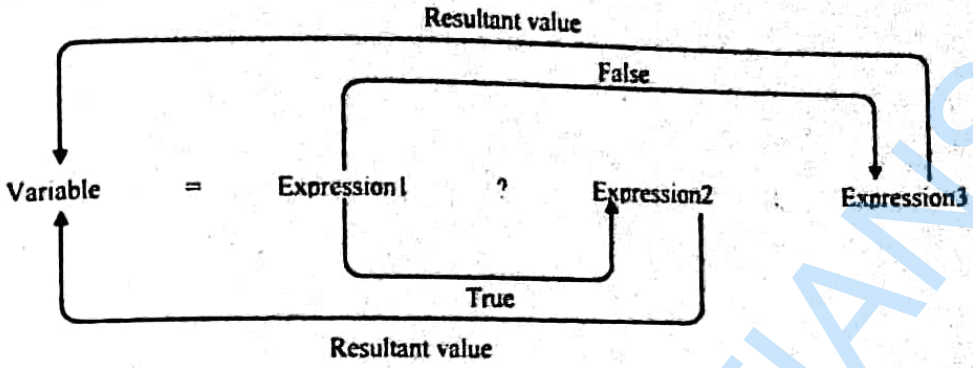
The conditional operator is also known as a **ternary operator.** The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

Expression1? expression2: expression3;



3. Write a program in C to explain ternary operator. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
int main()
{
      char February;
      int days;
      printf("If this year is leap year, enter 1. If not enter any integer: ");
      scanf("%c", &February);
      // If test condition (February == '1') is true, days equal to 29.
      // If test condition (February == '1') is false, days equal to 28.
      days = (February == '1') ? 29 : 28;

      printf("Number of days in February = %d", days);
      return 0;
}
```

4. Write a program in C to explain sizeof operator. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
int main()
{
      int a, e[10];
      float b;
      double c;
      char d;
      printf("Size of int=%lu bytes\n", sizeof(a));
      printf("Size of float=%lu bytes\n", sizeof(b));
      printf("Size of double=%lu bytes\n", sizeof(c));
      printf("Size of char=%lu byte\n", sizeof(d));
      printf("Size of integer type array having 10 elements = %lu
      bytes\n", sizeof(e));
      return 0;
```

5. Write a program in C to explain right and left shift operators. [MODEL QUESTION]

Answer:

```
#include <stdio.h>
int main()
{
    int num=212, i;
    for (i=0; i<=2; ++i)
        printf("Right shift by %d: %d\n", i, num>>i);
     printf("\n");
     for (i=0; i<=2; ++i)
        printf("Left shift by %d: %d\n", i, num<<i);

     return 0;
}
```

Long Answer Type Questions

1. Write short note on Type Casting. [WBUT 2019]

Answer:

Type casting n C language is used to convert a variable from one data type to another data type. New data type should be mentioned before the variable name or value in brackets which to be typecast.

It is best practice to convert lower data type to higher data type to avoid data loss. Like int to float.

Data will be truncated when higher data type is converted to lower. For example, if float is converted to int. data which is present after decimal point will be lost.

2. Write a program to swap two nos. without using third variable. [MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(void)
{
  int a, b;
printf("Enter first no. :  ");
scanf("%d", &a);
printf("Enter second no. :  ");
scanf("%d", &b);
printf("\nBefore swapping first no. = %d", a);
printf("\nBefore swapping second no. = %d", b);
  if (a > b)
  {
    a = a + b;
    b = a - b;
    a = a - b;
  }
```

```
  else
  {
    b = b - a;
    a = a + b;
    b = a - b;
  }
printf("\nAfter swapping first no. = %d", a);
printf("\nAfter swapping second no. = %d", b);
//getch();
  return EXIT_SUCCESS;
}
```

3. Write a program to display WXYZ in the following way: **[MODEL QUESTION]**

```
W
W X
W X Y
W X Y Z
W X Y
W X
W
```

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(void)
{
  int i, j;
  int m, k = 4;
  char st[] = "WXYZ";
  for (i = 0; i< 4; i++)
  {
    for (m = 0; m < k; m++)
    {
      printf(" ");
    }
    for (j = 0; j <= i; j++)
    {
    printf("%2c", st[j]);
    }
      printf("\n");
    k--;
  }
  k = k + 2;
  for (i = 3; i> 0; i--)
  {
    for (m = 0; m < k; m++)
    {
      printf(" ");
    }
```

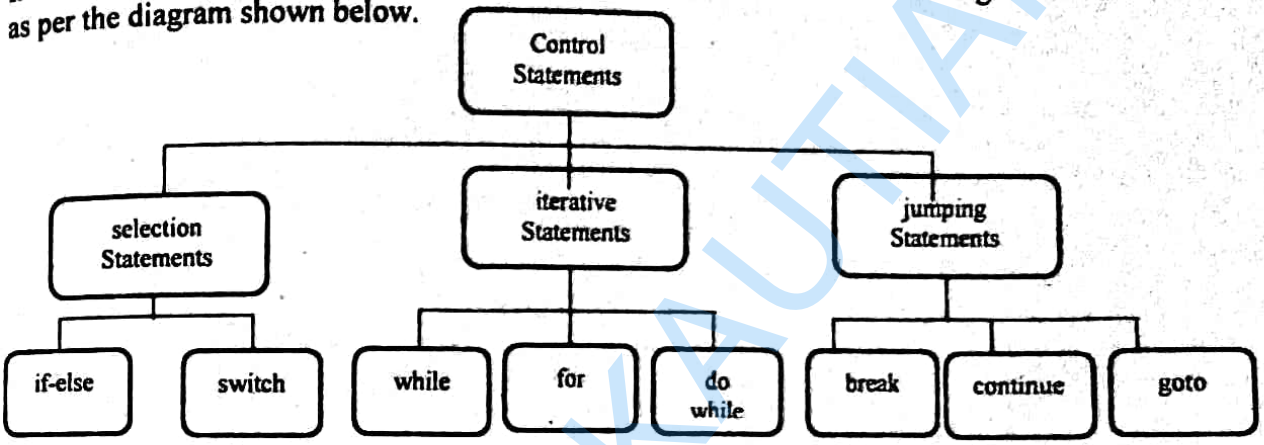
```
    for (j = 0; j <i; j++)
    {
      printf("%2c", st[j]);
    }
      printf("\n");
    k++;
  }
//getch();
  return EXIT_SUCCESS;
}
```

CONDITIONAL BRANCHING AND LOOPS

Chapter at a Glance

Introduction

In C language the decision making is done using one or more of the following control statements as per the diagram shown below.



if-else statement

If statement is a conditional branching statement. In conditional branching statement a condition is evaluated, if it is evaluate true a group of statement is executed else other statements are executed.

```
if (x)
    printf("True value");
  else
    printf("False value");
```

Nested-if

```
if (x)
  {
    if (x > 3)
      printf("x is greater than 3");
    else
      printf("x is lesser than 3");
  }
  else
  {
    printf("You must have entered zero!");
  }
```

switch-case statement

Switch statements simulate the use of multiple if statement with case blocks associated with switch conditions.

```
switch(x)
      int i = 2;
  switch (i)
```

```
{
  case 1:
    printf(" \n Inside case 1");
  case 2:
    printf("\n Inside case 2");
  case 3:
    printf("\n Inside case 3");
  default:
    printf("\n Inside default");
}
```

while loop

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

```
int x = 2;
      while(x < 1000)
      {
            printf("%d\n", x);
            x = x * 2;
      }
```

for loop

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

```
for (i = 0; i < 10; i++)
      printf("i is %d\n", i);
```

do-while loop

Is also used for looping. In this case the loop condition is tested at the end of the body of the loop. Hence the loop is executed at least one.

```
int i = 4;
do
{
  i = i+1;
} while (i<10);
```

break statement

The break statement is a jump instruction and can be used inside a switch construct, for loop, while loop and do-while loop.

continue statement

Continue statement is a jump statement. The continue statement can be used only inside for loop, while loop and do-while loop.

goto statement

The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program unconditionally.

Very Short Answer Type Questions

1. The type of the controlling expression of a switch statement cannot be of the type [WBUT 2019]

a) int b) char c) float d) none of these

Answer: (b) and (c)

2. What can be output of the following code? [WBUT 2019]

```
enum Months {JAN=10, FEB, MAR, APR};
enum Months X=FEB;
switch(X){
      case 1:printf("January");
             break;
      case 11:printf("February");
             break;
}
```

a) January b) February
c) Generates compilation error d) Results in runtime error

Answer: (b)

3. What will be output of the following code? [WBUT 2019]

```
int i;
for(i=0; i<5; i++) printf("%d\n", 1L<<i);
```

a) 5, 4, 3, 2, 1 b) 0, 1, 2, 3, 4 c) 0, 1, 2, 4, 8 d) 1, 2, 4, 8, 16

Answer: (d)

4. What is the use of continue in loop? [WBUT 2023]

Answer:

The continue statement terminates execution of the statements in the current iteration of the current or labeled loop, and continues execution of the loop with the next iteration.

5. What is nested if-else? [WBUT 2023]

Answer:

In C programming, a nested if-else statement is when you have an if-else block inside another if or else block.

6. What is the use of goto statement? [WBUT 2023]

Answer:

The goto statement can be used to alter the normal flow of control in a program.

7. What is goto statement in C? [WBUT 2023]

Answer:

The goto statement can be used to alter the normal flow of control in a program.

8. What is the output of a? [MODEL QUESTION]

```
int main()
{
    int a = 0, i = 0, b;
    for (i = 0; i < 5; i++)
   {
      a++;
      continue;
    }
    printf("%d", a);
}
```

a) 2 b) 3 c) 4 d) 5

Answer: (d)

9. What is the output of this C code? [MODEL QUESTION]

```
int main() {
       int a = 0, i = 0, b;
       for (i = 0; i < 5; i++) {

             a++;
             if (i == 3)
                   break;
       }
       printf("%d", a);
}
```

a) 2 b) 3 c) 4 d) 5

Answer: (c)

10. The keyword "break" cannot be simply used within: [MODEL QUESTION]

a) do-while b) if-else c) for d) while

Answer: (b)

11. What is the output of this C code? [MODEL QUESTION]

```
int main()
    {
        int i = 0;
       do
       {
           i++;
           if (i == 2)
               continue;
               printf("In while loop ");
       } while (i < 2);
       printf("%d\n", i);
   }
```

a) In while loop 2 b) In while loop c) In while loop 3 d) Infinite loop

Answer: (a)

Short Answer Type Questions

1. Write a C program to check whether an input number is Armstrong Number or not. An Armstrong number is a number that is the sum of its own digits each raised to the power of the number of digits. **[WBUT 2019]**

Answer:

```
#include <stdio.h>
/* Function declarations */
int isArmstrongNumber(int num);
void printArmstrong(int start, int end);

int main() {
  int start, end;
  printf("Enter lower limit to print armstrong numbers: ");
  scanf("%d", & start);
  printf("Enter upper limit to print armstrong numbers: ");
  scanf("%d", & end);

  printf("All armstrong numbers between %d to %d are: \n", start,
end);
  printArmstrong(start, end);

  return 0;
}
/**
 * Check whether the given number is armstrong number or not.
 * Returns 1 if the number is armstrong otherwise 0.
 */
int isArmstrongNumber(int num) {
  int temp, lastDigit, sum;
  temp = num;
  sum = 0;

  /* Calculate sum of cube of digits */
  while (temp != 0) {
    lastDigit = temp % 10;
    sum += lastDigit * lastDigit * lastDigit;
    temp /= 10;
  }
  /*
   * Check if sum of cube of digits equals to original number. */
  if (num == sum)
    return 1;
  else
    return 0;
}
/**
 * Print all armstrong numbers between start and end.
```

```
 */
void printArmstrong(int start, int end) {
  /*
   * Iterates from start to end and print the current number
   * if it is armstrong
   */
  while (start <= end) {
    if (isArmstrongNumber(start)) {
      printf("%d, ", start);
    }

    start++;
  }
}
```

2. Differentiate do-while and while loop with suitable example. **[WBUT 2019]**

Answer: ***Refer to Chapter at a Glance.***

3. What will be the output of the following codes with proper reason? **[WBUT 2019]**

i)

```
#include<stdio.h>
int main(){
        int i, j;
        i=2, j=2;
        while(--i &&j++)
          printf("%d%d", i, j);
        return 0;
}
```

Answer: –13

ii)

```
#include<stdio.h>
int main(){
        float me=2.1;
        double you=2.1;
        if (me==you) printf("Hello");
        else printf("Hi");
        return 0;
}
```

Answer: Hi

3. What will be the output of the following program segment? Explain. **[MODEL QUESTION]**

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
  int i;
  for (i =- 1; i<= 10; i++)
```

```
  {
    if (i< 5)
      continue;
    else
      break;
  }
printf(" Gets printed only  once");
getch();
  return EXIT_SUCCESS;
}
```

Answer:

Output No output

Explanation

The "if condition" inside the for loop satisfies for i = -1 to 4. As a result the continue statement forces the "for loop" to execute without entering the else part. As soon as the value of i becomes 5; the condition fails, enters the else part. But the first statement is break. Therefore the entire control comes out from the for loop & the printf( ) is never executed.

4. Compare between while, for and do-while loop and draw the flow chart representation with a suitable example. **[MODEL QUESTION]**

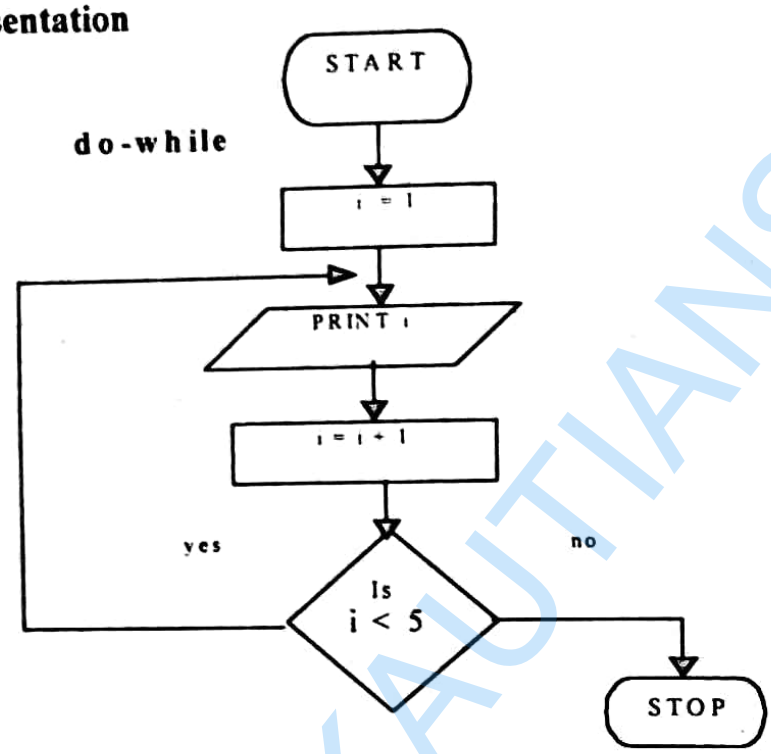
Answer:

while loop	for loop	do-while loop

main ()
{
int i
i = 1 ;
while (i< 10)
{
printf("%d\n",i);
i = i + 1 ;
}
}
	main ()
	{
	int i
	for (i = 1; i<10; i++)
	{
	printf (%d\n", i);
	}
	}
		main()
		{
		int i ;
		i = 1 ;
		do
		{
		print f ("%d\n", i);
		i = i + 1;
		} while (i< 10);
		}

Output : 1 2 3 4 5 6 7 8 9	**Output :** 1 2 3 4 5 6 7 8 9	**Output :** 1 2 3 4 5 6 7 8 9
Initialization, testing and increment of the loop counter i is done in separate steps.	Initialization, testing and increment of the loop counter i is done in the same step.	Initialization testing and increment of the loop counter i is done in separate step.
The condition is tested before executing body of the loop	The condition is tested before executing body of the loop	The condition is tested after executing body of the loop

Flow chart representation



5. What will be the output of the following program segment? [MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
int j, x = 0;
for (j = 0; j <= 5; j ++ )
  {
    switch (j - 1)
    {
      case 0:
      case -1: x += 1;
        break;
      case 1:
      case 2:
      case 3:
        x += 2;
        break;
      default:
        x += 3;
    }
printf("x = %d\n", x);
  }
getch();
  return EXIT_SUCCESS;
}
```

Output

x = 1
x = 2
x = 4
x = 6
x = 9
x = 11

Explanation

The break statement causes the control to come out prematurely from the switch-case block not from the for loop. The for loop iterates so long the conditions are satisfied and accordingly the switch case block is executed

j	j – 1	Case satisfied	Value of x
0	-1	Case –1	1
1	0	Case 0	2
2	1	Case 1	4
3	2	Case 2	6
4	3	Case 3	8
5	4	default	11

Long Answer Type Questions

1. a) Write a C program to print the following pattern for n number of rows, where n is given as input. **[WBUT 2019]**

```
*
**
****
******
********
**********
```

Answer:

```
void main()
{
  int rows, i = 0;
  printf ("Enter the rows :");
  scanf ("%d", &rows);
  for (i = 0; i < rows; i++)
    {
      int j, k;
      j = i * 2;
      if (j == 0)
      printf ("\n*");
      else
      {
        printf ("\n");
        for (k = 0; k < j; k++)
          {
            printf (" ");
```

```
            printf ("*");
          }
        }
      }
}
```

b) What is the difference between break and continue statement? Explain with example. **[WBUT 2019]**

Answer:

A break statement terminates a loop (for, while, do-while, case) from any further iteration. This statement is useful when based on a certain condition (if statement), you don't want to proceed with further iteration.

```
intmain()
{
int i = 0;
for (i = 0; i < 10; i++)
  {
if(i==5)
break;
   }
}
```

The continue statement allows a loop to iterate by skipping some conditional blocks. For example you may want to execute something when a loop iterates to even numbers. Odd numbers you want to skip. In such a scenario you use continue.

```
intmain()
{
int i = 0;
for (i = 0; i < 10; i++)
  {
if(i==0 || i%2 !=0)
continue;
else
printf("%d", i);
   }
}
```

2. a) Write a program to check a number is even or odd.
b) Write a program to check a number is prime or not.
c) Add all even numbers from an array. **[WBUT 2023]**

Answer:

a)

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
```

```
    // true if num is perfectly divisible by 2
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```

b)

```
#include <stdio.h>

int main() {

  int n, i, flag = 0;
  printf("Enter a positive integer: ");
  scanf("%d", &n);

  // 0 and 1 are not prime numbers
  // change flag to 1 for non-prime number
  if (n == 0 || n == 1)
    flag = 1;

  for (i = 2; i <= n / 2; ++i) {

    // if n is divisible by i, then n is not prime
    // change flag to 1 for non-prime number
    if (n % i == 0) {
      flag = 1;
      break;
    }
  }

  // flag is 0 for prime numbers
  if (flag == 0)
    printf("%d is a prime number.", n);
  else
    printf("%d is not a prime number.", n);

  return 0;
}
```

c)

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

    int a[10],i,sum=0;
    printf("Enter upto 5 Values: ");
```

```
    for(i=0; i<5; i++)
        scanf("%d",&a[i]);
    for(i=0; i<5; i++)
    {
        if(a[i]%2==0)
            sum=sum+a[i];
    }
    printf("Total Sum of Even values is: %d ",sum);
}
```

3. Write a program in C which finds the largest among three number (integers) taken as input. **[MODEL QUESTION]**

Answer:

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

4. Write a program in C, which illustrates the do-while loop. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
int main(void)
{
  int x = 4;
  do
  {
    x = x + 1;
printf("%3d", x);
}while (x < 10);
  return EXIT_SUCCESS;
}
```

Output 5 6 7 8 9 10

Explanation

The do-while loop construct says that irrespective of the condition check the statement block is executed at least once. Thus, x = x + 1 = 5 is executed first any how before the condition (x< 10).

5. Write a program to read an integer number from keyboard and test if it is an even no. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
int main(void)
{
  int n;
printf("Enter an integer no.\n");
scanf("%d", &n);
  if (n%2 == 0)
printf("%d is an even no.\n", n);
  else
printf("%d is an odd no. \n", n);
getch();
  return EXIT_SUCCESS;
}
```

6. A shopkeeper, gives discount 20% if customer's purchased amount exceeds Rs. 2000 ; 15% for amount greater than 1000 but less then 2000 and 10% for amount greater than 500 and less than 1000. Write a program to read the gross amount and evaluate discount, if any, and to display the net amount payable. **[MODEL QUESTION]**

Answer:

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

```
int main(void)
{
  float gross, dis;
printf("Enter the gross amount\n");
scanf("%f", &gross);
  if (gross > 2000.00)
    dis = 0.20 * gross;
  else if (gross > 1000.00)
    dis = 0.15 * gross;
  else if (gross > 500.00)
  dis = 0.10 * gross;
  else
    dis = 0.0;
printf("Gross amount % .2f \n", gross);

printf("Discount % .2f\n", dis);
printf("Net amount payable % .2f \n", gross - dis);
getch();
  return EXIT_SUCCESS;
}
```

7. Write a program to find the maximum and minimum nos. among some real numbers (how many numbers that will not be given in advance). But the condition is that all the numbers will be given in a single line in the input.

[MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
int main(void)
{
  float x;
  char c = ' ';
  float min = 1.0E38;
  float max = - 1.0E38;
printf("Enter the nos. all in a single line \n");
  while (c != '\n')
  {
scanf("%f%c", &x, &c);
    if (x > max)
      max = x;
    if (x < min)
      min = x;
  }
printf("The max. no. is % f \n", max);
printf("The min. no. is % f \n", min);
getch();
  return EXIT_SUCCESS;
}
```

8. Write a program to test if a number entered from the keyboard is prime.
[MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
int main(void)
{
  int n, test, prime = 1;
printf("Enter the no.\n ");
scanf(" %d", &n);
  for (test = 2; test <= sqrt(n); test++)
  {
    if (n%test == 0)
    {
      prime = 0;
      break;
    }
  }
  if (prime)
printf("%d is a prime no. \n", n);
  else
printf(" %d is not a prime no.\n", n);
getch();
  return EXIT_SUCCESS;
}
```

9. Write a program to find out the roots (real/ imaginary) of quadratic equation of the form $ax^2 + bx + c = 0$; take the values of a, b and c as input.
[MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include<stdlib.h>
int main(void)
{
  float a, b, c, d;
  float x, y;
printf("Give the values of a, b, and c\n");
scanf("%f %f %f", &a, &b, &c);
  d = b * b - 4 * a * c;
  if (d >= 0)
  {
    x = (-b + sqrt(d)) / (2 *a);
    y = ( - b - sqrt(d)) / (2 *a);
printf("The roots are real and are %f  &  %f", x, y);
```

```
    }
    else
    {
      x = sqrt((- 1))*d;
      y = 0.5* a;
  printf("\nThe roots are imaginary and are\n ");
  printf("% f(-%f + i %f), %f ( -%f - i%f)", y, b, x, y, b, x);
    }
  getch();
    return EXIT_SUCCESS;
  }
```

10. Square of 12 is 144. 21, which is reverse of 12 has a square 441, which is same as 144 reversed. There are few nos. which have this property. Write a program to find out whether any more such nos. exists in the range 10 to 100.

[MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
int main(void)
{
int m, n, i, j, x, y, z, temp ;
      for (m = 10 ; m < 100 ; m ++)
      {
            x = m*m ;
            temp = x;
            y = 0 ;
            while ( x!= 0)
            {
                  i = x % 10;
                  x = x /10;
                  y = y * 10 + i;
            }
            i = m % 10;
            j = m / 10;
            n = i * 10 + j ;
            z = n * n ;
            if (y == z)
            {
                  printf( "Number =%d Square = %2d ||", m, temp) ;
                  printf (" Reverse Number =%d Square = %d", n, z) ;
                  printf("\n");
            }
  }
getch();
  return EXIT_SUCCESS;
}
```

```
/*Output:
Number =10 Square = 100 || Reverse Number =1 Square = 1
Number =11 Square = 121 || Reverse Number =11 Square = 121
Number =12 Square = 144 || Reverse Number =21 Square = 441
Number =13 Square = 169 || Reverse Number =31 Square = 961
Number =20 Square = 400 || Reverse Number =2 Square = 4
Number =21 Square = 441 || Reverse Number =12 Square = 144
Number =22 Square = 484 || Reverse Number =22 Square = 484
Number =30 Square = 900 || Reverse Number =3 Square = 9
Number =31 Square = 961 || Reverse Number =13 Square = 169
*/
```

11. Write a program to check whether a given number is a palindrome or not. For example, the no. 12321 is a palindrome. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
int main(void)
{
  long int x, y, n;
  int i;
printf("Enter a number\n");
scanf("%ld", &x);
y = 0;
n = x;
while (x != 0)
  {
i = x % 10;
    x = x / 10;
    y = y * 10+i;
  }
  if (n == y)
  {
printf("The number% ld is a palindrome", n);
  }
  else
  {
printf("The number% ld is not a palindrome", n);
  }
getch();
  return EXIT_SUCCESS;
}
```

ARRAYS

☞ Chapter at a Glance

Fundamentals of Arrays

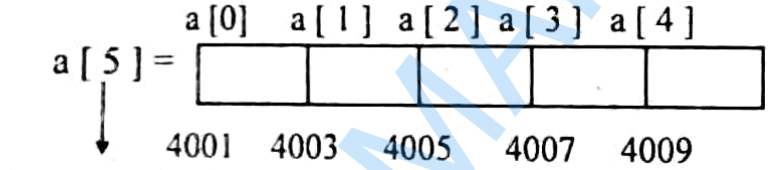
An array is a finite set of ordered, contiguous homogeneous (of same type) elements. It can also be called as a static linear data structure which means, memory allocated during initialization of an array cannot be increased or decreased during the runtime of the program. This initialization is done during compile time of a program. Intuitively, an array is a set of pairs, index and value. For each index, which is defined, there is a value associated with the index.

Single-Dimension Array

A single dimension/1D array can be represented in the following ways:

```
int arr [6] = {4,5,20, 99,14,6}; /* integer array*/
float arr [5] = {4.5, 6.7, 8.3, 9.1, 10.2}; /*float array*/
char arr [5] = {'A', 'B' 'C', 'D', 'E'}; /*character array*/
```

Representation of 1D array in memory



indicates the size of array as 5

4001 , 4003,.....4009 are contiguous memory addresses.

4001 = base address or the starting address of the array.

a [0], a [1], ... a [4] where 0, 1, 2,...4 are called the subscript or indices of an array. Each subscript represents the position of the elements stored in the array.

a [0] → Zeroth or first element

a [1] → Second element

a [4] → the last element.

The addresses are 4001, 4003, 4005, although does not look like contiguous memory locations, but in C they are. It's because as integer requires two bytes to store information in memory then for every element of the array a memory location of size 2 bytes is reserved. That means for five elements a total of 10 locations or 10 bytes are reserved.

Consider the following arrays & their memory representation.

```
float arr [] = {1.2, 1.3, 1.4, 1.5, 1.6};
char arr [] = {'a', 'b', 'c', 'd', 'e'};
```

float arr[] =

a[0]	a[1]	a[2]	a[3]	a[4]
1.2	1.3	1.4	1.5	1.6
4001	4005	4009	4013	4017

floating point variable requires 4 bytes each. So if the address of the first element is 4001, it will take 4 bytes & later on the next element will have the address i.e., 4005 in our case.

char arr[] =

a[0]	a[1]	a[2]	a[3]	a[4]
a	b	c	d	e
4001	4002	4003	4004	4005

Character variable require 1 byte in memory so the address scheme is 4001, 4002, 4002, 4003, ...4005.

Two-Dimension Array

In C, two-dimensional arrays of different data types are declared in the following manner

```
int arr1[ 10] [ 10];
float arr2[20] [20];
```

The first line defines arr1 as an integer array having 10 rows and 10 columns and hence 10 x 10 = 100 elements can be stored. The second line declares arr2 as a floating- point array having 20 rows & 20 columns and hence 20 x 20 = 400 elements can be stored.

Representation of 2-D array in memory

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

a is an array of integer having 3 rows & 4 columns, that means each row of 'a' contains 4 elements. The result of this initial assignment is

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

A point related to this type of declaration: If there are too few values within a pair of branches, the remaining elements of that row will be assigned **Zeros**. On the other hand, the number of values within each pair of braces cannot exceed the defined row size.

Let us consider the following declaration:

```
int a [3][4] = {{1, 2, 3}, {6, 7, 8}, {9, 10, 11}};
```

As per the dimensions of array, 'a' there should be '12' values present in initialization part, but only 9 values are available and hence these values are assigned to only first three elements in each row.

```
a[0][0] =1 ; a[0][1] =2  ; a[0][2] = 3 ;  a[0][3] = 0
a[1][0] = 6 ; a[1][1]= 7  ;  a[1][2]= 8 ; a[1][3] = 0
a[2][0] = 9 ; a[2][1] = 10 ; a [2][2] = 1; a [2][3] = 0
```

Another example:

```
int a [3] [4] = {1,2,4,5,8,9,11,12,13};
```

The result will be

```
a[0][0] = 1 ; a[0][1] = 2 ; a[0][2]= 4  ; a[0][3] = 5
a[1][0] = 8 ; a[1][1] = 9; a [1][ 2] = 11; a[1][3] = 12
a[2][0] = 13; a[2][1] = 0 ; a[2][2] = 0 ; a[2][3] = 0
```

Here, three array elements a [2] [1], a [2] [2] and a [2] [3] will again be assigned zeros.

Character Array and Strings

Let us consider the following structure and the C program. The array declared below consists of characters in contiguous memory locations and is thus treated as string.

char arrStr[] =

arrStr[0]	arrStr[1]	arrStr[2]	arrStr[3]	arrStr[4]	arrStr[5]	arrStr[6]
M	A	T	R	I	X	\0'
4001	4002	4003	4004	4005	4006	4007

NULL ('\0') character indicating the end of the string

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(void)
{
  char arrStr [] = " MATRIX EDUCARE ";
  int i = 0;
  while (arrStr [i] != '\0') /* checks for the end of string */
  {
    printf("%c", arrStr[i]);
    i++;
  }
  getch();
  return EXIT_SUCCESS;
}
```

Explanation: The null character' \0 ' is automatically inserted at the end of the character array (string). The programmer need not have to worry for that.

The set of characters to the right of = is termed as a *string literal*.

Since strings are characters arrays, pointers to strings are treated in a way similar to pointers to array of other data type. The name of the character array (string) points to the base address of the string (i.e. the first character of the string) .

The expression (<arrayname> + i) points to the i^{th} character in the string. The expressions (array_name) ,and array_name [0] are equivalent and points to the first character in the string.

Very Short Answer Type Questions

1. What is the string function used to reverse a string? **[WBUT 2023]**

Answer:

A given string can be reversed in the C language by using **strrev function.**

2. What header file is used for string operation? **[WBUT 2023]**

Answer:
String.h is a standard header file in the C language that contains functions for manipulating strings (arrays of characters).

3. An array elements are always stored in _____ memory locations. [MODEL QUESTION]

a) Sequential
b) Random
c) Sequential and Random
d) None of the above

Answer: (a)

4. In C Programming, If we need to store word "INDIA" then syntax is as below – [MODEL QUESTION]

```
char name[];
name = "INDIA"
```

a) char name[6] = {'I','N','D','I','A'};
b) char name[6] = {'I','N','D','I','A','\0'}
c) char name[6] = {"I","N","D","I","A"}
d) name = "INDIA"

Answer: (b)

5. What will be printed after execution of the following code? [MODEL QUESTION]

```
int main()
{
      int arr[10] = {1,2,3,4,5};
      printf("%d", arr[5]);
}
```

Garbage Value

a) 5 b) 6 c) 0 d) None of these

Answer: (c)

6. What is right way to initialize array? [MODEL QUESTION]

a) int num[6] = { 21, 41, 2, 15, 4, 5 };
b) int n[] = { 21, 41, 2, 15, 4, 5 };
c) int n{6} = { 21, 41, 2 };
d) int n(6) = { 21, 41, 2, 15, 4, 5 };

Answer: (a)

7. What will be the output of the following code? [MODEL QUESTION]

```
#include"stdio.h"
void main()
{
      int a[10];
      printf("%d %d", a[-1], a[12]);
}
```

a) 0 0
b) Garbage value 0
c) 0 Garbage Value
d) Garbage value Garbage Value
e) Code will not compile

Answer: (c)

8. Let x be an array. Which of the following operations are illegal? [MODEL QUESTION]

```
++x
x+1
x++
x*2
```

a) a and b b) a, b and c c) b and c
d) a, c and d e) c and d

Answer: (c)

9. What is the maximum number of dimensions an array in C may have? [MODEL QUESTION]

a) 2 b) 8 c) 20 d) 50
Theoretically no limit. The only practical limits are memory size and compilers

Answer: (d)

10. What will be the output of the program? [MODEL QUESTION]

```
#include "stdio.h"
int main()
{
    int arr[5] = {10};
    printf("%d", 0[arr]);

    return 0;
}
```

a) 1 b) 0 c) 10 d) 6

Answer: (c)

Short Answer Type Questions

1. Define Array. Each element of an array DATA [15] [20] requires 4 bytes of storage. Base address of the DATA is 3020. Locate the address of DATA [12][13] when the elements are stored in row-major and column-major order. [WBUT 2019]

Answer:

An array is a linear data structure that can hold elements of same data type.
Row major order can be calculated by the following formula
ADDR (A [i, j]) = Base_Address + W [M (i) + (j)]
Here,
Base_Address is the address of first element in the array.
W is the word size. It means number of bytes occupied by each element.
N is number of rows in array.
M is number of columns in array.
Suppose we want to calculate the address of element DATA[12,13].
It can be calculated as follow:

Here,
Base_Address = 2000, W= 4, M=15, N=20, i=12, j=13
Address of DATA[12,13] = 3020 + 4*[15*12 + 13] = 3792
Column major order can be calculated by the following formula
ADDR (A [i, j]) = Base_Address + W [N (j) + (i)]
Address of DATA[12,13] = 3020 + 4*[20*13 + 12] = 4108

2. What will be output of the code below? **[WBUT 2023]**

```
#include <stdio.h>
int main()
{
int a[5]={1, 2, 3, 4, 5};
int i;
for (i=0; i<5; i++)
if ((char)a[i]=='5')
printf("%d\n", a[i]);
else
printf("FAIL\n");
}
```

Answer: Error

3. What is the advantage of passing an array as a function argument rather than passing individual elements of the array one at a time? **[MODEL QUESTION]**

Answer:

When an array is passed to a function, the original array is passed as an argument. No temporary array is created in this case. So if the array elements are changed, suppose they are sorted then the original array also gets sorted. So even if the function to which this array was passed does not return any value i.e. the return type is void, the sorted array is obtained directly in the main function from where it is passed. This is a classic example of call by value.

4. What are the advantages and disadvantages of arrays? **[MODEL QUESTION]**

Answer:

Advantages:

1. It is used to represent data items of same type by using only single name.
2. It can be used to implement other data structures like stacks, queues, trees, graphs etc.
3. It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.
4. Iterating the arrays using their index is faster compared to any other methods like linked list etc.
5. It allows to store the elements in any dimensional array - supports multidimensional array.

Disadvantages:

1. It allows us to enter only fixed number of elements into it. We cannot alter the size of the array once array is declared. Hence if we need to insert more number of records than declared then it is not possible. We should know array size at the compile time itself.
2. Inserting and deleting the records from the array would be costly since we add / delete the elements from the array, we need to manage memory space too.

It does not verify the indexes while compiling the array. In case there is any indexes pointed which is more than the dimension specified, then we will get run time errors rather than identifying them at compile time.

5. With the help of a suitable program explain the various ways how strings are read and printed in C? [MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char str[50];
  printf("Enter the string \n ");
  scanf("%s",str); /*note there is no '&' before str in scanf means the base address
of the character array is implicit. The %s is a format specification for handling a string.
*/
  printf(" The  Entered String is  =%s ", str);
  return EXIT_SUCCESS;
}
```

Long Answer Type Questions

1. a) Write a program to print diagonal matrix from 5×5 matrix?
b) Write a program to concatenate two strings without using strcat() function in C.
c) Write a program to sort n elements from an array. [WBUT 2023]

Answer:

a)

```
#include <stdio.h>

#define MAXROW      10
#define MAXCOL      10

int main()
{
    int matrix[MAXROW][MAXCOL];
    int i,j,r,c;
    printf("Enter number of Rows :");
    scanf("%d",&r);
```

```
    printf("Enter number of Cols :");
    scanf("%d",&c);

    printf("\nEnter matrix elements :\n");
    for(i=0;i< r;i++)
    {
        for(j=0;j< c;j++)
        {
            printf("Enter element [%d,%d] : ",i+1,j+1);
            scanf("%d",&matrix[i][j]);
        }
    }

    /*check condition to print diagonals, matrix must be square
matrix*/
    if(r==c)
    {
            /*print diagonals*/
            for(i=0;i< c;i++)
            {
                for(j=0;j< r;j++)
                {

                    if(i==j)
                        printf("%d\t",matrix[j][i]);        /*print
elements*/
                    else
                        printf("\t");   /*print space*/
                }
                printf("\n");   /*after each row print new line*/
            }
    }
    else
    {
        printf("\nMatrix is not a Square Matrix.");
    }
    return 0;
}
```

b)

```
#include<stdio.h>

void main(void)
{
  char str1[25],str2[25];
  int i=0,j=0;
  printf("\nEnter First String:");
  gets(str1);
```

```
  printf("\nEnter Second String:");
  gets(str2);
  while(str1[i]!='\0')
  i++;
  while(str2[j]!='\0')
  {
    str1[i]=str2[j];
    j++;
    i++;
  }
  str1[i]='\0';
  printf("\nConcatenated String is %s",str1);
}
```

c)

```
#include <stdio.h>

void main()
{
    int arr1[100];
    int n, i, j, tmp;

    // Prompt user for input
    printf("\n\nSort elements of array in ascending order:\n");
    printf("-----------------------------------------\n");
    printf("Input the size of array : ");
    scanf("%d", &n);

    // Input elements for the array
    printf("Input %d elements in the array :\n", n);
    for (i = 0; i < n; i++)
    {
        printf("element - %d : ", i);
        scanf("%d", &arr1[i]);
    }

    // Sorting elements in ascending order using the Bubble Sort
algorithm
    for (i = 0; i < n; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (arr1[j] < arr1[i])
            {
                // Swap elements if they are in the wrong order
                tmp = arr1[i];
                arr1[i] = arr1[j];
```

```
                arr1[j] = tmp;
            }
        }
    }

    // Print sorted elements in ascending order
    printf("\nElements of array in sorted ascending order:\n");
    for (i = 0; i < n; i++)
    {
        printf("%d  ", arr1[i]);
    }
    printf("\n\n");
}
```

2. Suppose, X is a matrix with m rows and k cols. And Y is a matrix with k rows and n cols. Then only the product of X and Y is defined. Write a program to read the matrices X and Y from the keyboard and find Z = X * Y if the product is defined.

[MODEL QUESTION]

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void matmult(int,int,int,float[][10],float[][10]);
int main(void)
{
  int i, j, m1, n1, m2, n2, k;
  float x[10][10], y[10][10];
  printf("No. of rows & cols. of 1st matrix \n");
  scanf("%d %d", &m1, &n1);
  printf("No. of rows. & cols. of 2nd matrix \n");
  scanf("%d%d", &m2, &n2);
  if (n1 == m2)     /*rule:- no of cols of 1st matrix = no of rows of 2nd matrix
*/
    k = n1;
  else
  {
    printf("\nWe can multiply matrices when \n");
    printf("no of cols of 1st matrix = no of rows of 2nd matrix");
    getch();
    return EXIT_FAILURE;
  }
  printf(" Enter the elements for 1st matrix\n");
  for (i = 0; i < m1; i++)
  {
    for (j = 0; j < k; j++)
    {
```

```
      scanf("%f", &x[i][j]);
    }
  }
  printf("Enter the elements for 2nd matrix  \n");
  for (i = 0; i < k; i++)
  {
    for (j = 0; j < n2; j++)
    {
      scanf("%f", &y[i][j]);
    }
  }
  matmult(m1, k, n2, x, y);   /* function call */
  getch();
  return EXIT_SUCCESS;
}
/*function definition */
void matmult(int m1, int k, int n2, float x[][10], float y[][10])
{
  int u, v, i;
  float z[10][10];
  for (u = 0; u < m1; u++)
  {
    for (v = 0; v < n2; v++)
    {
      z[u][v] = 0.0;
      for (i = 0; i < k; i++)
      {
        z[u][v] += x[u][i] *y[i][v];
      }
    }
  }
  printf("\nMatrix Z = X * Y \n");
  for (u = 0; u < m1; u++)
  {
    for (v = 0; v < n2; v++)
    {
      printf("%9.3f", z[u][v]);
    }
    printf(" \n");
  }
}
```

3. Write a program to find the transpose of a matrix. **[MODEL QUESTION]**

Answer:

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

```
int main(void)
{
  int a[10][10], b[10][10], i, j, m, n;
  printf("Enter the order of the matrix \n");
  printf("No. of rows \n");
  scanf("%d", &m);
  printf("No. of columns \n");
  scanf("%d", &n);
  printf("Enter the matrix elements \n");
  for (i = 0; i < m; i++)
  {
    for (j = 0; j < n; j++)
    {
      scanf("%d", &a[i][j]);
      b[j][i] = a[i][j];
    }
  }
  printf("The original matrix was \n");
  for (i = 0; i < m; i++)
  {
    for (j = 0; j < n; j++)
    {
      printf("%3d", a[i][j]);
    }
    printf(" \n");
  }
  printf("The transpose of this matrix is  \n");
  for (i = 0; i < m; i++)
  {
    for (j = 0; j < n; j++)
    {
      printf("%3d", b[i][j]);
    }
    printf(" \n");
  }
  return EXIT_SUCCESS;
}
```

4. Write a program to find the determinant of a matrix consisting of positive numbers. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  float a[10][10], c, det = 1;
  int i, j, k, m, n, flag = 0;
```

```
  printf(" How many cols? \n ");
  scanf(" %d", &n);
  printf(" Enter the elements \n ");
  for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
    {
      scanf(" %f", &a[i][j]);
    }
  }
  for (i = 0; i < n; i++)
  {
    if (a[i][i] == 0)
    {
      flag = 1;
      break;
    }
    for (j = 0; j < n; j++)
    {
      if (i != j)
      {
        c = a[j][i] / a[i][i];
        for (k = 0; k < n; k++)
        {
          a[j][k] = a[j][k] - a[i][k] *c;
        }
      }
    }
  }
  if (flag)
    printf("the determinant of the matrix is 0");
  else
    printf(" The determinant is ");
  for (i = 0; i < n; i++)
  {
    det *= a[i][i];
  }
  printf("%3f", det);
  return EXIT_SUCCESS;
}
```

5. Write a program to merge two sorted arrays in third array, which is again sorted.
[MODEL QUESTION]

Answer:

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

```
int main(void)
{
  int p, q, m, n, c;
  int a1[50], a2[50], a3[50];
  printf("How many elements for the 1st sorted array\n");
  scanf("%d", &p);
  printf("Enter the elements of the 1st sorted array?\n");
  for (m = 0; m < p; m++)
  {
    scanf("%d",&a1[m]);
  }
  printf("How many elements for the 2nd sorted array?\n");
  scanf("%d", &q);
  printf("Enter the elements of the 2nd sorted array\n");
  for (n = 0; n < q; n++)
  {
    scanf("%d", &a2[n]);
  }
  c = 0,m = 0, n = 0;
  while ((m < p) && (n < q))
  {
    if (a1[m] <= a2[n])
      a3[c] = a1[m++];
    else
      a3[c] = a2[n++];
    c++;
  }
  while (m < p)
  {
    a3[c] = a1[m];
    m++;
    c++;
  }
  while (n < q)
  {
    a3[c] = a2[n];
    n++;
    c++;
  }
  printf("The merged array in ascending order \n ");
  for (m = 0; m < c; m++)
  {
    printf("%2d", a3[m]);
  }
  getch();
  return EXIT_SUCCESS;
}
```

6. Write a program to input a matrix, determine if it is symmetrical matrix (a matrix is said to be symmetrical when (A=A'). **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  int x[10][10], n, i, j, flag;
  printf("How many cols?\n");
  scanf("%d", &n);
  printf(" Enter the elements \n");
  for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
    {
      scanf("%d", &x[i][j]);
    }
  }
  flag = 1;
  for (i = 0; i < n; i++)
  {
    for (j = i + 1; j < n; j++)
    {
      if ((x[i][j] - x[j][i]) != 0)
      {
        flag = 0;
        break;
      }
    }
  }
  if (flag)
    printf(" Symmetric matrix ");
  else
    printf(" Not a symmetric matrix");
  return EXIT_SUCCESS;
}
/* output
How many cols?
3
 Enter the elements
1 0 0
0 1 0
0 0 1
 Symmetric matrix
*/
```

7. Write a program to read a line of text from keyboard and to convert it into a coded text by changing its character depending upon a code number. This code number must be taken as input. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  char x[80], y[80];
  int code, len, i;
  printf("Give a line of text\n");
  gets(x); /*instead of scanf, we can use gets to read strings */
  printf("Enter the code No. \n");
  scanf("%d", &code);
  for (i = 0; i < strlen(x); i++)
    y[i] = x[i] + code;
  printf("The coded text is given below \n");
  puts(y); /*instead of printf, we can use puts to display strings */
  return EXIT_SUCCESS;
}
```

8. Write a C program that will use the following library functions in C i) strlen() ii) strcpy() iii) strcat(). **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  char str1[40], str2[40], ch;
  int choice, choice1, n;
  printf("Enter the 1st string:");
  gets(str1);
  printf("\n Enter the 2nd string:");
  gets(str2);
  printf("1. :Length of the strings\n");
  printf("2. :Concatenate two strings\n");
  printf("3. :Copy string2 to string1\n");
  printf("Enter your choice \n");
  scanf("%d", &choice);
  fflush(stdin);
  switch (choice)
  {
    case 1:
```

```
      printf("The length of the 1st string is %d\n",
strlen(str1));
      printf("The length of the 2nd string is %d\n",
strlen(str2));
      break;
    case 2:
      printf("Do you want to append the entire string(y/n)?");
      scanf("%c", &ch);
      if (ch == 'y' || ch == 'Y')
      {
        strcat(str1, str2); /* appends entire string */
      }
      else
      {
        printf("How many characters do you want to append
        \n");
        scanf("%d", &n);
        strncat(str1, str2, n); /* appends n characters to another string */
      }
      printf("After appending now the string is");
      puts(str1);
      break;
    case 3:
      strcpy(str1, str2);
      printf("string1 has been change to = ");
      puts(str1);
  }
  return EXIT_SUCCESS;
}
```

9. Write a C program that takes a string as input and find out the frequency of each character in the string. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  char x[40];
  int i, j;
  static int count[256];
  printf("Enter the string:");
  gets(x);
  for (i = 0; i <= strlen(x); i++)
  {
    for (j = 0; j < 256; j++)
```

```
    {
      if (x[i] == j)
      {
        count[j]++;
        break;
      }
    }
  }
  printf("\nThe frequency of the different characters are given
below\n");
  for (i = 0; i < 256; i++)
  {
    if (count[i] != 0)
      printf(" \nThe frequency of the character %c is
      %d", i, count[i]);
  }
  getch();
  return EXIT_SUCCESS;
}
```

10. Write a program to count the number of vowels and digits in a given string. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(void)
{
  char str[50];
  int i, x = 0, y = 0;
  printf("Enter a string\n ");
  gets(str);
  i = 0;
  while (str[i] != '\0')
  {
    if((str[i]=='a')||(str[i] == 'e') ||(str[i] == 'i')
    || (str[i] == 'o')|| (str[i] == 'u'))
      x++;
    if ((str[i] >= '0') && (str[i] <= '9'))
      y++;
    i++;
  }
  printf("The number of vowels are % d \n", x);
  printf("The number of digits are % d \n", y);
  getch();
  return EXIT_SUCCESS;
}
```

BASIC ALGORITHMS

☞ Chapter at a Glance

Searching

Searching is the term defined to retrieve a record with a particular value.
The most popular types of searching techniques are

- Linear (Sequential) Search
- Binary Search

Sorting

Sorting can be defined as a mechanism (operation) performed to arrange a given set of elements (e.g. an array of n elements) in a particular ordered fashion like in ascending or in descending order.
Sorting can be broadly classified into two categories.

- Internal sorting
- External sorting

Very Short Answer Type Questions

1. What is the time complexity of bubble sort? **[WBUT 2023]**

Answer:

The bubble sort algorithm's average/worst time complexity is $O(n^2)$.

2. What is linear searching in an array? **[WBUT 2023]**

Answer:

Linear search, the simplest search algorithm, is mainly used to find the element from an unordered list.

3. Where is linear searching used? **[MODEL QUESTION]**

a) When the list has only a few elements
b) When performing a single search in an unordered list
c) Used all the time
d) When the list has only a few elements and When performing a single search in an unordered list

Answer: (d)

It is preferred to use binary search in case of large list of items to reduce the time for searching.

4. What is the best case for linear search? **[MODEL QUESTION]**

a) O(nlogn) **b) O(logn)** **c) O(n)** **d) O(1)**

Answer: (d)

Best case is when the desired element is at the beginning of the array.

5. What is the worst case for linear search? [MODEL QUESTION]
a) O(nlogn) b) O(logn) c) O(n) d) O(1)

Answer: (c)
Worst case is when the desired element is at the end of the array or not present at all, in this case you have to traverse till the end of the array. hence the complexity is O(n).

6. What are the applications of binary search? [MODEL QUESTION]
a) To find the lower/upper bound in an ordered sequence
b) Union of intervals
c) Debugging
d) All of the above

Answer: (d)

7. Binary Search can be categorized into which of the following? [MODEL QUESTION]
a) Brute Force technique
b) Divide and conquer
c) Greedy algorithm
d) Dynamic programming

Answer: (b)

8. Given an array arr = {15,16,67,84,110} and key = 84. How many iterations are done until the element is found? [MODEL QUESTION]
a) 1 b) 3 c) 4 d) 2

Answer: (d)
Iteration1 : mid = 67; Iteration2 : mid = 84;

9. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance? [MODEL QUESTION]
a) Insertion sort
b) Selection sort
c) Quick sort
d) Merge sort

Answer: (a)

10. Consider the situation in which assignment operation is very costly. Which of the following sorting algorithm should be performed so that the number of assignment operations is minimized in general? [MODEL QUESTION]
a) Insertion sort
b) Selection sort
c) Heap sort
d) None

Answer: (b)

11. A sorting technique is called stable if it [MODEL QUESTION]
a) Takes O(nlogn) times
b) Maintains the relative order of occurrence of non-distinct elements
c) Uses divide-and-conquer paradigm
d) Takes O(n) space

Answer: (b)

Short Answer Type Questions

1. Define linear search and binary search. **[MODEL QUESTION]**

Answer:

Linear or Sequential Search is a method where the search begins at one end of the list, scans the elements of the list from left to right (if the search begins from left) until the desired record is found.

In **Binary Search** the entire sorted list is divided into two parts. We first compare our input item with the mid element of the list and then restrict our attention to only the first or second half of the list depending on whether the input item comes left or right of the mid-element. In this way we reduce the length of the list to be searched by half.

2. What is internal and external sorting? **[MODEL QUESTION]**

Answer:

Sorting can be broadly classified into two categories.

- ***Internal sorting:*** It is the method when the sorting takes place within the main memory. The time required for read and write operations are considered to be insignificant e.g. Bubble Sort, Selection sort, Insertion sort etc.
- ***External sorting:*** It is the method when the sorting takes place with the secondary memory. The time required for read and write operations are considered to be significant e.g. sorting with disks, sorting with tapes.

3. Define Bubble, Insertion and Selection sort. **[MODEL QUESTION]**

Answer:

Bubble Sort

Given an array of unsorted elements, Bubble sort performs a sorting operation on the first two adjacent elements in the array, then between the second & third, then between third & fourth & so on.

Selection Sort

In selection sorting we select the first element and compare it with rest of the elements. The minimum value in each comparison is swapped in the first position of the array. During each pass elements with minimum value are placed in the first position, then second then third and so on. We continue this process until all the elements of the array are sorted.

Insertion Sort

In insertion sort data is sorted data set by identifying an element that is out of order relative to the elements around it. It removes that element from the list, shifting all other elements up one place.

Finally, it places the removed element in its correct location.

For example, when holding a hand of cards, players will often scan their cards from left to right, looking for the first card that is out of place. If the first three cards of a player's hand are 4, 5, 2, he will often be satisfied that the 4 and the 5 are in order relative to each other, but, upon getting to the 2, desires to place it before the 4 and the 5. In that case, the

player typically removes the 2 from the list, shifts the 4 and the 5 one spot to the right, and then places the 2 into the first slot on the left.

Long Answer Type Questions

1. Explain Linear search with a suitable function. What is the time complexity of this function? **[MODEL QUESTION]**

Answer:

Let us consider an array with the following elements:

```
a [ ] =   33,  51,  27,  85,  66,  23,  13,  57
```

Suppose we want to search for the element 66. We scan the array from left starting from the first element comparing 66 with all other elements in the array. If a suitable match is not found we move on the next element for comparison. In our case the suitable match is found at a [4] position.

The C function, which accomplishes the above task, is as follows, where int flag represents two values 1 and 0 respectively. "1" is for successful search and "0" is for unsuccessful search.

```
int linear_search(int a[], int n, int item)
{
  int position = 0,i;
  flag = 0; // element not found
  for (i = 0; i < n; i++)
  {
    if (a[i] == item)
    {
      flag = 1; // element found
      position = i;
      break;
    }
  }
  return flag;
}
```

Time complexity of Linear Search:

The "for loop" iterates n times hence the time complexity is O (n) in worst case.

2. Explain Binary search with a suitable function. What is the time complexity of this function? **[MODEL QUESTION]**

Answer:

In Binary Search, the entire ***sorted*** list is divided into two parts. We first compare our input item with the mid element of the list and then restrict our attention to only the first or second half of the list depending on whether the input item comes before or after the mid-element. In this way we reduce the length of the list at each step until we are getting a single element in a list.

Let us consider a sorted array with the following.

```
a [ ] =   13  23  27  33  51  66  85
```

We find the mid-element by the formula

`mid = a [ left + right ] / 2;` left and right are the leftmost and rightmost index of the array. In our example left = 0 and right = 6.

Therefore mid = a [0 + 6] / 2 = a[3] = 33.Thus the array now looks like

Left array = 13 23 27 **33** **Right array** = 51 66 85

Let us consider our input search item is 23.

Step 1:

We compare 23 with our mid element i.e. 33. We find 23 < 33. Hence we move to the left of 33. The left side of 33 contains all the elements lesser than 33. In this way we restrict our search to one side of an array.

Step 2:

We again compute the mid of left array by the formula a [left + right] / 2 = a [0 + 2] / 2 = 1. We compare 23 with our new mid element i.e. 23. We found the match. Hence the program ends here. If the match were not found we would have continued with step 2. The C function, which accomplishes the above task, is as follows.

```
int bin_search(int A[], int n, int item)
{
  int left = 0,
  right = n - 1;
  int flag = 0; //a flag to indicate whether the element is found
  int mid = 0;
  while (left <= right) // till left and right cross each other
  {
    mid = (left + right)/2; // divide the array into two halves
    if (item == A[mid])
    {
      flag = 1; //set flag = 1if the element is found
      break;
    }

    else if (item < A[mid])
      right = mid - 1; /* compute new right from the right sub array */
    else
      left = mid + 1; /* compute new left from the left sub array */
  }
  return flag;
}
```

The time complexity of Binary Search is as follows:
In each iteration, the array is split into two halves. Thereby we can say that the binary search takes the form of a binary tree. The time complexity is thus O ($\log_2 n$) in worst case also.

3. Write a C-function for Bubble Sorting? Explain the time complexity. [MODEL QUESTION]

Answer:
We call a function bubble_sort(a , n) with the arguments an entire array of elements and a variable for the number of elements .We define the function body as given below.

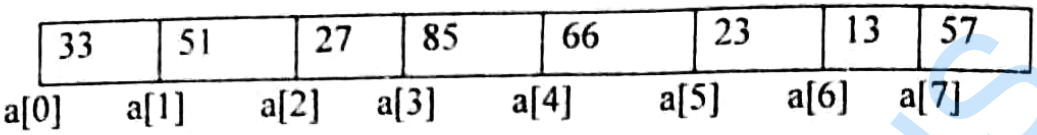
```
void bubble_sort(int a[], int n) // n is the no. of elements

{
  int temp, i, j;
  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('\n The sorted elements are\n");
  for (i = 0; i < n; i ++ )
    printf("%d", a[i]);
} //end of function bubble-sort
```

Time Complexity* of *Bubble sort
For the first pass the inner loop iterates n - 1 times. In the next iteration n - 2 times and in the last pass only once.
Hence the complexity of the bubble sort is
$(n - 1) + (n - 2) + (n - 3) + \ldots\ldots\ldots\ldots\ldots + 2 + 1$
$= [(n-1) * (n-1+1)]/2$
$= [n(n-1)]/2$
$= O(n^2)$
This expression is true for average case as well as worst case.

4. What is selection sorting? Explain with a suitable example. [MODEL QUESTION]
Answer:
In selection sorting we select the first element and compare it with rest of the elements. The minimum value in each comparison is swapped in the first position of the array. During each pass elements with minimum value are placed in the first position, then

second. then third and so on. We continue this process until all the elements of the array are sorted. Let us consider an array of the following unsorted element.



Step 1:
We start with selection the first element of the array here it is 33. We assume it to be minimum. The minimum value is at location zero of the array.

Step 2:
At the first iteration we scan all the elements of the array to find the least element. Here we find that element to be a [6]=13. And we find the least element at the location no 6 of the array.

Step 3:
We swap between a[0] & a[6]. Thus a[0] now is =13 & a[6] is=33. Thereby we ensure that, during the first iteration of selection sorting, the least element is placed at the first position of the array.

So,
Iteration 1: 13 51 27 85 66 23 33 57
Iteration 2: Now 51 is selected to be the least. We scan the array for least element occurring after 51. We find 23 to be that least element so we swap 23 & 51. The position of the rest of the elements remain unchanged

13 23 27 85 66 51 33 57

Iteration 3: Now 27 is selected to be the least. We scan the array for the least element occurring after 27. Here there is no such element. So there will be no swapping.
Iteration 4: Now 85 is selected to be the least. We scan the array for the least element occurring after 85. Here it is 33 so we swap between 85 &33

13 23 27 33 66 51 85 57

We follow the above method and the following iterations are shown below
Iteration 5: 13 23 27 33 51 66 85 57
Iteration 6: 13 23 27 33 51 57 85 66
Iteration 7: 13 23 27 33 51 57 66 85.
Thus at the final iteration we get the sorted array.
23 27 33 51 57 66 85.

5. Write a C-function for Selection Sorting? Explain the time complexity.
[MODEL QUESTION]

Answer:
We call a function selecsort(a , n) with the arguments an entire array of elements and a variable to store number of elements. We define the function body as given below.

```
void selecsort(int a[], int n)        //n denotes the no. of elements
{
  int loc, min, temp, i, j;
```

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

    if (loc ! = i)
    {
      temp = a[i];
      a[i] = a[loc];
      a[loc] = temp;
    }
  }
  printf("\nThe sorted array is:\n");
  for (i = 0; i < n; i++)
    printf("%3d", a[i]);
}
```

Time Complexity of Selection Sort

For first pass the inner for loop iterates n - 1 times. For second pass the inner for loop iterates n - 2 times and so on. Hence the time complexity of selection sort is

$(n - 1) + (n - 2) + (n - 3) + \ldots\ldots\ldots\ldots\ldots + 2 + 1$

$= [(n - 1) * (n - 1 + 1)]/2$

$= [n(n-1)]/2$

$= O(n^2)$

The selection sort minimizes the number of swaps. Swapping data items is time consuming compared with comparing them. This sorting technique might be useful when the amount of data is small.

6. Write a C-function for Insertion Sorting? Explain the time complexity. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void fib(int);
int main(void)
{
  int n;
  printf("How many numbers in the fibo sequence?\n");
```

```
  scanf("%d", &n);
  printf("The fibonacci sequence is\n");
  fib(n);
  getch();
  return EXIT_SUCCESS;
}
void fib(int m)
{
  int nt, pt, ct, count;
  ct = 1;
  pt = 0;
  printf("%5d%5d", pt, ct);
  count = 2;
  while (count < m)
  {
    nt = ct + pt;
    printf("%5d", nt);
    pt = ct;
    ct = nt;
    count++;
  }
}
```

Time complexity of Insertion sort

If the above array a[ ] is in order, then only one comparison will be made in the inner loop. Therefore, total number of comparisons made will be decided by the outer loop, which are n - 1. Hence the time complexity of insertion sort for best case is O (n).

In the worst case inner while loop iterates once for the first pass, in the second pass 2 times and so on. For n passes the outer for loop iterates n times. Hence the time complexity of insertion sort in the worst case is

$1 + 2 + 3 + \ldots\ldots\ldots\ldots\ldots\ldots + (n-1) + n$

$= (n * (n - 1)) / 2$

$= O (n^2)$

FUNCTION

Chapter at a Glance

Function

A function is a self-contained body that consists of several lines of execution to achieve a certain behavior of a program. The idea behind a function is that it *compartmentalizes* part of the program and in particular, that the code within the function has some useful properties

Some of the significant reasons for which a modular approach is preferable are as follows:

- It often happens that some lines of a program are required at multiple places of the program. Instead of repeating these common lines, it is better to move them into a separate function. Then this function can be called from multiple places wherever it is required. This approach is commonly known as reusability.
- Instead of having all the logic in a main function body, it is preferred to break them into small functions and call them at appropriate places. This makes the program more readable. This approach is commonly known as readability.
- A good function body typically should have less number of arguments and few lines of execution. It is advisable to keep the number of arguments restricted to three or four and the number of lines of execution to fifty or sixty lines. But these numbers will definitely vary in accordance with the programming needs and complexity.

Skeleton of a C function

```
<return type> <function name> (<argument list>)
{
      function body;
      return (expression);
}
```

return type: This field is mandatory. It signifies whether a function should return a value or it should return nothing. A function can return an integer value, float value, double value, or nothing. When a function does not return anything, the return type is written as ***void***. A function can return only one value at a time

function-name: This field is mandatory. In a C program there may be more than one function. In that case each function name should be different otherwise the compiler will throw an error.

argument-list: This is optional. It contains valid variable names separated by commas. It provides means for data communication with other functions.

return (expression): This is mandatory for functions which returns a value. The return statement is the mechanism for returning a value. This is also an optional statement for a function, which does not return any value, and hence the return type is said to be ***void***.

Scope and Visibility of function

An identifier's "visibility" determines the portions of the program in which it can be referenced — its "scope." An identifier is visible (i.e., can be used) only in portions of a program encompassed by its "scope." which may be limited (in the order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears. The scope of an identifier is the part of the program in which the name can be used. This is sometimes called "lexical scope." There are four kinds of scope: function, file, block, and function prototype.

All identifiers except labels have their scope determined by the level at which the declaration occurs. The following rules for each kind of scope govern the visibility of identifiers within a program:

File scope

The declaratory or type specifier for an identifier with file scope appears outside any block or list of parameters and is accessible from any place in the program unit after its declaration. Identifier names with file scope are often called "global" or "external." The scope of a global identifier begins at the point of its definition or declaration and terminates at the end of the program unit.

Function scope

A label is the only kind of identifier that has function scope. A label is declared implicitly by its use in a statement. Label names must be unique within a function.

For example, the go to statements.

Block scope

The declaratory or type specifier for an identifier with block scope appears inside a block or within the list of formal parameter declarations in a function definition. It is visible only from the point of its declaration or definition to the end of the block containing its declaration or definition. Its scope is limited to that block and to any blocks nested in that block and ends at the curly brace that closes the associated block. Such identifiers are sometimes called "local variables."

Parameter passing in functions

The different types of parameter passing techniques in C language are:

- Call by Value
- Call by Reference (Call by Address)

Call by Value: In this type of parameter passing the caller function passes a temporary copy of the parameter rather the original one to the called function. The called function works on this temporary copy and modifies it. This modification only reflects in the called function body and does not change the original value of the variable in the caller function.

Call by Reference: In this type of parameter passing the caller function only allows to change the internal value of the parameter in the called function. This change will be available in the caller function also. That is the original value changes. No temporary local copy of the parameter is created in this case. This is achieved when a reference i.e. an address is passed as argument.

Very Short Answer Type Questions

1. A function may contain [WBUT 2019]

a) one return statement
b) two return statements
c) more than two return statements
d) none of these

Answer: (a)

2. What is pre-processor? [WBUT 2023]

Answer:
A software program that processes the source file before sending it to actual compilation is called preprocessor.

3. Recursion uses which data structure? [WBUT 2023]

Answer:
We conclude that stack data structure is used to implement recursion.

4. What is Macro? [WBUT 2023]

Answer:
Macros in C are powerful tools that allow developers to define reusable code snippets.

5. What is the keyword used to transfer control from a function back to the calling function [MODEL QUESTION]

a) switch
b) go to
c) return
d) move

Answer: (c)

6. Which of the following is a correct format for declaration of function? [MODEL QUESTION]

a) return-type function-name(argument type);
b) return-type function-name(argument type) {}
c) return-type (argument type)function-name;
d) Both (a) and (b)

Answer: (a)

7. Recursion is similar to which of the following? [MODEL QUESTION]

a) int 1bhk(int);
b) int 1bhk(int a);
c) int 2bhk(int*, int []);
d) All of the mentioned;

Answer: (d)

8. What is the return-type of the function sqrt()? [MODEL QUESTION]

a) int
b) float
c) double
d) Depends on the data type of the return parameter

Answer: (d)

Short Answer Type Questions

1. Explain the salient features of a function. [MODEL QUESTION]

Answer:
i) Function name should follow the rules of a variable name.
ii) Function may or may not return any value.
iii) Function can return only one value at a time. To return any value from function return statement should be used.

2. What are the important parts of designing recursive algorithms? **[MODEL QUESTION]**

Answer:
The important parts of designing recursive algorithms are:

- Find the key step
- Find a stopping rule
- Outline the algorithm
- Check the termination
- Draw the recursion tree.

3. Clearly differentiate between function prototype, function definition and function call. **[MODEL QUESTION]**

Answer:
A **function definition** consists of the name of function, a list of arguments, the return type, declaration of local variables along with the body of the function that constitutes the set of instruction to perform a specific task. The first line of function definition that constitutes the name and the list of arguments along with the return type is called **Function declaration.**

Function prototype is the same as the function declaration, only that it ends with a semicolon (;). It is a single statement and expects input parameters. It is generally used to inform the compiler regarding the number and the nature of arguments. If any mismatch occurs in argument type in the function declaration, its prototype and the function call, a compilation error is reported.

Function call on the other hand is a statement to invoke the function in some other function body. It constitutes the name of the function, along with the list of arguments (values that are transferred to the function body). It also ends with a semicolon (;). To illustrate all the above, let us consider the following example.

4. What is the difference between formal and actual arguments in a function? **[MODEL QUESTION]**

Answer:
The major difference between actual and formal arguments is that actual arguments are the source of information; calling programs pass actual arguments to called functions. The called functions access the information using corresponding formal arguments.

5. How to convert a normal recursive function to tail recursive? **[MODEL QUESTION]**

Answer:
Let us consider the following Fibonacci function

```
int fib_recur ( int n , int next, int result)
{
if (n == 0)
      return result;
return fib_recur(n - 1, next + result, next);
}
```

The corresponding iterative version is as follows

```
int fact_iter(int n, int result) {
int temp_n;
int temp_result;
while (n != 1) {
temp_n = n;
temp_result = result;
n = temp_n - 1;
result = temp_n * temp_result;
}
return result;
}
```

6. State the advantages and disadvantages of recursion. **[MODEL QUESTION]**

Answer:
The advantages and disadvantages of recursion are:

Advantages

i) The advantages of recursion are that it reduces program code, thus making it simpler.
ii) It works well for computations, which are larger in depth

Disadvantages

i) If the stopping rule is not correctly implemented, then the function may enter into infinite loop giving us an erroneous output.
ii) The performance of recursion degrades for computations, which expands recursively to large values.

Long Answer Type Questions

1. a) Explain the role of C pre-processor. What is macro and how is it difference from a C function? **[WBUT 2019]**

Answer:
C pre-processor directives help us to modify the program text before it is compiled. Basically, C pre-processors directives have the capability of simple text substitution tool, so that you can modify parts of program before it's actually compiled.
In common a preprocessor directive performs the following activities.

- Macro definition
- Conditional compilation
- Inclusion of named files

b) Write a C function to find the length of a string and call the function from the main() function. Do not use 'strlen' function in your program. **[WBUT 2019]**

Answer:

```
#include <stdio.h>
int main (void)
{
      char str[50];
      printf("Enter the string :- \n");
      gets(str);
      printf("Length of string = %d", strlen1(str));
}
int strlen1(char ch[])
{
      int i = 0;
      while (ch[i] != '\0')
      {
          i++;
      }
      return i;
}
```

2. Write short note on Function Prototype. **[WBUT 2019]**

Answer:

A function prototype declares the return type and argument list of a function. For example, float sum (float , float, float), is a function prototype and it says that the function sum returns a floating value by adding three real numbers which it takes as float arguments. The general syntax of a function prototype is:

return type function name (arg_1, arg_2, ……..arg_n).

3. Explain in detail C storage class specifiers- auto, register, static, extern with suitable examples. **[MODEL QUESTION]**

Answer:

Storage class specifiers are used for the following purposes:

1. Where to store the value of variable: registers of memory
2. Specify the scope of the variable
3. Specify life time of variable

Auto storage class

A variable declared as 'auto' qualifier implies that the variable exists/accessible only with in the code block they are declared. For all the code blocks outside the declared code block, it's as if the variable doesn't exist. If some code outside of the code block tries to access an 'auto' variable, it will cause compilation error. By default, all variables are of auto storage class.

Example

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

```
int main( )
{ //Code block 1
auto int a = 1;
{ //Code block 2
auto int a = 2;
{ //Code Block 3
auto int a = 3;
printf ( "\n Value of 'a' in Code Block 3: %d", a);
}
printf ( "\n Value of 'a' in Code Block 2: %d", a);
}
printf ( "\n Value of 'a' in Code Block 1: %d", a);
}
```

Output:

Value of 'a' in Code Block 3: 3
Value of 'a' in Code Block 2: 2
Value of 'a' in Code Block 1: 1

Register storage class

register' storage class is same as 'auto' except that 'register' variables are stored in CPU registers compared to 'auto' where 'auto' variables are stored in RAM. The scope of 'register' class variable is same as 'auto' class variable, i.e. they can't be accessed outside the code block they are declared. 'register' variables will also be initialized with garbage value.

Since CPU register access is faster than memory access, declaring frequently accessed variables as 'register' will improve the performance of program. But one important thing to note about 'register' variable is, it's not guaranteed that those variables will be stored in CPU registers because number of registers is very limited for a CPU. So, if all the registers are allocated for other variables or purposes, then a variable will get stored in RAM even if it's declared as 'register' type.

Also, you cannot request address of a register variable using address operator (&). This will result in compilation error because 'register' variables might not be stored on RAM.

Example

```
#include <stdio.h>
int main( )
{ //Code block 1

register int a = 1;
{ //Code block 2
register int a = 2;
{ //Code Block 3
register int a = 3;
printf ( "\n Value of 'a' in Code Block 3: %d", a);
}
printf ( "\n Value of 'a' in Code Block 2: %d", a);
}
```

```
printf ( "\n Value of 'a' in Code Block 1: %d", a);
}
```

Output

Value of 'a' in Code Block 3: 3
Value of 'a' in Code Block 2: 2
Value of 'a' in Code Block 1: 1

Static storage class

Value of a 'Static' variable will be maintained across the function invocations. If a variable is declared using 'Static' qualifier at the top of the source file, then the scope of that variable is within the entire file. Files outside the declared file can't access the declared static variable.

'Static' variables declared within a function or a block, also known as local static variables, have the scope that, they are visible only within the block or function like local variables. The values assigned by the functions into static local variables during the first call of the function will persist/available until the function is invoked again.

'Static' variables are by default initialized with value 0.

Example

```
#include <stdio.h>
static int global_static_variable = 1;

static void static_variable_test() {
static int local_static_variable;
printf("local_static_variable:%d\t", local_static_variable);
local_static_variable++;
printf("global_static_variable:%d\n", global_static_variable);
global_static_variable++;
}
int main()
{
static_variable_test();
static_variable_test();
}
```

- 'Static' variable got initialized with value 0 by default. (We didn't initialize the variable 'local_static_variable' with a value, but it got initialized with 0 automatically)
- Value of the the local variable 'local_static_variable' is retained across invocations of function 'static_variable_test()'

Output

```
local_static_variable:0    global_static_variable:1
local_static_variable:1    global_static_variable:2
```

Static and Global functions/variables

In C programming language, functions are global by default. That means, by default function declared in a file can be accessed from code with in another file. But when you

apply static keyword to a functions, it makes the function accessible only from that file. So if you want to restrict access to a function from code in other files, you can declare the function as static.

In the example given above, the function 'static_variable_test' will be visible only with in the file it's declared. Also, if you declare a global variable as static, code in other files cannot access the variable. In simple words, declaring a global function or variable as static gives it internal linkage.

Extern storage class

extern' storage class defines a global variable visible /accessible to all files when linking. That means, 'extern' keyword informs the compiler that the variables declared using the keyword is already defined somewhere else in the program. Since the extern is defined at some other place in the program, trying to initialize an extern variable as part of declaration creates compilation error, because initialization makes the declaration statement a definition and it's in turn redefining a variable defined already. If the 'extern' variable is not defined else where in program, then you can initialize it as part of declaration (But compiler might give a warning message).

Example

```
#include <stdio.h>
// This is only a declaration
extern int extern_variable;
int main()
{
printf("extern_variable: %d", extern_variable);
return 0;
}

// Actual definition occurs here
int extern_variable = 20;
```

Output

```
extern_variable: 20
```

In the above program, 'extern_variable' is defined at the end of file, but since it's declared using extern keyword, compiler can properly link it.

'Extern' and Functions

We can use extern keyword along with function declaration as well, but that's a redundant operation since functions have external linkage by default. i.e. they can be accessed globally from other places.

4. Give a tabular representation of the various storage class specifiers in terms of space, default value, scope and life. [MODEL QUESTION]

Answer:

Storage class	Space	Initial value	Scope	Life
auto	Memory	Garbage	Local Scope	Alive within the

				declared code block
register	CPU Register	Garbage	Local Scope	Alive within the declared code block
static	Memory	0	Local Scope	Value retained between invocations
extern	Memory	0	Global Scope	Alive till program exit

5. What is a recursive function? Explain in brief with a suitable function. **[MODEL QUESTION]**

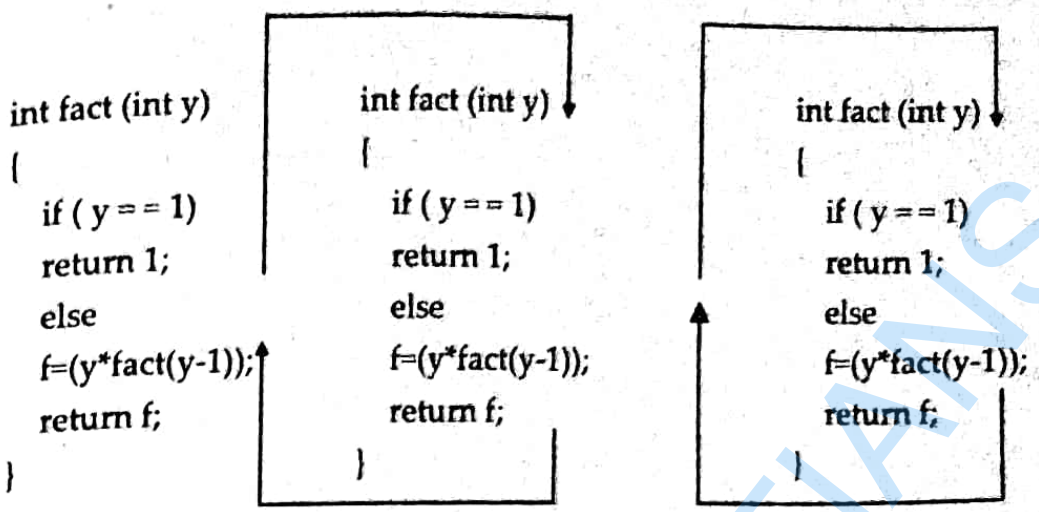
Answer:

When a function is called from the body of itself then the function is said to be as recursive function. Whenever function is called from itself then there is a chance that the function will enter into an infinite loop. To prevent a program from entering into an infinite loop in such a manner, every recursive function must have a terminating condition that decides whether to call the function once more or not. This terminating condition is also known as ***stopping rule*** in recursive function.

To illustrate the of recursive behaviour of a function, let us consider the following sample program.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int fact(int); /* function prototype */
int main(void)
{
int n, f; // local variables
printf("Enter any number:\n");
scanf("%d", &n);
f = fact(n);  /*function call*/
printf(" Factorial = %d\n", f);
getch();
return EXIT_SUCCESS;
}
/***************function body starts****************/
int fact(int y)
{
if (y == 0) /* stopping rule*/
return (1);
else
return (y *fact(y - 1));  /*recursion takes place here*/
}
```

*/***************function body ends****************/*



Assuming that 3 is entered. The first time the function fact (int y) takes y = 3. Since (y = = 1) condition fails, the if block is skipped and in the else part in the return statement fact( ) is called with argument (3-1) i.e. 2. This is a recursive call. Since once again the condition (y = = 1) fails fact ( ) is called yet another time, with argument (2-1). This time as x is 1, control goes back to the previous fact ( ) with the value 1, and f is evaluated as 2. Similarly, each fact( ) evaluates its f from the returned value and finally 6 is returned to main ( ).

6. Write four functions for addition, subtraction, multiplication and division. From the main function take the option from the user. Based on the option, call the appropriate function to perform the corresponding operation. The program will behave like a menu-based calculator. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <stdlib.h>
float add(float, float);
float sub(float, float);
float mult(float, float);
float div(float, float);
float add(float x, float y)
{
float z;
z = x + y;
return (z);
}
float sub(float x, float y)
{
return (x - y);
}
float mult(float x, float y)
{
return (x *y);
}
float div(float x, float y)
{
```

```
return (x / y);
}
int main(void)
{
float a, b, c;
int d;
printf("1. Addition\n");
printf("2. subtraction\n");
printf("3. Multiplication\n");
printf("4. Division\n");
printf("Enter your choice 1,2,3,4\n");
scanf("%d", &d);
printf("Enter two nos. \n");
scanf("%f%f", &a, &b);
switch (d)
{
case 1:
c = add(a, b);
break;
case 2:
c = sub(a, b);
break;
case 3:
c = mult(a, b);
break;
case 4:
c = div(a, b);
break;
default:
printf("Wrong choice");
}
printf("The result is %f", c);
return EXIT_SUCCESS;
}
```

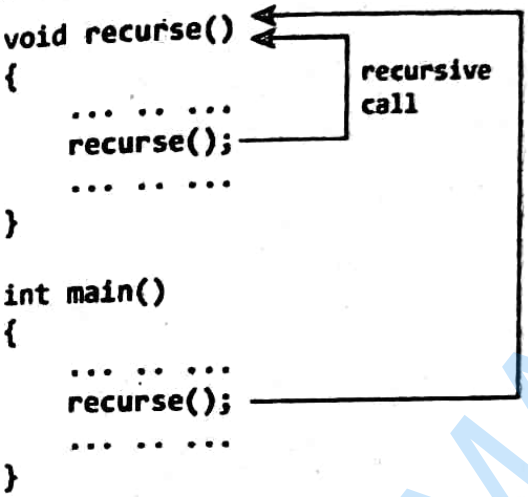
RECURSION

Chapter at a Glance

Recursion

Recursion is a cyclic programmatic behavior or a construct when a subprogram invokes itself or invokes a series of other subprograms that eventually invokes the first subprogram again.

How recursion works



The recursion continues until some condition is met to prevent it. To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int sum(int n);

int main()
{
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);
    return 0;
}
int sum(int num)
{
    if (num!=0)
        return num + sum(num-1); // sum() function calls itself
    else
```

```
        return num;
}
```

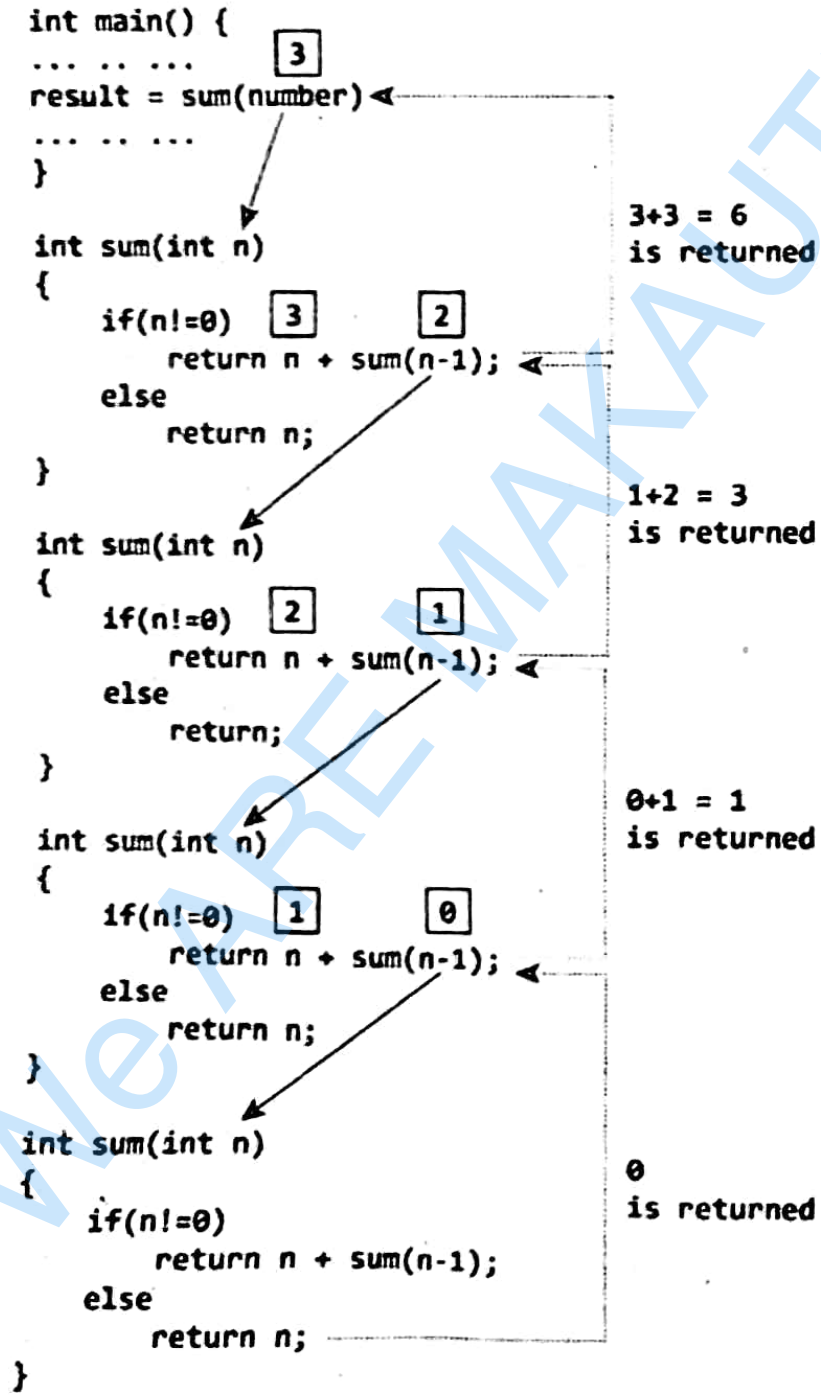
Output

Enter a positive integer:3
sum = 6

Initially, the sum() is called from the main() function with number passed as an argument.

Suppose, the value of num is 3 initially. During next function call, 2 is passed to the sum() function. This process continues until num is equal to 0.

When num is equal to 0, the if condition fails and the else part is executed returning the sum of integers to the main() function.



Advantages and Disadvantages of Recursion

Recursion makes program elegant and more readable. However, if performance is vital then, use loops instead as recursion is usually much slower.

Note that, every recursion can be modeled into a loop.

Recursion Vs Iteration? Need performance, use loops, however, code might look ugly and hard to read sometimes. Need more elegant and readable code, use recursion, however, you are sacrificing some performance.

Very Short Answer Type Questions

1. Recursion is a method in which the solution of a problem depends on _____ [MODEL QUESTION]

a) Larger instances of different problems
b) Larger instances of the same problem
c) Smaller instances of the same problem
d) Smaller instances of different problems

Answer: (c)

2. Which of the following problems can be solved using recursion? [MODEL QUESTION]

a) Factorial of a number
b) Nth fibonacci number
c) Length of a string
d) All of the mentioned

Answer: (d)

3. Recursion is similar to which of the following? [MODEL QUESTION]

a) Switch Case
b) Loop
c) If-else
d) None of the mentioned

Answer: (b)

4. In recursion, the condition for which the function will stop calling itself is _____ [MODEL QUESTION]

a) Best case
b) Worst case
c) Base case
d) There is no such condition

Answer: (c)

5. Consider the following code snippet: [MODEL QUESTION]

```
void my_recursive_function()
{
   my_recursive_function();
}
int main()
{
   my_recursive_function();
   return 0;
}
```

What will happen when the above snippet is executed?

a) The code will be executed successfully and no output will be generated
b) The code will be executed successfully and random output will be generated
c) The code will show a compile time error
d) The code will run for some time and stop when the stack overflows

Answer: (d)

6. What is the base case for the following code? [MODEL QUESTION]

```
void my_recursive_function(int n)
{
      if(n == 0)
      return;
      printf("%d ",n);
      my_recursive_function(n-1);
}
int main()
{
      my_recursive_function(10);
      return 0;
}
```

a) return
b) printf("%d ", n)
c) if(n == 0)
d) my_recursive_function(n-1)

Answer: (c)

7. How many times is the recursive function called, when the following code is executed? [MODEL QUESTION]

```
void my_recursive_function(int n)
{
      if(n == 0)
      return;
      printf("%d ",n);
      my_recursive_function(n-1);
}
int main()
{
      my_recursive_function(10);
      return 0;
}
```

a) 9 b) 10 c) 11 d) 12

Answer: (c)

8. What does the following recursive code do? [MODEL QUESTION]

```
void my_recursive_function(int n)
{
      if(n == 0)
      return;
      my_recursive_function(n-1);
```

```
    printf("%d ",n);
}
int main()
{
    my_recursive_function(10);
    return 0;
}
```

a) Prints the numbers from 10 to 1
b) Prints the numbers from 10 to 0
c) Prints the numbers from 1 to 10
d) Prints the numbers from 0 to 10

Answer: (c)

Short Answer Type Questions

1. Let a and b denotes positive integers. Suppose a function Fun defined as follows:

$$\text{Fun}(a, b) = \begin{cases} 0 & \text{if } a < b \\ \{\text{Fun}(a-b, b)+1\} & \text{if } b <= a \end{cases}$$

Explain the function calls Fun (2, 3) and Fun (14, 3) based on the given definition. **[WBUT 2019]**

Answer:

Fun (2, 3) = a =2, b=3.

Since a<b hence the return to the function will be "0".

Fun (14, 3) = a =14, b=3.

Iteration 1:- b<a, hence Fun(11,3), new a= 11, b = 3
Iteration 2:- b<a, hence Fun(8,3), new a= 8, b = 3
Iteration 3:- b<a, hence Fun(5,3) , new a= 5, b = 3
Iteration 4:- b<a, hence Fun(5,3) , new a= 2, b = 3
Iteration 5:- a<b, hence Fun(2,3) , 0+1=1

```
#include<stdio.h>
intfunc(intx,inty);
int i=1;
intmain(void)
{
printf("\nValue = %d",func(14,3));

return 0;
}

intfunc(int a, int b)
{
if(a<b)
return 0;
elseif(b<=a)
```

```
    {

int x = func((a-b),b)+1;
printf("\nIteration = %d",i,x);
      i++;
return(x);
    }
}
```

Output for Fun (14, 3)

```
Iteration = 1
Iteration = 2
Iteration = 3
Iteration = 4
Final Value = 4
```

2. Differentiate recursion and iteration with suitable example. **[WBUT 2019]**

Answer:

RECURSION	ITERATION
In recursion, function call itself until the base condition is reached.	Iteration means repetition of process until the condition fails. For example – when you use loop (for, while etc.) in your programs
In recursive function, only base condition (terminate condition) is specified.	Iterative approach involves four steps, initialization, condition, execution and update.
Recursion keeps your code short and simple	Iterative approach makes your code longer
Recursion is slower than iteration due to overhead of maintaining stack	Iteration is faster
If recursion is not terminated (or base condition is not specified) than it creates stack overflow (where your system runs out of memory).	In iteration no such constraint.
We can't solve all problems using recursion	Any recursive problem can be solved iteratively

Recursive function

```
int fib_recur ( int n , int next, int result)
{
  if (n == 0)
      return result;
  return fib_recur(n - 1, next + result, next);
}
```

Iterative function

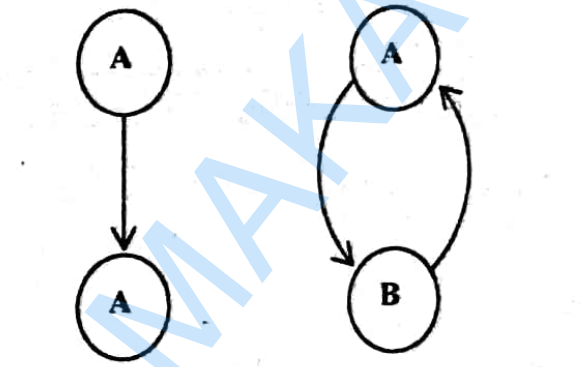
```
   int fact_iter(int n, int result) {
     int temp_n;
```

```
    int temp_result;
    while (n != 1) {
      temp_n = n;
      temp_result = result;
      n = temp_n - 1;
      result = temp_n * temp_result;
    }
    return result;
}
```

3. Which are the different types of recursion? Explain in brief. [MODEL QUESTION]

Answer:

Suppose A is a function which by sequence calls the function A or A is a function which by sequence calls the function B which by sequence calls the function A, in both the cases A is a recursive function. In the first case the recursion is called ***direct recursion*** whereas in the second case the recursion is called ***indirect recursion***.



Direct recursion

Indirect recursion

4. What are the important parts of designing recursive algorithms? [MODEL QUESTION]

Answer:

The important parts of designing recursive algorithms are:

- Find the key step
- Find a stopping rule
- Outline the algorithm
- Check the termination
- Draw the recursion tree.

5. What is a tail recursion? [MODEL QUESTION]

Answer:

A recursive function is tail recursive if the final result of the recursive call is the final result of the function itself. To explain Tail recusrion let us reconsider the factorial program. The highlighted rectangle is the actaul recursive call.

```
int fact(int n){
   if (n==0)
      return 1;
```

```
    else
        return n*fact(n-1);
}
```

The above call is not tail recursive because while returning back from a recursive call, there is still one pending operation, that is the multiplication. However the following program is a guranteed tail recursive call.

```
void prog(int i) {
    if (i>0) {
        prog(i-1);
        System.out.print(i+"");
        prog(i-1);
    }
}
```

The significance of tail recursion is that when making a tail-recursive call, the caller's return position need not be saved on the call stack. When the recursive call returns, it will branch directly on the previously saved return position. Therefore, on compilers that support tail-recursion optimization, tail recursion saves both space and time.

Long Answer Type Questions

1. Write a program in C to add the following series for given value of n.

$$\frac{1}{4!}-\frac{1}{6!}+\frac{1}{8!}-\cdots\cdots\cdots\cdots\frac{1}{n!}$$

[WBUT 2019]

Answer:

```
#include<stdio.h>
Double factorial (double n);
int main()
{
double sum = 0.0;
double fact, start=4;
double n;
int track = 0;
printf("Enter the value of N :\n");
scanf("%lf",&n);
  fact = factorial(start);
printf("****** %lf\n", fact);
  sum = sum + (1/fact);
while(start<=n)
  {
     start = start+2;
if(track ==0)
     {
       fact = factorial (start);
     printf("****** %lf\n", fact);
       sum = sum - (1/fact);
       track = track+1;
     }
```

```
elseif(track == 1)
      {
        fact = factorial (start);
      printf("****** %lf\n", fact);
        sum = sum + (1/fact);
        track = track-1;
      }
    }
printf ("Final Sum = %f", sum);
return 0;
}
Double factorial (double n)
{
if (n == 1)
    {
return 1;
    }
else
    {
return n * factorial (n - 1);
    }
}
```

2. What is Eight Queen Problem? Write a recursive function to solve this puzzle?
[MODEL QUESTION]

Answer:

The problem of 8-Queen is to place 8 Queens in an 8 x 8 chessboard in such a way that no two queens attack each other. (Queens in the chessboard can move in any direction, i.e., row wise, column wise or diagonally).

The general version of this problem is known as N-Queen problem.

In order to design a solution for this problem, we have to keep the following points in mind:

- As it's a n x n chessboard, so will place different queens in different rows. So, n queens will be placed in n different rows. At the end we have to find actually, the column positions of these n queens. We will keep an array x[n]to store the column values off n different queens assuming that queen 1 will be placed in row 1, queen 2 will be placed in row 2 and so on.
- As array x[] stores the column number of different queens, in order to check whether a new queen can be placed in ith column or not, we have to check x[j] = i for all j's, where j denotes the column values for all earlier queens already placed in the chessboard.
- In order to check whether the new queen is on the same diagonal or not with respect to the earlier queens, we have to check

$$Abs(x[j] - i) = Abs(j - k)$$

where the new queen is to be placed in kth row and ith column.

Considering the above conditions, the recursive solution of the above problem is given below:

```
NQueen (k, n)
{
      for(i=1; i<=n; i++)
      {
            if Place(k, i) then
            x[k] = i;
            if (k ==n)
            print(x[1..n]);
            else
            NQueen(k+1, n);
      }
}
```

3. How do we find the time and space requirement of a recursive algorithm? **[MODEL QUESTION]**

Answer:

The time complexity of an algorithm is basically how many operations are necessary to compute it. While for non-recursive algorithms the reasoning is quite straight-forward, for recursive algorithms, it is a little tricky, but still easy to follow.

Let's take a simple example, the recursive algorithm to compute the factorial of a natural number n as in the C language code below.

```
int Factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n*Factorial(n-1);
}
```

First, we compute how many operations there are in a basic call, in this example, it will be 3 operations:

- one for the if condition (is n equal to 0?)
- one for the product of n by the output of the next call of Factorial()
- one to subtract 1 from n in the parameter of the next call of Factorial()

Therefore, calling T(n) the time complexity of the factorial recursive program above, it will be expressed recursively as:

$T(n) = 3 + T(n - 1)$

Deriving it to a general case, and considering that T(0) = 1, the time complexity becomes an expression of order n. It means that the time complexity to compute the factorial of n is a function of n. A complexity like this one is said to be **linear** since it is directly proportional to the parameter n.

To compute the **space complexity**, it is important to observe not the operations, but how much memory is needed in each program execution. Using the factorial example, in each call of Factorial(), it is necessary a fixed amount k of memory. The amount of memory

needed would be directly proportional to the number of calls that the function makes, and since for values of n > 0 this function calls Factorial() just once for n-1, the space complexity S(n) is expressed as:

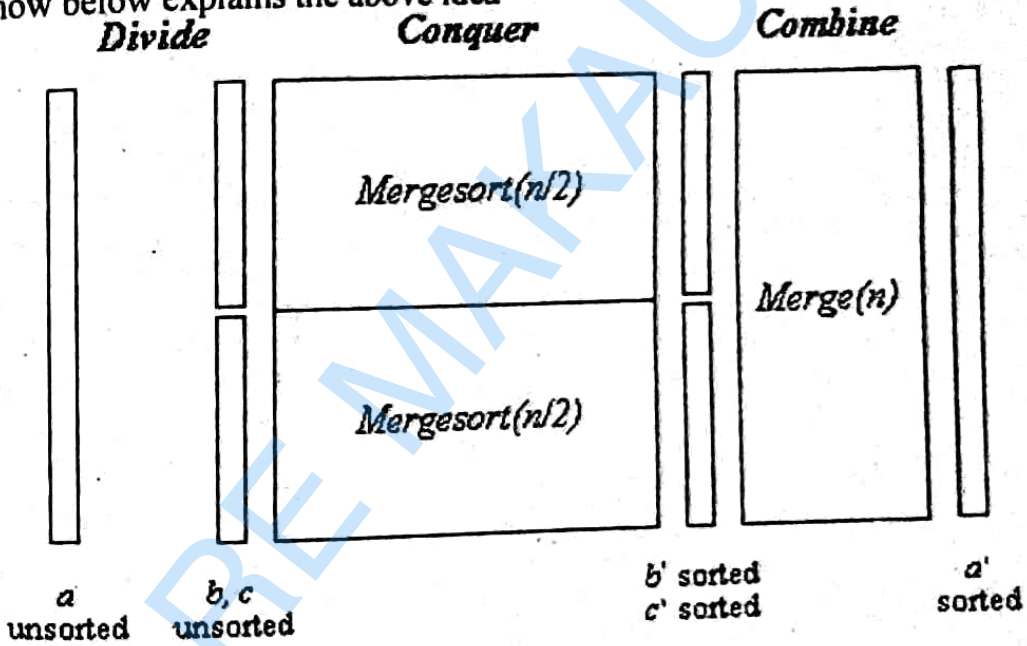
$S(n) = kn \rightarrow O(n)$

Since k is a constant value (a fixed amount of memory for each function call), this complexity is once again a direct function of n, i.e., in terms of space complexity the recursive factorial program is also linear.

4. Explain mergesort algorithm in brief. Write a recursive function of mergesort in C. **[MODEL QUESTION]**

Answer:

The Mergesort algorithm is based on a divide and conquer strategy. First, the sequence to be sorted is decomposed into two halves (Divide). Each half is sorted independently (Conquer). Then the two sorted halves are merged to a sorted sequence (Combine). The figure show below explains the above idea



Divide: Divide the n-element array into two n/2-element subarrays..
Conquer: Sort the two subarrays recursively using merge sort.
Combine: Merge the two sorted subsequences to form the sorted array.

```
#include<stdio.h>
int arr[20];        // array to be sorted

int main()
{
  int n,i;

  printf("Enter the size of array\n");  // input the elements
  scanf("%d",&n);
  printf("Enter the elements:");
  for(i=0;i<n;i++)
```

```
    scanf("%d",&arr[i]);
  merge_sort(arr,0,n-1);   // sort the array
  printf("Sorted array:");   // print sorted array
  for(i=0;i<n;i++)
    printf("%d",arr[i]);
  return 0;
}
int merge_sort(int arr[],int low,int high)
{
  int mid;
  if(low<high)
  {
    mid=(low+high)/2;
   // Divide and Conquer
    merge_sort(arr,low,mid);
    merge_sort(arr,mid+1,high);
   // Combine
    merge(arr,low,mid,high);
  }

  return 0;
}
int merge(int arr[],int l,int m,int h)
{
  int arr1[10],arr2[10];  // Two temporary arrays to
                          hold the two arrays to be merged
  int n1,n2,i,j,k;
  n1=m-l+1;
  n2=h-m;
  for(i=0;i<n1;i++)
    arr1[i]=arr[l+i];
  for(j=0;j<n2;j++)
    arr2[j]=arr[m+j+1];
  arr1[i]=9999;  // To mark the end of each temporary array
  arr2[j]=9999;

  i=0;j=0;
  for(k=l;k<=h;k++)  //process of combining two sorted arrays
  {
   if(arr1[i]<=arr2[j])
     arr[k]=arr1[i++];
   else
     arr[k]=arr2[j++];
  }
    return 0;
}
```

STRUCTURES

Chapter at a Glance

Structure

Unlike Arrays a structure is a data type that holds different data types in one single unit. In C language a structure is represented by the "struct" key word. It is used to declare a new data-type. Basically this means grouping different variables together.

As a simple example, if we want to define a data structure for representing data type Book, then it can be represented by a structure declaration as shown below

```
struct books
{
  char title[20];
  char author[15];
  int pages;
  float price;
};
```

The keyword struct declares a structure to hold the details of four fields namely title, author pages and price. These are members of the structures. Each member may belong to different or same data type. The structure we just declared is not a variable by itself but a template to be used to design the data structure.

We can declare structure variables using the tag name any where in the program. For example the statement,

`struct books book1,book2,book3;` declares book1,book2,book3 as variables of type struct books. Each declaration has four elements of the structure books and holds separate set of values of the structure elements. The complete structure declaration is as shown below

```
struct books
{
char title[20];
char author[15];
int pages;
float price;
};
struct books book1, book2, book3;
```

We can also combine both template declaration and variables declaration in one statement, the declaration

```
struct books
{
char title[20];
char author[15];
int pages;
float price;
} book1, book2, book3;
```

is valid. The use of tag name is optional for example

```
struct
```

```
{
   ...
   ...
   ...
}
```

book1, book2, book3 declares book1,book2,book3 as structure variables representing 3 books bu does not include a tag name for use in the declaration.

Arrays of Structure

As we mentioned earlier than a structure may have members of different types, so in many case we can have an array as one of the members of a structure.

A structure declaration using arrays is given below:

```
struct student
{
      int roll ;
      char name [50] ;
      char sex ;
};
```

The values initialized in the above structure declaration is

```
struct student data[3] = {
            {1, 'M', "X"},
            {2, 'F', "Y"},
            {3, 'M', "Z"}
                        };
```

The above initialized values will be assigned by the compiler as:

```
data[0].roll = 1, data[0].sex = M, data[0].name = X
data[1].roll = 2, data[1].sex = F, data[1].name = Y
data[2].roll = 3, data[3].sex = M, data[3].name = Z
```

If in case, the structure object or any elements of structure are not initialized, the compiler will automatically assign zero to the fields of that particular record.

Nested Structure

The concept of nested structures refers to use of structures within a structure. It means that a structure can also be used as a member field of another structure. The general format of using a structure within another structure is given below:

```
struct first
{
  int x ;
} ;
struct second
{
  int y ;
      struct first z ;
};
struct second a ;
```

In the above declaration, the instance of first structure is being declared in second structure. Since, the first structure is also a member of second, therefore any member field of structure first will be called by the following way:

```
a . z . x ;
```

Unions

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computers memory where as each member within a structure is assigned its own unique storage area. Thus unions are used to conserve memory. They are useful for application involving multiple members. Where values need not be assigned to all the members at any one time. Like structures union can be declared using the keyword union as follows

```
union item
{
  int m;
  float p;
  char c;
} code;
```

This declares a variable code of type union. The union contains three members each with a different data type. However we can use only one of them at a time. This is because if only one location is allocated for union variable irrespective of size. The compiler allocates a piece of storage that is large enough to access a union member we can use the same syntax that we use to access structure members. That is

```
code.m
code.p
code.c
```

are all valid member variables.

Let us consider the following examples

Example: 1

```
struct example
{
      int x ;
      float y ;
      char z ;
};
struct example m ;
```

Example: 2

```
union example
{
      int x ;
      float y ;
      char z ;
} ;
union example  n ;
```

Now, in case of the structure variable m, compiler will allocate 7 bytes whereas in the case of union variable n, compiler will allocate 4 bytes, so that any member of that union can be stored.

Structures and TypeDef

A typedef declaration lets you define your own identifiers that can be used in place of type specifiers such as int, float, and double. A typedef declaration does not reserve storage. The names you define using typedef are not new data types, but synonyms for the data types or combinations of data types they represent. Once a new name is given to the data type, then new variables, arrays, structures and so on can be declared in terms of this new data type. The general syntax of typedef statement is given below:

```
typedef  existing_data_type new_type ;
```

In the above syntax,

- typedef is a keyword for declaring the new datatypes.
- existing_data_type can be any data type for example int, char, float, etc.
- new_type is the newly created datatype.

For example, if we create new datatype as

```
typedef int xyz ;
typedef float abc ;
```

then the variables of integer and float datatypes can be declared as

```
xyz   a, b, c ;
abc   x, y, z ;
```

We can further use the keyword typedef to create new datatype from datatypes being used. For example :

```
typedef  xyz  integer ;
typedef  abc  real ;
```

Usage

The structure's variable declaration without using typedef:

```
struct student
{
      int roll ;
      char name [30] ;
      char sex ;
      char addr [50] ;
} ;
struct student x [100] ;
```

Using typedef:

```
typedef struct student
{
   int roll ;
      char name [30] ;
      char sex ;
      char addr [50] ;
} Student;
Student x[100];
```

Enumerated Data Type

The enumerated data types are also the user defined data types but they are somewhat different, from the typedef data types. Enumerated data type provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. These data types work when you know in advance a finite list of values that a data type can take on. For example, if you want to define a data type for the days of a week, you know in advance the days of week can have seven values, i.e., Sunday, Monday, ...Saturday. Now, we can define an enumerated data type for the above given days.

Note: The variable defined for the enumerated date type can access only the member of that data type, it can't access or assign any other value.

The general format for the enumerated data type is :

```
enum user_define_name {element_1, element_2,... .....element_n} ;
```

In the above syntax,

enum is a keyword telling the compiler about the declaration of enumerated data type.

user_define_name is the name given to the enumerated data type.

For the week, we can define the enumerated data type as follows:

```
enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday} ;
week  obj1 ;
```

Bit Fields

Unlike some other computer languages, C has a built in feature called a bit-field that allows you to access a single bit. Bit fields can be useful for a number of reasons, such as:

- If storage is limited, you can store several Boolean variables in one byte.
- Certain devices transmit status information encoded into one or more bits within a byte.
- Certain encryption routines need to access the bits within a byte.

Although these tasks can be performed using the bitwise operators, a bit-field can add more efficiency to your code.

To access individuals bits, C uses a method based on structure. In fact, a bit field is really just a special type of structure member that defines how long , in bits, the field is to be. The general form of a bit-field definition is

```
struct  struct_type_name
{
        type_name_1 : length ;
        type_name_2 : length ;
        :
        type_name_N : length ;
} variable_list ;
```

Here type is the type of the bit-field and length is the number of bits in the field. A bit-field must be declared as an integral or enumeration type. Bit-field of length 1 should be declared as unsigned because a single bit can't have a sign.

Bit fields are frequently used when analyzing input from a hardware device. For example, the status port of a serial communications adapter might return a status byte organized like this:

Bit	Meaning when set
0	Change in clear-to-send line
1	Change in data-set-ready
2	Trailing edge detected
3	Change in receiving line
4	Clear-to-send
5	Data-set-ready
6	Telephone ringing
7	Receiving Signal

You can represent the information in a status byte using the following bit field:

```
struct status_type
{
        unsigned delta_cts : 1 ;
        unsigned delta_dsr : 1 ;
        unsigned tr_edge   : 1 ;
        unsigned delta_rec : 1 ;
        unsigned cts :       1 ;
        unsigned dsr :       1 ;
        unsigned ring :      1 ;
        unsigned rec_line :  1 ;
} status ;
```

You might use a routine to enable a program to determine when it can send or receive data.

```
status = get_port_status ( ) ;
if (status_cts) printf ( "clear to send") ;
if (status_dsr) printf("data ready") ;
```

To assign a value a bit-field, simply use the form you would use for any other type of structure element. For example, this code fragment clears the ring field:

```
status.ring = 0 ;
```

Bit fields can have certain restrictions. You can't take the address of a bit field. Bit-field can't be arrayed. They can't be declared as static. You can't know, from machine to machine, whether the fields will run from left to right or from right to left; this implies that any code using bit fields may have some machine dependencies.

Very Short Answer Type Questions

1. How many bytes will be allocated for the following code? **[WBUT 2019]**

```
struct A{
        short int a;
        char ch;
        float f;
};
```

a) 7 bytes **b) 8 bytes**
c) no memory will be allocated **d) none of these**

Answer: (b)

2. Which of the following are themselves a collection of different data types? [MODEL QUESTION]
   a) String
   b) Structure
   c) Char
   d) All of the mentioned

Answer: (b)

3. User-defined data type can be derived by_____. [MODEL QUESTION]
   a) struct
   b) enum
   c) typedef
   d) All of the mentioned

Answer: (d)

4. Which of the following cannot be a structure member? [MODEL QUESTION]
   a) Another structure
   b) Function
   c) Array
   d) None of the mentioned

Answer: (b)

5. Number of bytes in memory taken by the below structure is? [MODEL QUESTION]
   a) Multiple of integer size
   b) integer size+character size
   c) Depends on the platform
   d) Multiple of word size

Answer: (b)

Short Answer Type Questions

1. What is structure? What is array of structures? Show with an example. [WBUT 2023]

Answer: *Refer to Chapter at a Glance.*

2. Which is the difference between structure and array? [MODEL QUESTION]

Answer:

ARRAY	STRUCTURE
It is a collection of data items of same data type	It is a collection of data items of different data type
It has declaration only	It has declaration and definition
There is no keyword to represent arrays	keyword struct is used to declare stuctures
array name represents the address of the starting element	Structure name is known as tag it is the short hand notation of the declaration

3. What are structure types in C? [MODEL QUESTION]

Answer:

Structure types are primarily used to store records. A record is made up of related fields. This makes it easier to organize a group of related data

4. What are the important parts of designing recursive algorithms? [MODEL QUESTION]

Answer:
The important parts of designing recursive algorithms are:

- Find the key step
- Find a stopping rule
- Outline the algorithm
- Check the termination
- Draw the recursion tree.

5. What is a self-referential structure? **[MODEL QUESTION]**
Answer:
A structure containing the same structure pointer variable as its element is called as self-referential structure.

Long Answer Type Questions

1. a) What is the difference between Array and Structure? Show with example.
b) What are the differences between Structure and Union? Show with example.
c) Write a program using structure to take 10 students name, roll, marks and print the records with average marks. **[WBUT 2023]**
Answer:
a)

S. No.	Array	Structure
1	**Data Collection:** Array is a collection of homogeneous data.	**Data Collection:** Structure is a collection of heterogeneous data.
2	**Element Reference:** Array elements are referred by subscript.	**Element Reference:** Structure elements are referred by its unique name.
3	**Access Method:** Array elements are accessed by it's position.	**Access Method:** Structure elements are accessed by its object as '-' operator.
4	**Data type:** Array is a derived data type.	**Data type:** Structure is user defined data type.
5	**Syntax:** <data_type>array_name[size];	**Syntax:** **struct** struct_name { Structure_element_1; Structure_element_2; _____ _____ Structure_element_n; }struct_var_nm;
6	**Example:** **int** rn_array[5];	**Example:** **struct** item_mst { int rno; char m_array[50]; }it;

b)

S. No.	STRUCTURE	UNION
1	The amount of memory required to store a structure variable is the sum of the size of all the members.	The amount of memory required is always equal to that required by its largest member.
2	In structure, each member have their own memory space.	In union, one block is used by all the member of the union.

3	**Syntax:**
	`struct struct_name`
	`{`
	`Structure_element_1;`
	`Structure_element_2;`
	`_____`
	`_____`
	`Structure_element_n;`
	`}struct_var_nm;`
		Syntax:
		`union union_name`
		`{`
		`Union_element_1;`
		`Union_element_2;`
		`_____`
		`_____`
		`Union_element_n;`
		`} union_var_nm;`

4	**Example:**
	`struct item_mst`
	`{`
	`int rno;`
	`char m_array[50];`
	`}it;`
		Example:
		`union item_mst`
		`{`
		`int rno;`
		`char m_array[50];`
		`}var 1;`

c)

```
#include <stdio.h>
#include<string.h>
struct student
{
char name[50];
int roll;
float marks[3];
}st[60];
 int main()
{
int i, n, rollno;
float sum =0, avg;
printf ("Enter the number of students \n");
scanf("%d",&n);
for (i=0; i<n; i++)
{
printf("Enter the name of %d student\n",i+1);
scanf("%s",st[i].name);
printf("Enter the roll no. of %d student\n",i+1);
scanf("%s", &st[i].roll);
printf("Enter the marks of %d student \n", i+1);
scanf("%f %f %f", &st[i].marks[0], &st[i].marks[1],
&st[i].marks[2]);
```

```
}
for (i=0; i< n; i++)
{ sum = sum + st[i].marks[0] + st[i].marks[1] + st[i].marks[2];
}
avg = sum / n ;
printf("The average marks of %d students is: %f\n", n, avg);
printf("Enter the roll no. of student detail required:");
scanf("%d",&rollno);
for (i=0; i<n ; i++)
{
if (st[i].roll == rollno)
{
printf("The name of students is : %s \n" , st[i].name);
printf("The 3 test scores are : \n %f %f %f ", st[i].marks[0],
st[i].marks[1], st[i].marks[2]);
break;
}
else
continue;
}
if (i==n)
printf("Roll no. not found\n");
}
```

2. Explain how can we pass an entire structure as parameters to other Functions?
[MODEL QUESTION]

Answer:

A structure variable can be passed as a parameter to a function just like any other variable. When a structure is passed as an argument, each member of the structure is copied. This can sometime prove an expensive operation when the structures are large or functions are called frequently. Usage of pointers in such case would be more efficient.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct Employee
{
  int emp_id;
  char name[25];
  char department[20];
  float salary;
} emp;
int main(void)
{
  struct Employee emp = {1034, "Dennis Ritchie", "Research Labs",
1000.00};
  /* pass the entire structure to display function*/
```

```
  display(emp);
  getch();
  return EXIT_SUCCESS;
}
/* function to display structure variables*/
display(struct Employee emp)
{
  printf("\n%d\n%s,\n%s,\n%f",emp.emp_id,emp.name,emp.salary,
emp.department,);
}
```

3. Write a program to define a structure student to store name, roll number and marks. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
struct student
{
    char name[50];
    int roll;
    float marks;
} s[10];

int main()
{
    int i;
    printf("Enter information of students:\n");
    // storing information
    for(i=0; i<10; ++i)
    {
        s[i].roll = i+1;
        printf("\nFor roll number%d,\n",s[i].roll);
        printf("Enter name: ");
        scanf("%s",s[i].name);
        printf("Enter marks: ");
        scanf("%f",&s[i].marks);
        printf("\n");
    }
    printf("Displaying Information:\n\n");
    // displaying information
    for(i=0; i<10; ++i)
    {
        printf("\nRoll number: %d\n",i+1);
        printf("Name: ");
        puts(s[i].name);
        printf("Marks: %.1f",s[i].marks);
        printf("\n");
```

```
    }
    return 0;
}
```

4. The time at a given point has three elements hours, minutes and seconds. Write a program to add two times. Use a function to add the times and return result.
[MODEL QUESTION]

Answer:

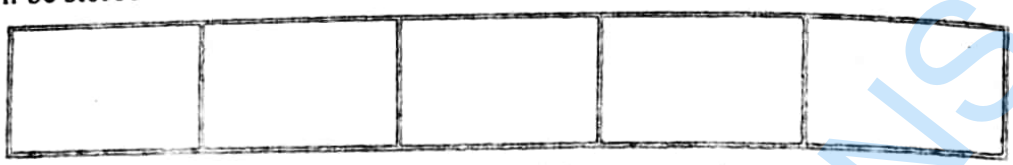
```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef struct timex
{
  int hr;
  int min;
  int sec;
} time;
time addtime(time, time); /* function prototye */
time addtime(time t1, time t2)
{
  time tt;
  int carry = 0;
  tt.sec = t1.sec + t2.sec;
  carry = tt.sec;
  tt.sec = tt.sec % 60;
  tt.min = carry / 60;
  tt.min = tt.min + t1.min + t2.min;
  carry = tt.min;
  tt.min = tt.min % 60;
  tt.hr = carry / 60;
  tt.hr = tt.hr + t1.hr + t2.hr;
  return (tt);
}
int main(void)
{
  time t1, t2, t3;
  printf(" Enter the 1st time  hr  min  sec\n ");
  scanf("%d%d%d", &t1.hr, &t1.min, &t1.sec);
  printf("\nEnter the 2nd time  hr  min  sec\n ");
  scanf("%d%d%d", &t2.hr, &t2.min, &t2.sec);
  t3 = addtime(t1, t2);
  printf(" total time %d hrs :%d  mins: %d  secs ", t3.hr,
t3.min, t3.sec);
  getch();
  return EXIT_SUCCESS;
}
```

5. A factory producing alloys like brass, bronze and stainless steel, wants to keep record of the composition of the metals produced. Brass is an alloy of copper and zinc. Bronze consists of copper and tin. Stainless steel is produced by adding right quantity of nickel and chromium. Factory wants to keep information about 100 alloys.. Write a program using structure and union to efficiently store this information. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define N 2
struct Brass
{
  float cu;
  float zn;
};
struct Bronze
{
  float cu;
  float sn;
};
struct Sls
{
  float ni;
  float cr;
};
union Comp
{
  struct Brass brass;
  struct Bronze bronz;
  struct Sls sls;
};
typedef struct Alloyx
{
  int type;
  union Comp comp;
}Alloy;
int main(void)
{
  Alloy alloy[N];
  int i;
  float n1, n2;
  printf(" Enter type 1. Stainless Steel 2, brass 3. bronze\n");
  printf(" Enter percentage of\n");
  printf(" Nickel and chromium for stainless steel\n");
  printf(" Copper and zinc for Brass \n");
```

Suppose we declare an array arr,
int arr[5] = { 1, 2, 3, 4, 5 };
Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default
We can also declare a pointer of type int to point to the array arr.

```
int *p;
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

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

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

Pointer to Multidimensional Array

A multidimensional array is of form, a[i][j]. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j], a will give the base address of this array, even a + 0 + 0 will also give the base address, that is the address of a[0][0] element.

Here is the generalized form for using pointer with multidimensional arrays.
((a + i) + j)
which is same as,
a[i][j]
Pointer and Character strings
Pointer can also be used to create strings. Pointer variables of char type are treated as string.

```
char *str = "Hello";
```

The above code creates a string and stores its address in the pointer variable str. The pointer str now points to the first character of the string "Hello". Another important thing to note here is that the string created using char pointer can be assigned a value at runtime.

```
char *str;
str = "hello";     //this is Legal
```

The content of the string can be printed using printf() and puts().

printf("%s", str);
puts(str);
Notice that str is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator *.

Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3] = {
    "Jack",
    "Mary",
    "Diane"
};
```

Very Short Answer Type Questions

1. Which of the following about the following two declarations is true? **[WBUT 2019]**

I. `int*f()`
II. `int(*f)()`

a) Both are identical
b) The first is a correct declaration and the second is wrong
c) The first declaration is a function returning a pointer to an integer and the second is a pointer to function returning int.
d) Both are different ways of declaring pointer to a function

Answer: (c)

2. Comment on the following **[MODEL QUESTION]**
const int *ptr;

a) You cannot change the value pointed by ptr
b) You cannot change the pointer ptr itself
c) Both (a) and (b)
d) You can change the pointer as well as the value pointed by it

Answer: (a)

3. Which of the following does not initialize ptr to null (assuming variable declaration of a as int a=0)? **[MODEL QUESTION]**

a) int *ptr = &a;
b) int *ptr = &a – &a;
c) int *ptr = a – a;
d) All of the mentioned

Answer: (a)

4. What is the output of this C code? **[MODEL QUESTION]**

```
int main()
{
        int i = 97, *p = &i;
        foo(&i);
        printf("%d ", *p);
}
```

```
    printf(" Copper and tin for Bronze \n");
    for (i = 0; i < N; i++)
    {
      scanf(" %d%f%f", &alloy[i].type, &n1, &n2);
      switch (alloy[i].type)
      {
        case 1:
          alloy[i]. comp.sls.ni = n1;
          alloy[i]. comp. sls. cr = n2;
          break;
        case 2:
          alloy[i].comp.brass.cu = n1;
          alloy[i]. comp. brass. zn = n2;
          break;
        case 3:
          alloy[i].comp.bronz.cu = n1;
          alloy[i].comp.bronz.sn = n2;
          break;
      }
    }
    for (i = 0; i < N; i++)
    {
      switch (alloy[i].type)
      {
        case 1:
          printf("Composition of stainless steel \n");
          printf("%%Nickel   =%6.2f\n", alloy[i].comp.sls.ni);
          printf("%%Chromium=  %6.2f\n", alloy[i].comp.sls.cr);
          break;
        case 2:
          printf("Composition of brass \n");
          printf("%%Copper=%6.2f\n", alloy[i].comp.brass.cu);
          printf("%% Zinc = %6.2f\n", alloy[i].comp.brass.zn);
          break;
        case 3:
          printf("Composition of bronze \n");
          printf("%%Copper = %6.2f\n", alloy[i].comp.bronz.cu);
          printf("%% Tin = %6.2f\n", alloy[i].comp.bronz.sn);
          break;
      }
    }
    getch();
    return EXIT_SUCCESS;
}
```

POINTERS

Chapter at a Glance

Pointers

A pointer is a special type of variable that can store memory address of any other variable like int, char, float or an array, linked list and structures. For example.

int * ptr; declares a pointer variable which can store address of integer variables.

ptr = &count; stores the address of variable count in ptr. The unary operator '&' returns the address of a variable.

The rules that C uses to distinguish between the contents of variables and their addresses are:

- An asterisk before a variable tells the compiler that this is a pointer variable.
- An asterisk before a pointer variable means that we are referring to the contents of the variable the pointer is addressing to.
- An ampersand before a variable means that we are referring to the address of the variable, not it's contents.

Pointer declaration and Initialization

The first thing to do with pointers is to declare a pointer variable, then set it to point somewhere, and finally manipulate the value that it points to. A simple pointer declaration looks like this:

```
int *ptr1 ; /* declares an integer pointer*/
    float *ptr2; /* declares an float pointer*/
    char *ptr3 ; /* declares an character pointer*/
```

Pointer to a Pointer(Double Pointer)

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as Pointer to Pointer variable or Double Pointer.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator(*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:

```
int ***p2
```

Pointer and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

```
void foo(int *p)
{
        int j = 2;
        p = &j;
        printf("%d ", *p);
}
```

a) 2 97
b) 2 2
c) Compile time error
d) Segmentation fault/code crash

Answer: (a)

5. What is the output of this C code? [MODEL QUESTION]

```
void main()
{
int a[3] = {1, 2, 3};
int *p = a;
int **r = &p;
printf("%p %p", *r, a);
}
```

a) Different address is printed
b) 1 2
c) Same address is printed
d) 1 1

Answer: (c)

Short Answer Type Questions

1. What is dynamic memory allocation? What is the difference between malloc() and calloc() function? [WBUT 2019]

Answer:

The ability to assign memory space during the execution of a program is called dynamic memory allocation.

malloc(): It takes the size in bytes and allocates that much space in the memory. It means that malloc(50) will allocate 50 byte in the memory.

calloc(): It takes the number of elements and the size of each element(in bytes), initializes each element to zero and then returns a void pointer to the memory.

2. What is a NULL pointer? [MODEL QUESTION]

Answer:

A pointer pointing to nothing or no memory location is called null pointer. For doing this we simply assign NULL to the pointer.

So while declaring a pointer we can simply assign NULL to it in following way.

int *p = NULL;

NULL is a constant which is already defined in C and its value is 0. So instead of assigning NULL to pointer while declaring it we can also assign 0 to it.

3. What are the usages of null pointer? [MODEL QUESTION]

Answer:

1. We can simply check the pointer is NULL or not before accessing it. This will prevent crashing of program or undesired output.

```
int *p = NULL;
if(p != NULL) {
//here you can access the pointer
}
```

2. A pointer pointing to a memory location even after its deallocation is called dangling pointer. When we try to access dangling pointer it crashes the program. So to solve this problem we can simply assign NULL to it.

```
void function(){
    int *ptr = (int *)malloc(SIZE);
    . . . . . .
    . . . . . .
    free(ptr);      //ptr now becomes dangling pointer which is
pointing to dangling reference
    ptr=NULL;    //now ptr is not dangling pointer
}
```

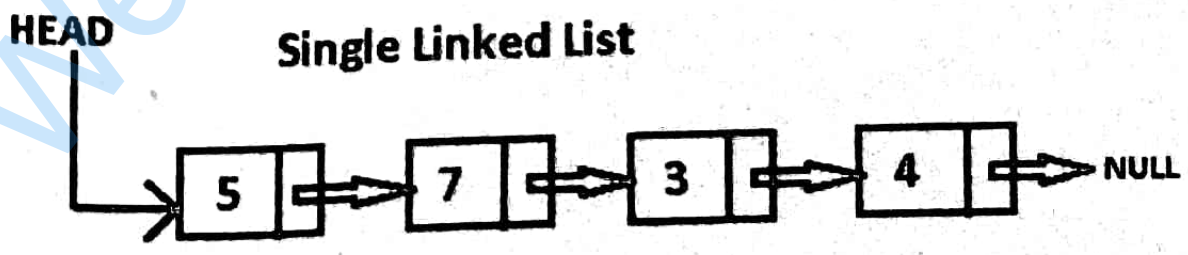
3. We can simply pass NULL to a function if we don't want to pass a valid memory location.

```
void fun(int *p) {
    //some code
}
int main() {

    fun(NULL);

    return 0;
}
```

4. Null pointer is also used to represent the end of a linked list. **[MODEL QUESTION]**



5. When do we use void pointer? **[MODEL QUESTION]**

Answer:

A pointer variable declared using a particular data type cannot hold the location address of variables of other data types. It is invalid and will result in a compilation error.

Ex:- char *ptr;
int var1;
ptr=&var1; // This is invalid because 'ptr' is a character pointer variable.

Here comes the importance of a "void pointer".
A void pointer is nothing but a pointer variable declared using the reserved word in C 'void'.

Ex:- void *ptr; // Now ptr is a general purpose pointer variable

When a pointer variable is declared using keyword void – it becomes a general purpose pointer variable. Address of any variable of any data type (char, int, float etc.)can be assigned to a void pointer variable.

Long Answer Type Questions

1. Write short note on Pointer of Pointer. **[WBUT 2019]**

Answer:

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as Pointer to Pointer variable or Double Pointer.

Syntax:

int **p1;

Here, we have used two indirection operator(*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:

int ***p2

2. Explain pointer arithmetic in detail. **[MODEL QUESTION]**

Answer:

The following operations are valid on pointers variables.

a) Addition of a number to a pointer
b) Subtraction of a number from a pointer
c) Subtraction of one pointer from another

a) Addition of a number to a pointer:

```
int i= 4, .j, .k
j = &I;
j = j+1;
j = j+9;
k = j+3;
```

b) Subtraction of a number from a pointer:

```
int i= 4, *j, *k
j = &I;
j = j-2;
j = j-5;
k = j-6;
```

c) Subtraction of one pointer from another:

```
int arr[  ] = {20,38,40};
int *i, *j;
i = & arr [1] ;
/*address of 2nd element (38) of the array is stored*/
j = & arr [2] ;
/*address of 3rd element (40) of the array is stored */
printf ("%d%d", j - i, *j - *i);
/* j-i will give us the difference of the addresses occupied by 40 & 38 respectively.
*j- *i will however return 10.*/
```

The other arithmetic operators available for use with pointers are

- Unary operators: ++(increment) and -- (decrement)

The following list shows the legal & illegal use of arithmetic operators with pointers.

Let us consider the following declaration

```
int a, b, *p, *q;
```

Pointer operations	Explanation
p= -q	illegal use of pointer
p<<=1;	illegal use of pointer
p = p-q	non portable pointer conversion
p= (int*)(p-q);	valid
p = p+q	invalid pointer addition
p = p*q	illegal use of pointer
p = p*a	illegal use of pointer
p= p/q	illegal use of pointer
p= p/b	illegal use of pointer
p = a/p	illegal use of pointer

3. Explain the different mechanism of dynamic memory allocation and deallocation functions available in C. **[MODEL QUESTION]**

Answer:

The commonly used library functions to implement dynamic memory allocation are malloc, free, calloc and realloc. The prototypes of these functions are defined either in the header file "alloc.h" or in "stdlib.h" or in both.

Malloc

The function most commonly used for dynamic memory allocation is malloc( ). The prototype of the function is given below:

```
void  * malloc (size_t   size)
```

malloc returns a pointer to space for an object of size size, or NULL if the request cannot be satisfied.

The syntax of the function is given below:

```
malloc(b),
```

where b is an unsigned integer which stands for the number of bytes you want to draw from heap. The function returns a pointer, if your request is successful, NULL otherwise.

Program Illustration

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main(void)
{
  void *ptr;
  ptr = malloc(2);
  if (ptr == NULL)
    printf("Memory allocation failed!! \n");
  else
    printf("Memory allocated successfully \n");
  getch();
  return EXIT_SUCCESS;
}
```

Explanation of Program Illustration

The pointer returned by function malloc( ) will be used for storing some value. In the above example, function malloc( ), has allocated two bytes of storage area. Suppose, we want to store an integer number at this address. We want that the pointer returned by the function malloc( ) should point to an integer variable. This is possible by type casting. malloc( ) returns a void pointer. It does not point to any thing. If we want it to point to any type of data we should write

```
(data_type *) malloc(b);
```

where data-type is the type of data to which the pointer returns by the function malloc( ) will point. It can be char, int, float, double or any other structure or union which has been defined by the user. So, here we have to write

```
p = (int *) malloc(2);
```

The above casts the pointer returning by malloc( ) so that it points to an integer variable.

sizeof

In the previous section we requested malloc to allocate 2 bytes for storing an integer number. If that is the case of a structure variable, in that case we have to calculate the no. of bytes required by adding the bytes required, by all structure members. For portability, we should not do it even if it does not cause much inconvenience. Some m/c's may store an integer number in 4 bytes. There is an operator in C, which can solve this problem. The operator is sizeof( ). sizeof(int), will return the bytes required for storing an integer

Note: Though sizeof( ) looks like a function, but it has not been defined in any header file. This is because it is not a function it is an operator.

calloc

Function calloc( ), like malloc( ) allocates memory in a dynamic manner. The prototype of the function is given below:

```
void  * calloc (size_ t  nobj, size_t size)
```

calloc returns a pointer to space for an array of nobj objects, each of size size, or NULL if the request cann't be satisfied.

It takes, as arguments, two values as given below:

```
p = (t *) calloc(n, b);
```

(t *) does the type casting to type t as in case of malloc( ). The above statement, allocates n blocks of memory, each block with b bytes. This function is useful for storing arrays, etc. If each element of an array requires b bytes and the array is required to store k elements we can allocate memory by the following statement

```
p = (t *) calloc(k, b);
```

The same allocation will done by malloc( ), if we call the function as:

```
p = (t *) malloc(k * b);
```

Free

The C compiler allocated memory when a variable is declared and deallocates the memory assigned to a variable when it scope comes to an end. The memory will not be deallocated even when it is no more required, if it has been pulled from the heap, by function malloc( ) or calloc( ). The memory must be returned to the heap, when it is no longer required. This is done with function free().The prototype of the function is given below:

```
void  free (void  *p)
```

free deallocates the space pointed to by p; it does nothing if p is NULL.

The syntax of the statement is as under:

```
free(ptr);
```

where ptr is the pointer to a block of memory which has been allocated from the heap on request. This function is void in nature and does not return anything.

Realloc

Suppose, a block of k bytes of memory has been allocated an request and ptr is a pointer to this block of memory. Suppose, this block has been used to store some records. Let us again assume that we have deleted a no. of records and would like to reduce the block of memory available at this address so that the same may be used elsewhere. This can be done by function realloc().The prototype of the function is given below:

```
void  *realloc (void  *p, size_t size)
```

realloc changes the size of the object pointed to by p to size. The contents will be unchanged upto the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized, realloc returns a printed to the new space, or NULL if the request can't be satisfied, in which case *p is unchanged.

The function takes the following two arguments.

i) The pointer to the block of memory
ii) The size of the now block of memory required at the same address.

The following function call reallocates a memory for storing 100 integer variables at address ptr.

```
realloc(ptr, 100 * sizeof(int));
```

4. Explain pointer as function argument. **[MODEL QUESTION]**

Answer:

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will effect the original variable.

Program Illustration

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

    swap(&m, &n); //passing address of m and n to the swap function
    printf("After Swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}
/*
  pointer 'a' and 'b' holds and  points to the address of 'm' and 'n'
*/
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

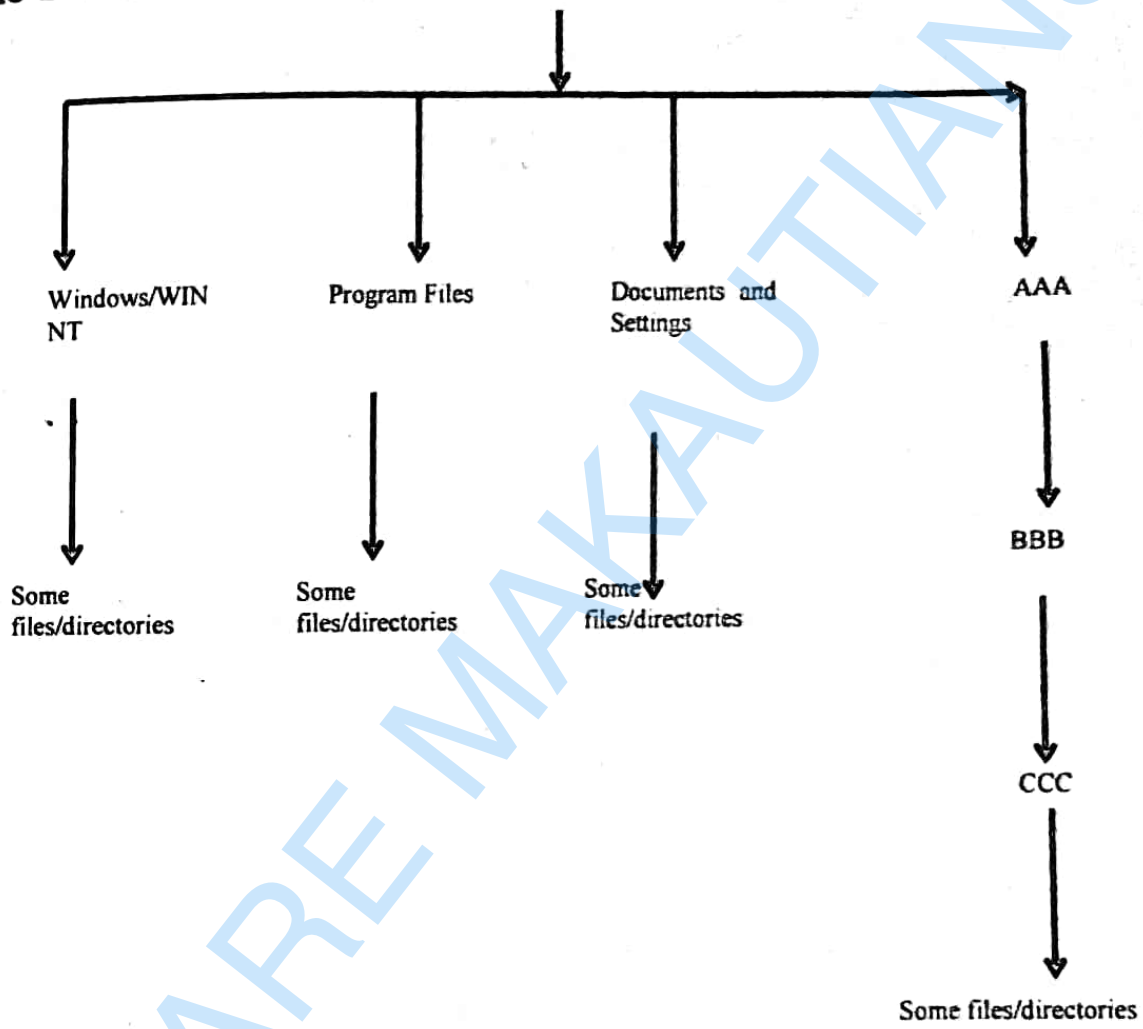
Output

m = 10
n = 20
After Swapping:
m = 20
n = 10

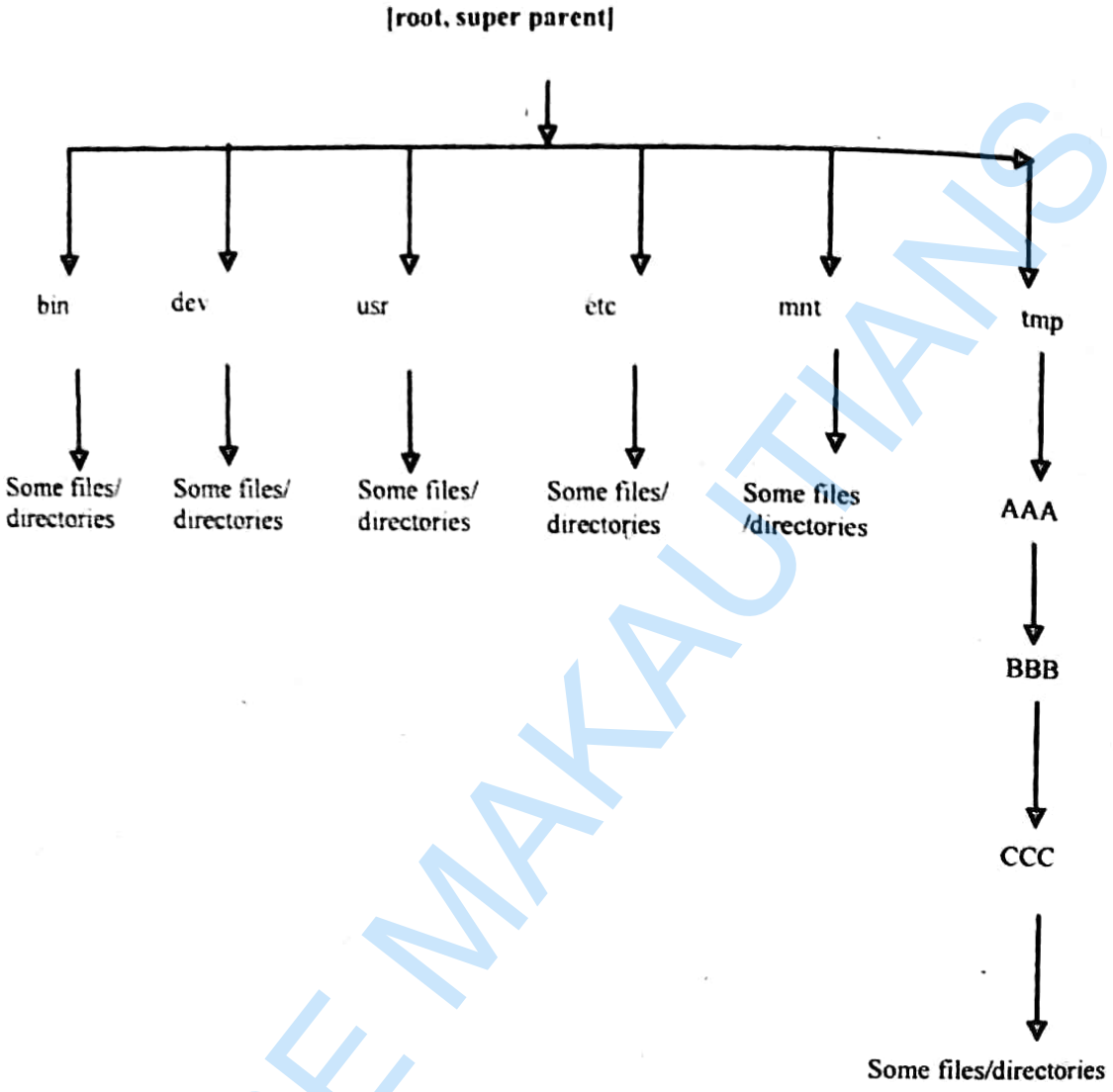
FILES

Chapter at a Glance

File Structure of DOS/Windows



File Structure of DOS/Windows



Very Short Answer Type Questions

1. The value of EOF is [WBUT 2019]

a) –1 b) 0 c) 1 d) 10

Answer: (a)

2. The first and second arguments of fopen are? [MODEL QUESTION]

a) A character string containing the name of the file & the second argument is the mode
b) A character string containing the name of the user & the second argument is the mode
c) A character string containing file pointer & the second argument is the mode
d) None of the mentioned of the mentioned

Answer: (a)

3. For binary files, a ___ must be appended to the mode string. [MODEL QUESTION]
a) Nothing b) "b" c) "binary" d) "01"
Answer: (b)

4. If there is any error while opening a file, fopen will return? [MODEL QUESTION]
a) Nothing b) EOF c) NULL d) Depends on compiler
Answer: (c)

5. Which is true about getc.getc returns? [MODEL QUESTION]
a) The next character from the stream referred to by file pointer
b) EOF for end of file or error
c) Both a & b d) Nothing
Answer: (c)

Short Answer Type Questions

1. What is the difference between windows and Linux file/directory structure? [MODEL QUESTION]

Answer:

Windows	Unix/Linux
C:	/dev/hda1
D:	/dev/hda5
E:	/dev/hda6
CDROM	/dev/cdrom or /mnt/cdrom
3 ½ Floppy drive	/dev/fd0 or /mnt/floppy

2. Explain in brief Linux directories or file strcutures? [MODEL QUESTION]
Answer:
bin: Here most of the executable files of applications are found. This directory can be said to be similar to the directory Windows\command.
boot: This is the place where the files needed for booting Linux are placed. In Windows such files—command.com, io.sys and msdos.sys—are placed in the c:\ drive. Here in unix/linux such files are initrd-2.6.9-1.667.img,vmlinuz
dev: This directory contains files representing driver files for devices like hard disk and floppy drives. It is just like 'system', 'system32' or 'system32/drivers' directory in Windows, which contain driver files for such devices.
etc: This directory consists the configuration files for various applications and system daemons (similar to services in Win NT/2000/XP). You have to manually edit these configuration files. In Windows, this is done through the GUI, so you may never locate and use the configuration files, though many of them do exist. In Windows, such configuration files may be found in Program files, Windows or the directory where the application is installed.
Home: This directory contains the home directories of the users. A user with login as matrix will be assigned a subdirectory named matrix in the home directory. The home directory is the same as Documents and Settings directory in Windows. Note that the root

(administrator) home directory is not located within home. The root's home directory is /root, a subdirectory named root within the ROOT (/) directory.

lib: The lib directory in Linux contains modules (similar to DLLs in Windows). Similar to system or system32 directory in WindowsNT/XP/2000.

mnt: This directory contains the mount points. They represent CD and floppy drives.

opt: This directory is used or can be used by third-party applications (not of the same distribution) as their installation directory. It's not very commonly used as most such applications install themselves (by default) in the /usr directory.

proc: This directory contains files to interface with the Linux kernel. Inexperienced users must not edit any files from this directory as it may cause serious damage to the OS.

sbin: Unix/Linux applications, commands used by the system administrator are located in this directory. Hence, files in this directory will be accessible only to the root user.

tmp: Similar to Windows Temp, the directory used for storing temporary files.

usr: Most Unix/Linux applications usually get installed in this directory. So this is akin to the Program files directory in Windows.

var: This directory is used for storing printer spools, log files, cache files etc. This is similar to logfiles and spool subdirectories of system or system32 directory in Windows.

3. What is Files in C and its Uses? **[MODEL QUESTION]**

Answer:

A file is a collection of related records stored in a secondary storage of the computer. This information is stored permanently. C allows programmers to read and write to the files when required.

4. Explain the various modes in which C files can be handled? [MODEL QUESTION]

Answer:

Mode	Meaning
r	opens a text file in reading mode
w	opens or create a text file in writing mode
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

5. What is the difference between Append and Write Mode? **[MODEL QUESTION]**

Answer:
Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already. The only difference they have is, when you open a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

Long Answer Type Questions

1. a) What is the significance of EOF? **[WBUT 2019]**

Answer:
EOF is a pre-defined MACRO which indicates that there are no characters left in the file to read. It is either denoted by a negative integer or a '\0' character.

b) What is the difference between *a* and *a*$^+$ mode? **[WBUT 2019]**

Answer:

Mode	Meaning
a	opens a text file in append mode
a$^+$	opens a text file in both reading and writing mode

c) Explain the functionality of fseek (), ftell () and rewind (). **[WBUT 2019]**

Answer:
fseek(): It is used to move the reading control to different positions using fseek function.
ftell(): It tells the byte location of current position of cursor in file pointer.
rewind(): It moves the control to beginning of the file.

d) Two files contain sorted list of integers. Write a C program to produce a third file which holds a single sorted, merged list of these two lists. Use command line arguments to specify the file names. **[WBUT 2019]**

Answer:

```
#include<stdio.h>
#include<stdlib.h>
intmain(int argc, char *argv[])
{
printf("Total number of argument passed: %d\n", argc);
  FILE *fptr1 = fopen(argv[1], "r");
  FILE *fptr2 = fopen(argv[2], "r");
  FILE *fptr3 = fopen(argv[3], "w");
char ch;
if (fptr1 == NULL || fptr2 == NULL || fptr3 == NULL)
    {
```

```
puts("\nCould not open files");
exit(EXIT_FAILURE);
    }
while ((ch = fgetc (fptr1)) != EOF)
fputc (ch, fptr3);
while ((ch = fgetc (fptr2)) != EOF)
fputc (ch, fptr3);
puts("Files file1.txt and file2.txt are merged into file3.txt");
fclose (fptr1);
fclose (fptr2);
fclose (fptr3);
return 0;
}
```

2. Write a C program to count the number of words and characters in a file. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    FILE *fptr;
    char ch;
    int wrd=1,charctr=1;
    char fname[20];
    printf("\n Count the number of words and characters in a file :\n");
      printf("-----------------------------------------\n");
      printf(" Input the filename to be opened : ");
      scanf("%s",fname);
    fptr=fopen(fname,"r");
    if(fptr==NULL)
     {
         printf(" File does not exist or can not be opened.");
     }
    else
     {
        ch=fgetc(fptr);
        printf(" The content of the file %s are : ",fname);
        while(ch!=EOF)
          {
              printf("%c",ch);
              if(ch==' '||ch=='\n')
                  {
                      wrd++;
                  }
                  else
```

```
                {
                    charctr++;
                }
            ch=fgetc(fptr);
        }
        printf("\n The number of words in the file %s are :
%d\n",fname,wrd-2);
        printf(" The number of characters in the file %s are :
%d\n\n",fname,charctr-1);
        }
    fclose(fptr);
}
```

3. Write a program in C to append multiple lines at the end of a text file.
[MODEL QUESTION]

Answer:

```
#include <stdio.h>
int main ()
{
  FILE * fptr;
  int i,n;
  char str[100];
  char fname[20];
  char str1;
      printf("\n\n Append multiple lines at the end of a text
file :\n");
      printf("-----------------------------------------------
---\n");
      printf(" Input the file name to be opened : ");
      scanf("%s",fname);
    fptr = fopen(fname, "a");
    printf(" Input the number of lines to be written : ");
    scanf("%d", &n);
    printf(" The lines are : \n");
    for(i = 0; i < n+1;i++)
    {
    fgets(str, sizeof str, stdin);
    fputs(str, fptr);
    }
  fclose (fptr);
 //----- Read the file after appended -------
      fptr = fopen (fname, "r");
      printf("\n The content of the file %s is  :\n",fname);
      str1 = fgetc(fptr);
      while (str1 != EOF)
            {
```

```
                printf ("%c", str1);
                str1 = fgetc(fptr);
            }
    printf("\n\n");
    fclose (fptr);
//------- End of reading -----------------
  return 0;
}
```

4. Write a C program to read name and marks of n number of students from user and store them in a file. **[MODEL QUESTION]**

Answer:

```
#include <stdio.h>
int main()
{
    char name[50];
    int marks, i, num;
    printf("Enter number of students: ");
    scanf("%d", &num);
    FILE *fptr;
    fptr = (fopen("C:\\student.txt", "w"));
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }
    for(i = 0; i < num; ++i)
    {
       printf("For student%d\nEnter name: ", i+1);
       scanf("%s", name);
       printf("Enter marks: ");
       scanf("%d", &marks);
       fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
    }
    fclose(fptr);
    return 0;
}
```