UNIT – 1

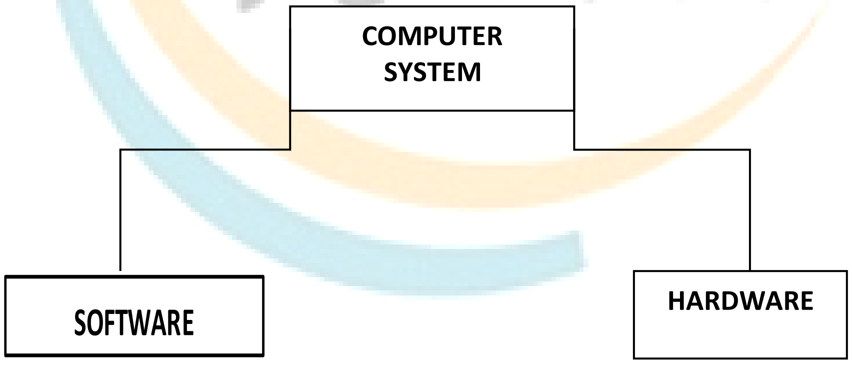
1.1 INTRODUCTION TO PROGRAMMING

A program is a set of instructions that tell the computer to do various things; sometimes the instruction it has to perform depends on what happened when it performed a previous instruction. This section gives an overview of the two main ways in which you can give these instructions, or “commands” as they are usually called. One way uses an *interpreter*, the other a *compiler*. As human languages are too difficult for a computer to understand in an unambiguous way, commands are usually written in one or other languages specially designed for the purpose.

Introduction to Computers:

1.2 Computer Systems: Introduction to components of a Computer System:

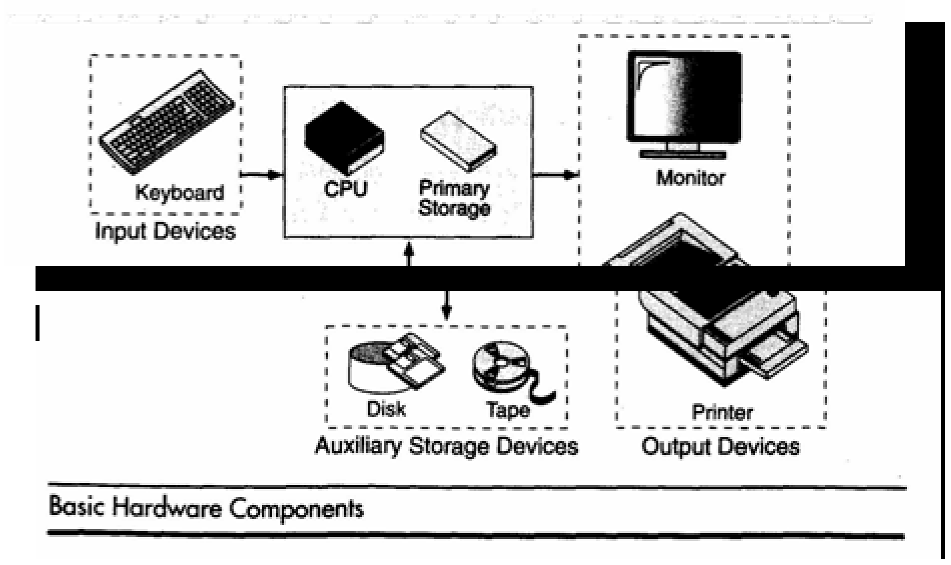
A computer is a system made of two major components: hardware and software. The computer hardware is the physical equipment. The software is the collection of programs (instructions) that allow the hardware to do its job.



Computer Hardware

The hardware component of the computer system consists of five parts: input devices, central processing unit (CPU) , primary storage, output devices, and auxiliary storage devices.

The **input device** is usually a keyboard where programs and data are entered into the computers. Examples of other input devices include a mouse, a pen or stylus, a touch screen, or an audio input unit.



Basic Hardware Components

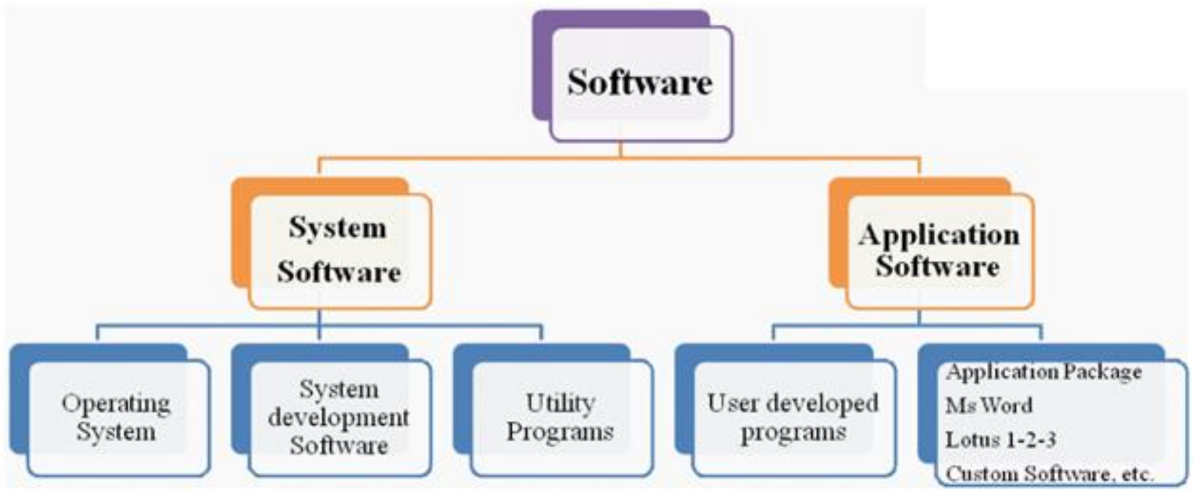
The **central processing unit (CPU**) is responsible for executing instructions such as arithmetic calculations, comparisons among data, and movement of data inside the system. Today's computers may have one ,two, or more CPUs .**Primary storage** ,also known as main memory, is a place where the programs and data are stored temporarily during processing. The data in primary storage are erased when we turn off a personal computer or when we log off from a time-sharing system.

The **output device** is usually a monitor or a printer to show output. If the output is shown on the monitor, we say we have a **soft copy**. If it is printed on the printer, we say we have a hard copy.

Auxiliary storage, also known as **secondary storage** , is used for both input and output. It is the place where the programs and data are stored permanently. When we turn off the computer, or programs and data remain in the secondary storage, ready for the next time we need them.

Computer Software

Computer software is divided in to two broad categories: system software and application software .System software manages the computer resources .It provides the interface between the hardware and the users. Application software, on the other hand is directly responsible for helping users solve their problems.



System Software:

System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The **operating system** provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consists of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category ,**system development software**, includes the language translators that convert programs into machine language for execution ,debugging tools to ensure that the programs are error free and computer –assisted software engineering(CASE) systems.

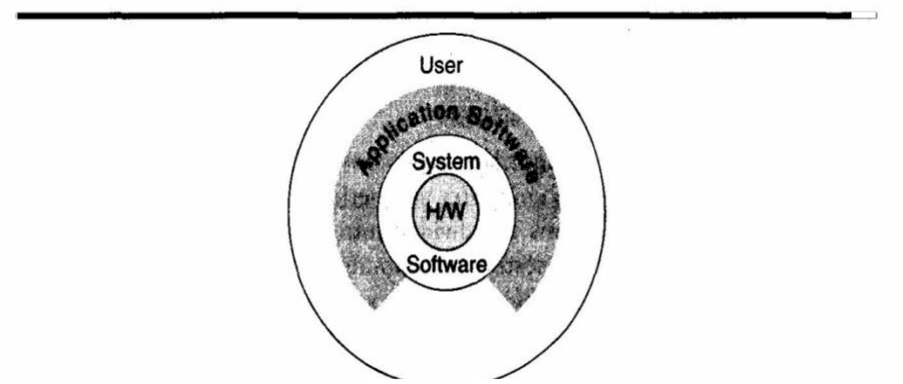
Application software

Application software is broken in to two classes :general-purpose software and application - specific software. **General purpose software** is purchased from a software developer and can be used for more than one application. Examples of general purpose software include word processors, database management systems ,and computer design systems. They are labeled general purpose because they can solve a variety of user computing problems.

Application –specific software can be used only for its intended purpose.

A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed they cannot be used for other generalized tasks.

The relation ship between system and application software is shown in fig-2.In this figure, each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system,the typical user uses some form of application software. The application software in turn interacts with the operating system ,which is apart of the system software layer. The system software provides the direct interaction with the hard ware. The opening at the bottom of the figure is the path followed by the user who interacts directly with the operating system when necessary.



1.2.1Disk

Alternatively referred to as a **diskette**, a **disk** is a hard or floppy round, flat, and magnetic platter capable of having information read from and written to it. The most commonly found disks with a computer are the hard disks and floppy disks (floppy diskette) shown in the picture to the right.

When talking about a disk or diskette, you're referring to a floppy diskette or hard disk drive. However, when talking about a Blu-ray, CD, or DVD, you should use "disc" and not "disk." For example, saying "my movie is on that DVD disc" is the proper usage of "disc." An easy way to remember this is that a "disk" is always a magnetic storage using magnets, and a "disc" is always an optical media using a lasers or lights.

Hard drives

A hard-disk drive (HDD) is a non-volatile device used for storage, located inside the computer case. Like the floppy drive, it holds its data on rotating platters with a magnetic upper exterior which are changed or read by electromagnetic tipped arms that move over the disk as it spins.

1.2.2 Memory

A memory is just like a human brain. It is used to store data and instructions. Computer memory is the storage space in the computer, where data is to be processed and instructions required for processing are stored. The memory is divided into large number of small parts called cells. Each location or cell has a unique address, which varies from zero to memory size minus one. For example, if the computer has 64k words, then this memory unit has 64 * 1024 = 65536 memory locations. The address of these locations varies from 0 to 65535.

Memory is primarily of three types −

- Cache Memory
- Primary Memory/Main Memory
- Secondary Memory

Cache Memory

Cache memory is a very high speed semiconductor memory which can speed up the CPU. It acts as a buffer between the CPU and the main memory. It is used to hold those parts of data and program which are most frequently used by the CPU. The parts of data and programs are transferred from the disk to cache memory by the operating system, from where the CPU can access them.

Advantages

The advantages of cache memory are as follows −

- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

Disadvantages

The disadvantages of cache memory are as follows −

- Cache memory has limited capacity.
- It is very expensive.

Primary Memory (Main Memory)

Primary memory holds only those data and instructions on which the computer is currently working. It has a limited capacity and data is lost when power is switched off. It is generally made up of semiconductor device. These memories are not as fast as registers. The data and instruction required to be processed resides in the main memory. It is divided into two subcategories RAM and ROM.

Characteristics of Main Memory

- These are semiconductor memories.
- It is known as the main memory.
- Usually volatile memory.
- Data is lost in case power is switched off.

- It is the working memory of the computer.
- Faster than secondary memories.
- A computer cannot run without the primary memory.

Secondary Memory

This type of memory is also known as external memory or non-volatile. It is slower than the main memory. These are used for storing data/information permanently. CPU directly does not access these memories, instead they are accessed via input-output routines. The contents of secondary memories are first transferred to the main memory, and then the CPU can access it. For example, disk, CD-ROM, DVD, etc.

Characteristics of Secondary Memory

- These are magnetic and optical memories.
- It is known as the backup memory.
- It is a non-volatile memory.
- Data is permanently stored even if power is switched off.
- It is used for storage of data in a computer.
- Computer may run without the secondary memory.
- Slower than primary memories.

RANDOM ACCESS MEMORY

RAM (Random Access Memory) is the internal memory of the CPU for storing data, program, and program result. It is a read/write memory which stores data until the machine is working. As soon as the machine is switched off, data is erased.

Access time in RAM is independent of the address, that is, each storage location inside the memory is as easy to reach as other locations and takes the same amount of time. Data in the RAM can be accessed randomly but it is very expensive.

RAM is volatile, i.e. data stored in it is lost when we switch off the computer or if there is a power failure. Hence, a backup Uninterruptible Power System (UPS) is often used with computers. RAM is small, both in terms of its physical size and in the amount of data it can hold.

RAM is of two types −

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

Static RAM (SRAM)

The word **static** indicates that the memory retains its contents as long as power is being supplied. However, data is lost when the power gets down due to volatile nature. SRAM chips use a matrix of 6-transistors and no capacitors. Transistors do not require power to prevent leakage, so SRAM need not be refreshed on a regular basis.

There is extra space in the matrix, hence SRAM uses more chips than DRAM for the same amount of storage space, making the manufacturing costs higher. SRAM is thus used as cache memory and has very fast access.

Characteristic of Static RAM

- Long life
- No need to refresh
- Faster
- Used as cache memory
- Large size
- Expensive
- High power consumption

Dynamic RAM (DRAM)

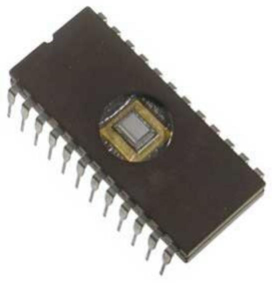
DRAM, unlike SRAM, must be continually **refreshed** in order to maintain the data. This is done by placing the memory on a refresh circuit that rewrites the data several hundred times per second. DRAM is used for most system memory as it is cheap and small. All DRAMs are made up of memory cells, which are composed of one capacitor and one transistor.

Characteristics of Dynamic RAM

- Short data lifetime
- Needs to be refreshed continuously
- Slower as compared to SRAM
- Used as RAM
- Smaller in size
- Less expensive
- Less power consumption

READ ONLY MEMORY

ROM stands for **Read Only Memory**. The memory from which we can only read but cannot write on it. This type of memory is non-volatile. The information is stored permanently in such memories during manufacture. A ROM stores such instructions that are required to start a computer. This operation is referred to as **bootstrap**. ROM chips are not only used in the computer but also in other electronic items like washing machine and microwave oven.



Let us now discuss the various types of ROMs and their characteristics.

MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs, which are inexpensive.

PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM program. Inside the PROM chip, there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

EPROM (Erasable and Programmable Read Only Memory)

EPROM can be erased by exposing it to ultra-violet light for a duration of up to 40 minutes. Usually, an EPROM eraser achieves this function. During programming, an electrical charge is trapped in an insulated gate region. The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use, the quartz lid is sealed with a sticker.

EEPROM (Electrically Erasable and Programmable Read Only Memory)

EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

Advantages of ROM

The advantages of ROM are as follows −

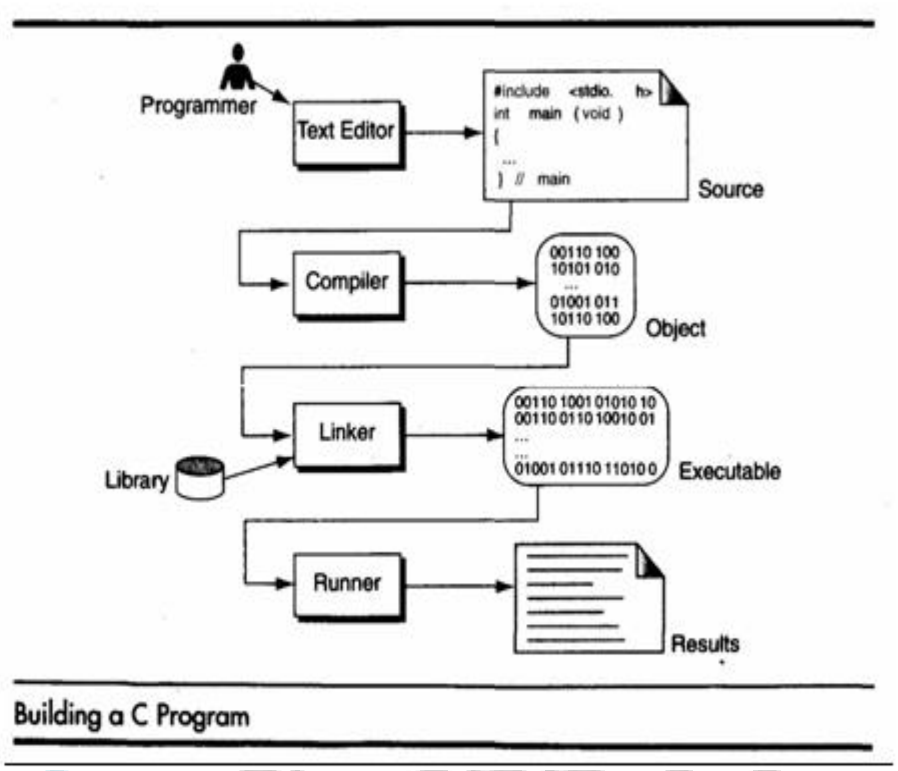
- Non-volatile in nature
- Cannot be accidentally changed
- Cheaper than RAMs
- Easy to test
- More reliable than RAMs
- Static and do not require refreshing
- Contents are always known and can be verified

1.3 Compilers:

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

The processor executes object code, which indicates when binary high and low signals are required in the arithmetic logic unit of the processor.

1.4 Creating and Running Programs:



Building a C Program

Computer hardware understands a program only if it is coded in its machine language. It is the job of the programmer to write and test the program .There are four steps in this process:1.Writing and Editing the program2.Compiliing the program 3.Linking the program with the required library modules 4.Executing the program.

Writing and Editing Programs

The software used to write programs is known as a **text editor.** A text editor helps us enter, change, and store character data. Depending on the editor on our system, we could use it to write letters, create reports, or write programs. The main difference between text processing and program writing is that programs are written using lines of code, while most text processing is done with character and lines.

Text editor is a generalized word processor, but it is more often a special editor included with the compiler. Some of the features of the editor are search commands to locate and replace statements, copy and paste commands to copy or move statements from one part of a program to another, and formatting commands that allow us to set tabs to align statements.
After completing a program, we save our file to disk. This file will be input to the compiler; it is known as a **source file.**

Compiling Programs:

The code in a source file stored on the disk must be translated into machine language , This is the job of the **compiler.** The c compiler is two separate programs. the **preprocessor** and the **translator.** The preprocessor reads the source code and prepares it for the translator. While preparing the code ,it scans for special instructions known as preprocessor commands. These commands tell the preprocessor to look for special code libraries, make substitutions in the code ,and in other ways prepare the code for translation into machine language. The result of preprocessing is called the translation unit.

After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting object module to a file that can then be combined with other precompiled units to form the final program. An object module is the code in machine language. The output of the compiler is machine language code, but it is not ready to run; that is ,it is not executable because it does not have the required C and other functions included.

Linking Programs:

A C program is made up of many functions. We write some of these functions, and they are a part of our source program. There are other functions, such as input/output processes and, mathematical library functions, that exist elsewhere and must be attached to our program. The linker assembles all of these functions, ours and systems into our final executable program.

Executing Programs:

Once program has been linked, it is ready for execution. To execute a program we use an operating system command, such as run, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the loader. It locates the executable program and reads it into memory. When everything is loaded, the program takes control and it begins execution.

In a typical program execution, the reads data for processing , either from the user or from a file. After the program processes the data, it prepares the output. at output can be to the user's monitor or to a file. When the program has finished its job, it tells the operating system , which then removes the program from memory.

1.5 Number systems:

Number systems Conversions Binary, Decimal, Hexadecimal:

Number systems are the technique to represent numbers in the computer system architecture, every value that you are saving or getting into/from computer memory has a defined number system.

Computer architecture supports following number systems.

- **Binary number system**
- **Octal number system**
- **Decimal number system**
- **Hexadecimal (hex) number system**

1) Binary Number System

A Binary number system has only two digits that are **0 and 1**. Every number (value) represents with 0 and 1 in this number system. The base of binary number system is 2, because it has only two digits.

2) Octal number system

Octal number system has only eight (8) digits from **0 to 7**. Every number (value) represents with 0,1,2,3,4,5,6 and 7 in this number system. The base of octal number system is 8, because it has only 8 digits.

3) Decimal number system

Decimal number system has only ten (10) digits from **0 to 9**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8 and 9 in this number system. The base of decimal number system is 10, because it has only 10 digits.

4) Hexadecimal number system

A Hexadecimal number system has sixteen (16) alphanumeric values from **0 to 9** and **A to F**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8,9,A,B,C,D,E and F in this number system. The base of hexadecimal number system is 16, because it has 16 alphanumeric values. Here **A is 10**, **B is 11**, **C is 12**, **D is 13**, **E is 14**and **F is 15**.

CONVERSIONS:

BINARY SYSTEM :

- Binary to decimal conversion
- Binary to octal conversion
- Binary to hexadecimal conversion

Here is an example of converting binary directly into decimal. We simply add up the place values of each 1 digit in the binary number.

Binary to decimal conversion

$100101_2 = 37_{10}$:

Exponents	2^5	2^4	2^3	2^2	2^1	2^0	
Place Values	32	16	8	4	2	1	
Bits	1	0	0	1	0	1	
Value	32			+ 4		+ 1	= 37

Binary to octal conversion

Step 1: Take the given binary number

Step 2: Multiply each digit by 2^{n-1} where n is the position of the digit from the decimal.If it is a decimal number multiply the each digit in the decimal part by $\frac{1}{2^m}$,m is the position of the digit from the decimal point

Step 3: The resultant is the equivalent decimal number for the given binary number.

Step 4: Divide the decimal with 8

Step 5: Note the remainder

Step 6: Continue the above two steps with the quotient till the quotient is zero

Step 7: Write the remainder in the reverse order

Step 8: The resultant is the required octal number for the given binary number.

Question: Convert 1010101_2 to octal

Solution:

Given binary number is 1010101_2

First we convert given binary to decimal

$1010101_2 = (1 * 2^6) + (0 * 2^5) + (1 * 2^4) + (0 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$

= 64 + 0 + 16 + 0 + 4 + 0 + 1

= 64 + 21

= 85 (Decimal form)

Now we will convert this decimal to octal form

8 | 85

----

8 | 10 -- 5

----

8 | 1 -- 2

----

8 | 0 --1

1010101_2 equivalent octal form is 125_8.

Binary to hexadecimal conversion

Step 1: The given number is in binary from.

Step 2: First, we have to change the **binary number into decimal number.**

Step 3: Then, we count the number of binary digits in the given number. Let there be n numbers.

Step 4: Then, we multiply each digit with 2^{n-1}, when n is equal to number of position from right side.

Step 5: Add all numbers after multiplication.

Step 6: Now, the binary number is in decimal number.

Step 7: Now, convert **decimal to hexadecimal**. If the decimal number is less than sixteen, it will be converted by above table.

Step 8: If decimal number is greater than sixteen, it should be divided by 16.

Step 9: Remainder must be less than 16. (It will be converted by table).

Step 10: Then, we write quotient first and then hexadecimal form of remainder together.

Step 11: The resultant is in hexadecimal form of given binary number.

Question 1: Convert 01011011 in hexadecimal number.

Solution:

The given binary number is 01011011

Now, we convert it first to decimal number

So, 01011011 =(0 ×× 277) + (1 ×× 266) + (0 ×× 255) + (1 ×× 244) + (1 ×× 233) + (0 ×× 222) + (1 ×× 211) + (1 ×× 200)
= (0 ×× 128) + (1 ×× 64) + (0 ×× 32) + (1 ×× 16) + (1 ×× 8) + (0 ×× 4) + (1 ×× 2) + (1 ×× 1)
= 0 + 64 + 0 + 16 + 8 + 0 + 2 + 1
= 91 (decimal form of binary number)
Now, we have to change it into hexadecimal number. So, 91 is greater than 16. So, we have to divide by 16.
After dividing by 16, quotient is 5 and remainder is 11.
Remainder is less than 16.
Hexadecimal number of remainder is B.
Quotient is 5 and hexadecimal number of remainder is B.
So, 5B is the hexadecimal number equivalent to decimal number 91.

OCTAL SYSTEM

Octal to Binary conversion

Step 1: Consider the given octal number
Step 2: Let the given number have n number of digits
Step 3: Multiply each digit of the number with 8^{n-1}, when the digit is in the nth position from the right end of the number.If the number has decimal part the multiply each digit in the decimal part by $\frac{1}{8^m}$ when the digit is in the mth position from the decimal point.
Step 4: Add all terms after multiplication
Step 5: The obtained value is the equivalent decimal number
Step 6: Consider the decimal number, divide it by 2
Step 7: Note the remainder
Step 8: Continue the above two steps for the quotient till the quotient is zero
Step 9: Write the remainders in the reverse order
Step 10: The obtained number is the equivalent binary number for the given octal number.

Question 1: Convert 41_8 to a binary number.

Solution:

Given number is 41_8

41_8

= $(4 * 8^1) + (1 * 8^0)$

= 4 * 8 + 1 * 1

= 32+1

= 33(Decimal number)

Now convert this decimal number to a binary number.

```
2 | 33
2 | 16 -- 1
2 | 8  -- 0
2 | 4  --0
2 | 2  -- 0
    1 -- 0
```

The binary number is 100001_2

$41_8 = 100001_2$

Octal to Decimal conversion

Step 1: Take the given octal number.

Step 2: Find out the number of digits in the number

Step 3: Let it have n digits.

Step 4: Multiply each digit in the number with 8^{n-1},when the digit is in the nth position.

Step 5: Add all digits after multiplication.

Step 6: The resultant is the equivalent decimal to the given octal number.

If octal number contains a decimal point

Step 7: Let m digits are there after the decimal

Step 8: Multiply each digit after decimal with $\frac{1}{8^m}$,when the digit is the mth position.

All other steps are same as above.

Example: Convert 5746_8 to decimal number

Solution:

The given number is 5746_8

5746_8

$= (5 * 8^3)+ (7 * 8^2) + (4 * 8^1) + (6 * 8^0)$

$=5 * 512 + 7 * 64 + 4 * 8 + 6 * 1$

$= 2560+448+32+6$

$= 3046$

The equivalent decimal number for 5746_8 is 3046

$5746_8 = 3046$

Octal to hexadecimal conversion

Step 1: Let the number of digits in the number be n

Step 2: Multiply the digits with $8^{n-1}$where n is position of digit from the right end of the number.If the number has decimalpart the multiply digits after decimal by $\frac{1}{8^m}$ where m is position of the number from the decimal

Step 3: Add the terms after multiplication

Step 4: The obtained number is equivalent decimal number to the given octal

Step 5: Consider the decimal number,divide it by 16

Step 6: Note the remainder.

Step 7: Continue the process till the quotient in zero

Step 8: Write the remainder in the reverse order

Step 9: The obtained number is equivalent hexadecimal number to the given octadecimal number.

Example: Convert 1002_8 to hexadecimal

Solution:

The given number is 1002_8

1002_8

$= (1 * 8^3) + (0 * 8^2) + (0 * 8^1) + (2 * 8^0)$

$=1 * 512 + 0 * 64 + 0 * 8 + 2 * 1$

$= 512 + 0 + 0 + 2$

= 514(decimal number)

Now we convert the above decimal to hexadecimal

```
16 | 514
16 | 32   --2
        2 -- 0
```

The hexadecimal number is 202

$1002_8 = 202_{16}$

DECIMAL SYSTEM

Decimal to Binary conversion

Step 1: Divide given number starting from 2 as suitable.

Step 2: Write remainder on the right side of quotient.

Step 3: Divide untill quotient will be 0.

Step 4: Now write binary number starting from lower end of that divison.

Step 5: Now write given number including quotient of lower end that should be starting point.

Example: Convert 35 into binary number.

Solution:

The binary number can be calculated by using L division method

```
2 | 35
-------
2 | 17 -- 1
-------
2 | 8 -- 1
-------
2 | 4 --0
```

----

2 | 2 -- 0

----

1 -- 0

answer is $(100011)_2$.

Decimal to octal conversion

Step 1: Take the given decimal number

Step 2: If the number is less than 8 the octal number is the same

Step 3: If the number is greater than 7 then Divide the number with 8

Step 4: Note the remainder

Step 5: Carry on the step 3 and 4 with the qoutient till it is less than 8

Step 6: Write the remainders in reverse order(bottom to top)

Step 7: The resultant is the equivalent octal number to the given decimal number

Question 1: Convert 128 to octal form.

Solution:

Given decimal number is 128

Start the division process

8 | 128

8 | 16 -- 0

8 | 2 -- 0

8 | 0 –2

answer is The equivalent octal number for 128 is $(200)_8$.

Decimal to Hexadecimal conversion

Step 1: 0 to 15 we can covert directly by the table.

Step 2: For other numbers. Divide the decimal number by 16.

Step 3: Remainder will always be less than 16.

Step 4: Quotient will write first

Step 5: Convert remainder by the help of table.

Step 6: After Quotient we will write the hexadecimal number of remainder.

Example: Convert 146 to hexadecimal number?

Solution:

146 is greater than 16 , so we have to divide by 16.

After dividing by 16 , quotient is 9 and remainder is 2.

remainder is less than 16.

the hexadecimal number of remainder is 2.

Quotient is 9 and hexadecimal number of remainder is 2.

so, the 92 is the hexadecimal number is equivalent to decimal number 146.

```
 16 | 146
 -------------
16 |  9 -- 2
-------------
       0 – 9
```

HEXADECIMAL CONVERSIONS

Hexadecimal to Binary conversion

Step 1: Take given hexadecimal number

Step 2: Find the number of digits in the decimal

Step 3: If it has n digits,multiply each digit with $2^{n-1}$where the digit is in the nth position

Step 4: Add the terms after multiplication

Step 5: The resultant is the decimal number equivalent to the given hexadecimal number.Now we have to convert this hexadecimal to binary number.

Step 6: Divide the decimal number with 2

Step 7: Note the remainder

Step 8: Do the above 2 steps for the quotient till the quotient is zero

Step 9: Write the remainders in the reverse order.

Step 10: The resultant is the required binary number.

Example: Convert $A2B_{16}$ to a equivalent binary number

Solution:

Given hexadecimal number is A2B

$A2B_{16} = (A * 16^2) + (2 * 16^1) + (B * 16^0)$

$= (A * 256) + (2 * 16) + (B * 1)$

$= (10 *256) + 32 + 11$

$= 2560 + 43$

= 2603(Decimal number)

Now we have to convert 2603 to binary

2 | 2603

2 |1301 -- 1

2 | 650 -- 1

2 | 325 -- 0

2 | 162 -- 1

2 | 81 -- 0

2 | 40 -- 1

2 | 20 -- 0

2 | 10 -- 0

2 | 5 -- 0

2 | 2 -- 1

2 | 1 -- 0

2 | 0 -- 1

The binary number is 101000101011_2

$A2B_{16} = 101000101011_2$

Hexadecimal to octal conversion

Step 1: Consider the given hexadecimal number

Step 2: First count the number of digits in the number

Step 3: If n is the position of the digit from the right end then multiply each digit with 16^{n-1}

Step 4: Add the terms after multiplication

Step 5: Resultant is the equivalent decimal form

Step 6: Divide the decimal number with 8

Step 7: Note down the remainder

Step 8: Continue step 6 and 7 with the quotient, until the quotient is zero

Step 9: Write the remainders in reverse order

Step 10: The obtained number is the required result

Example : Find the equivalent octal form of $C1_{16}$

Solution:

Given hexadecimal number is C1

$C1_{16}$

$= (C * 16^1) + (1 * 16^0)$

= C * 16 + 1 * 1

=12 * 16 + 1

= 192 + 1

=193 (Decimal form)

Now we have to convert this decimal to octal

8 | 193

8 | 24 -- 1

8 | 3 -- 0

8 | 0 --3

The octal number is 301_8

$C1_{16}$ = 3018

Hexadecimal to Decimal conversion

Step 1: First we find the number of hexadecimal digits in the number. Let there be n numbers.

Step 2: Then we multiply each hexadecimal digit with 16n−116n−1, when n is equal to number of position from right side.

Step 3: Then we add each number after multiplication.

Step 4: The resultant is equivalent hexadecimal number of the given decimal number.

Example : Convert 7B16 into decimal number.

Solution:

Given hexadecimal number is 7B1616.

7B1616

= 1611 ×× 7 + 1600 ×× B

= 16 ×× 7 + 1 ×× B

= 112 + 1 ×× 11

= 112 + 11

= 123

Answer is 123

1.6 Introduction to Algorithms: steps to solve logical and numerical problems

Definition of Algorithm

To write a logical step-by-step method to solve the problem is called algorithm, in other words, an algorithm is a procedure for solving problems. In order to solve a mathematical or computer problem, this is the first step of the procedure. An algorithm includes calculations, reasoning and data processing. Algorithms can be presented by natural languages, pseudo code and flowcharts, etc.

1.6.1 Example 1: Print 1 to 20:

Algorithm:

```
Step 1: Initialize X as 0,
Step 2: Increment X by 1,
Step 3: Print X,
Step 4: If X is less than 20 then go back to step 2.
```

Example 2: Convert Temperature from Fahrenheit (℉) to Celsius (℃)

Algorithm:

```
Step 1: Read temperature in Fahrenheit,
Step 2: Calculate temperature with formula C=5/9*(F-32),
Step 3: Print C,
```

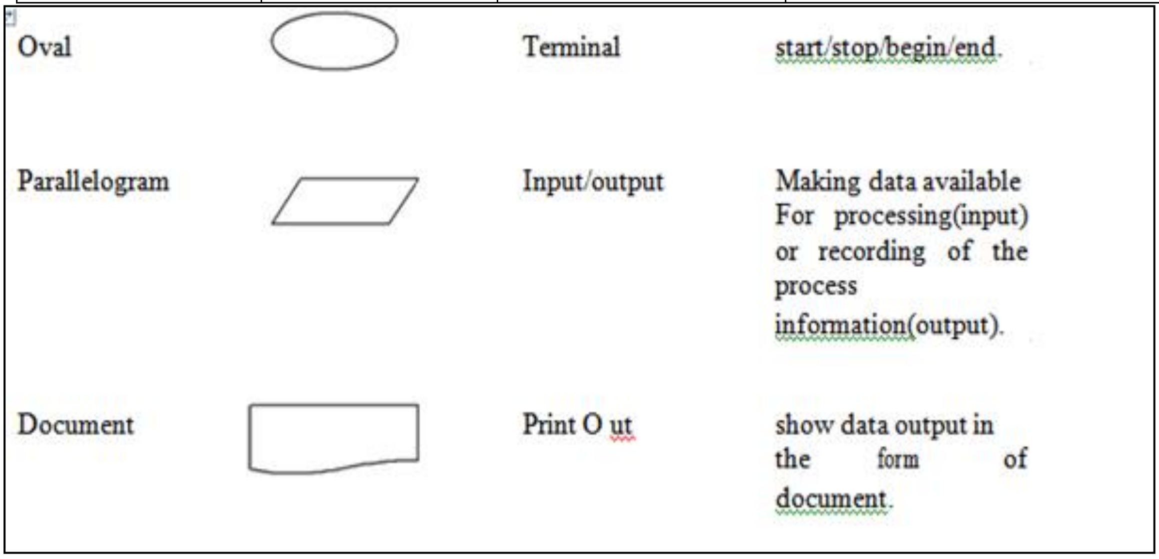
Definition of Flowchart

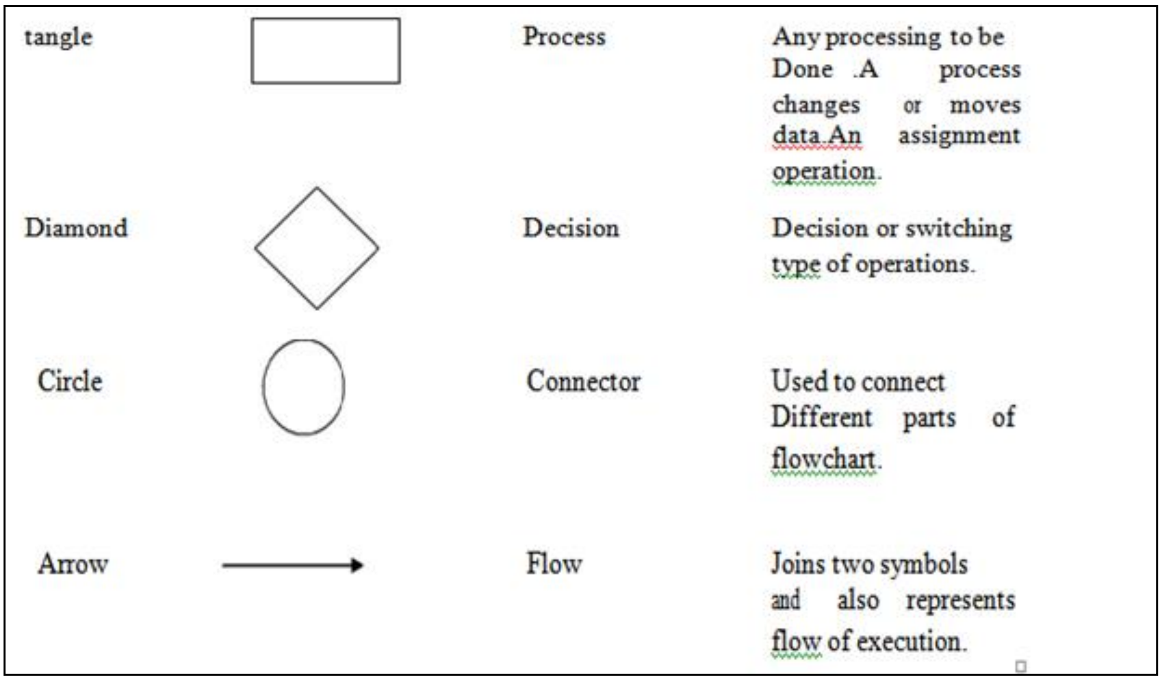
A Flow chart is a Graphical representation of an Algorithm or a portion of an Algorithm. Flow charts are drawn using certain special purpose symbols such as Rectangles, Diamonds, Ovals and small circles. These symbols are connected by arrows called flow lines.

(or)

The diagrammatic representation of way to solve the given problem is called flow chart.

The following are the most common symbols used in Drawing flowcharts:





Shape Type of the symbol	Symbol	Name of the Symbol	Meaning
Oval		Terminal	start/stop/begin/end.
Parallelogram		Input/output	Making data available For processing(input) or recording of the process information(output).
Document		Print O ut	show data output in the form of document.
tangle		Process	Any processing to be Done .A process changes or moves data.An assignment operation.
Diamond		Decision	Decision or switching type of operations.
Circle		Connector	Used to connect Different parts of flowchart.
Arrow		Flow	Joins two symbols and also represents flow of execution.

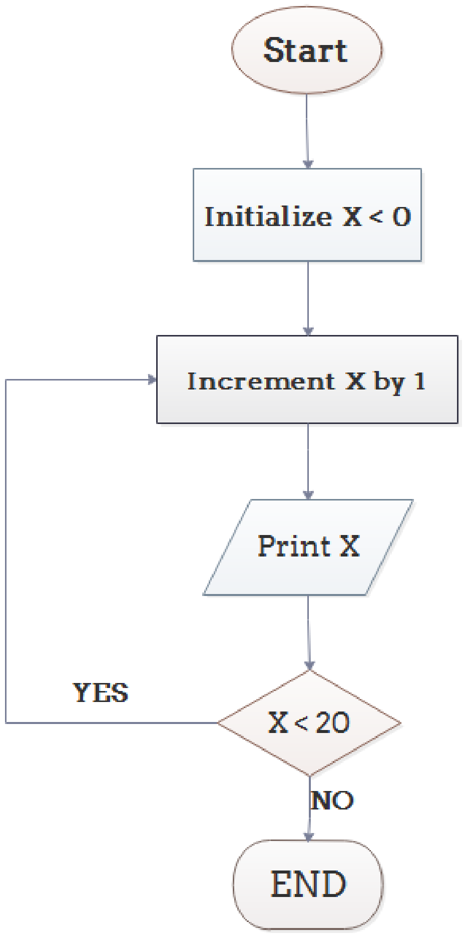
The graphics above represent different part of a flowchart. The process in a flowchart can be expressed through boxes and arrows with different sizes and colors. In a flowchart, we can easily highlight a certain element and the relationships between each part.

1.6.2 How to Use Flowcharts to Represent Algorithms

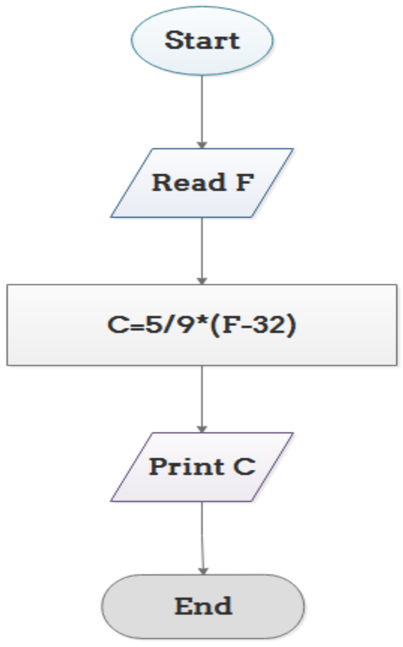
Now that we have the definitions of algorithm and flowchart, how do we use a flowchart to represent an algorithm?

Algorithms are mainly used for mathematical and computer programs, whilst flowcharts can be used to describe all sorts of processes: business, educational, personal and of course algorithms. So, flowcharts are often used as a program planning tool to visually organize step-by-step process of a program. Here are some examples:

Example 1: Flow chart for printing 1 to 20:



Example 2: Flow chart for converting Temperature from Fahrenheit (℉) to Celsius (℃)



PSEUDOCODE:

Pseudo code is an artificial and informal language that helps programmers develop algorithms. Pseudo code is similar to everyday English, it is convenient and user friendly although it is not an actual computer programming language. Psuedo code programs are not actually executed on computer rather they merely help the programmer "think out" a program before attempting to write it in a programming language such as C.

1.7 Program Design and structured programming:

1.7.1 Program Design Language (or **PDL**, for short) is a method for designing and documenting methods and procedures in software. It is related to pseudo code, but unlike pseudo code, it is written in plain language without any terms that could suggest the use of any programming language or library.

PDL was originally developed by the company Caine, Farber & Gordon and has been modified substantially since they published their initial paper on it in 1975. It has been described in some detail by Steve McConnell in his book *Code Complete*.

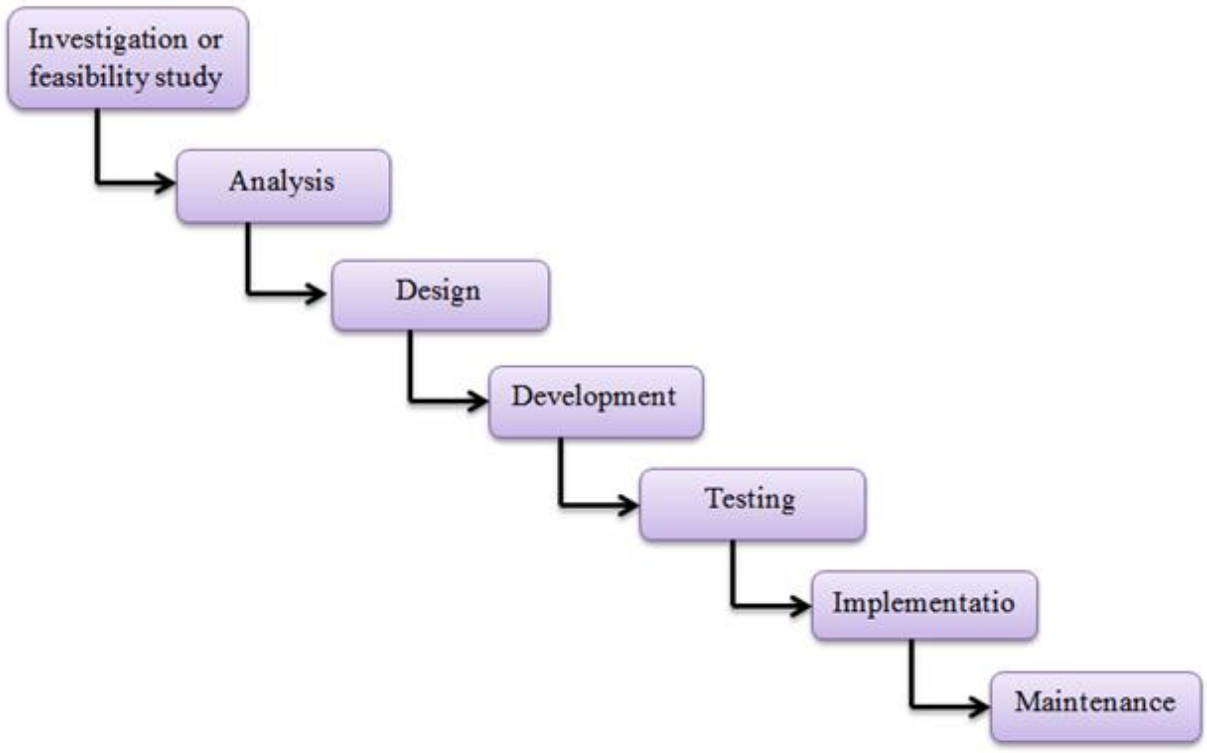
1.7.2 Program development System Development Method

A **software development process** is a structure imposed on the development of a software product. This critical process determines the overall quality and success of our program. If we carefully design each program using good structured development techniques, programs will be efficient, error-free, and easy to maintain.

System Development Life Cycle

Today's large-scale modern programming projects are built using a series of interrelates phases commonly referred to as the system development cycle. Although the exact number and names of the phases differ depending on the environment there is general agreement as to the steps that must be followed. One very popular development life cycle developed, this modal consists of between 5 and 6 phases.

The water fall modal starts with systems requirements in this phase the systems analyst defines requirements that specify what the proposed system is to accomplish.



The *requirements* are usually stated in terms that the user understands.
The *analysis phase* looks at different alternatives from a systems point of view while the design phase determined how the system will be built.

In the *design* phase the functions of the individual programs that will make up the system are determined and the design of the files and / or the databases is completed.
Finally in the *4th phase code*, we write the programs. After the programs have been written and tested to the programmer's satisfaction, the project proceeds to the system test.

All of the programs are tested together to make sure of the system works as a whole
The final phase maintenance keeps the system working once it has been put into production.

Although the implication of the water falls approach is that the phases flow in a continuous stream from the first to the last, this is not really the case. As each phase is developed, errors and omissions will often be found in the previous work. When this happens it is necessary to go back to the previous phase to rework it for consistency and to analyze the impact caused by the changes.

PROGRAM DEVELOPMENT STEPS:

Program Development is a multistep process that requires that we understand the problem, develop a solution, write the program, and then test it. When we are given the assignment to develop a program, we will be given a program requirements statement and the design of any program interfaces. We should also receive an overview of the complete project so that we will take the inputs we are given and convert them to the outputs that have been specified. This is known as program design.

Understand the Problem

The first step in solving any problem is to understand it. By reading the requirements statements carefully, we fully understand it, we review our understanding with the user and the systems analyst to know the exact purpose.

Develop the solution
Once we fully understand the problem we need to develop our solution. Three tools will help in this task. 1. Structure chart, 2.Psuedocode &3.Flowcharts. Generally we will use structure chart and either flowchart or Pseudo code

The structure chart is used to design the whole program .Pseudo code and flowcharts are used to design the individual parts of the program.

Structure chart: A structure chart, also known as hierarchy chart, shows the functional flow through our program. The structure chart shows how we are going to break our program into logical steps each step will be a separate module. The structure chart shows the interaction between all the parts (modules) of our program.

We can use flowchart or pseudo code to complete the design of your program will depend on experience and difficulty of the program your designing.

Write the program

When we write a program, we start with the top box on the structure chart and work our way to the bottom. This is known as top-down implementation. We will write the programs by using structure chart and flowchart or pseudo code.

Test the Program

Program testing can be a very tedious and time- consuming part of program development. As the programmer we are responsible for completely testing our program. In large-development projects test engineers are responsible for testing to make sure all the programs work together.

1.8 INTRODUCTION TO ‘C’ LANGUAGE:

1.8.1 BACK GROUND OF C:

C language facilitates a very efficient approach to the development and implementation of computer programs. The History of C started in 1972 at the Bell Laboratories, USA where Dennis M. Ritchie proposed this language. In 1983 the American National Standards Institute (ANSI) established committee whose goal was to produce “an unambiguous and machine independent definition of the language C” while still retaining it’s spirit .

C is the programming language most frequently associated with UNIX. Since the 1970s, the bulk of the UNIX operating system and its applications have been written in C. Because the C language does not directly rely on any specific hardware architecture, UNIX was one of the first portable operating systems. In other words, the majority of the code that makes up UNIX does not know and does not care which computer it is actually running on. Machine-specific features are isolated in a few modules within the UNIX kernel, which makes it easy for you to modify them when you are porting to a different hardware architecture.

C was first designed by Dennis Ritchie for use with UNIX on DEC PDP-11 computers. The language evolved from Martin Richard's BCPL, and one of its earlier forms was the B language, which was written by Ken Thompson for the DEC PDP-7. The first book on C was *The C Programming Language* by Brian Kernighan and Dennis Ritchie, published in 1978.

In 1983, the American National Standards Institute (ANSI) established a committee to standardize the definition of C. The resulting standard is known as *ANSI C,* and it is the recognized standard for the language, grammar, and a core set of libraries. The syntax is slightly different from the original C language, which is frequently called K&R for Kernighan and Ritchie. There is also an ISO (International Standards Organization) standard that is very similar to the ANSI standard. It appears that there will be yet another ANSI C standard officially dated 1999 or in the early 2000 years; it is currently known as "C9X."

Algorithms and flowcharts are two different tools used for creating new programs, especially in computer programming. An algorithm is a step-by-step analysis of the process, while a flowchart explains the steps of a program in a graphical way.

1.8.2 BASIC STRUCTURE OF C LANGUAGE:

The program written in C language follows this basic structure. The sequence of sections should be as they are in the basic structure. A C program should have one or more sections but the sequence of sections is to be followed.

```
Documentation section
Linking section
Definition section
Global declaration section
Main function section
{
Declaration section Executable section
}
Sub program or function section
```

DOCUMENTATION SECTION: comes first and is used to document the use of logic or reasons in your program. It can be used to write the program's objective, developer and logic details. The documentation is done in C language with /* and */ . Whatever is written between these two are called comments.

LINKING SECTION: This section tells the compiler to link the certain occurrences of keywords or functions in your program to the header files specified in this section.

e.g. #include <stdio.h>

DEFINITION SECTION: It is used to declare some constants and assign them some value.

e.g. #define MAX 25

Here #define is a compiler directive which tells the compiler whenever MAX is found in the program replace it with 25.

GLOBAL DECLARATION SECTION : Here the variables which are used through out the program (including main and other functions) are declared so as to make them global(i.e accessible to all parts of program)
e.g. int i; (before main())

MAIN FUNCTION SECTION: It tells the compiler where to start the execution from
main()
{
point from execution starts
}
main function has two sections

- declaration section : In this the variables and their data types are declared.
- Executable section : This has the part of program which actually performs the task we need.

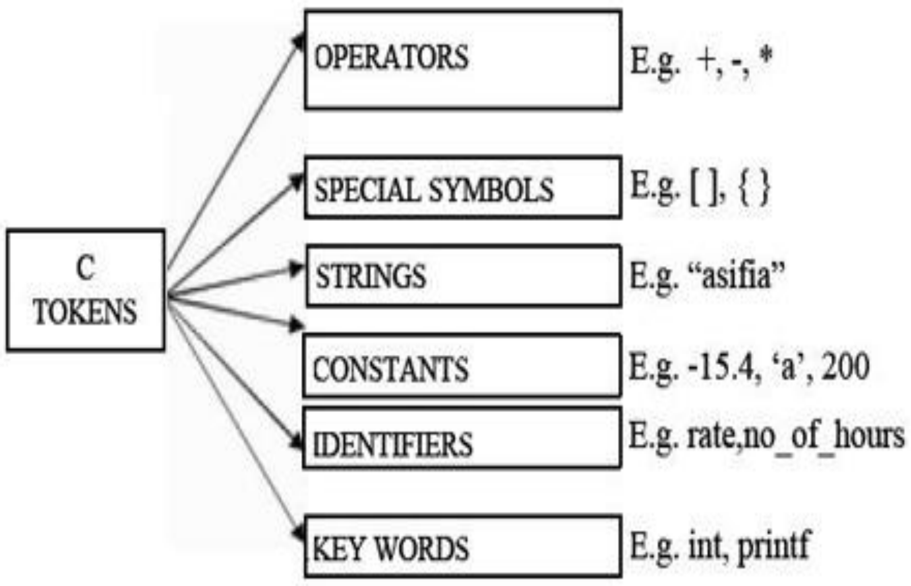
SUB PROGRAM OR FUNCTION SECTION : This has all the sub programs or the functions which our program needs.

SIMPLE ‘C’ PROGRAM:

```
/* simple program in c */
 #include<stdio.h> main()
{
printf(“welcome to c programming”);
} /* End of main */
```

1.8.3 C-TOKENS :

Tokens are individual words and punctuations marks in English language sentence. The smallest individual units are known as C tokens.



A C program can be divided into these tokens. A C program contains minimum 3 c tokens no matter what the size of the program is.

1.8.3.1 DATA TYPES:

To represent different types of data in C program we need different data types. A data type is essential to identify the storage representation and the type of operations that can be performed on that data. C supports four different classes of data types namely

- **Basic Data types**
- **Derives data types**
- **User defined data types**
- **Pointer data types**

BASIC DATA TYPES:

All arithmetic operations such as Addition, subtraction, etc are possible on basic data types.

E.g.: int a,b;
Char c;

The following table shows the Storage size and Range of basic data types:

TYPE	LENGTH	RANGE
Unsigned char	8 bits	0 to 255
Char	8 bits	-128 to 127
Short int	16 bits	-32768 to 32767
Unsigned int	32 bits	0 to 4,294,967,295
Int	32 bits	-2,147,483,648 to 2,147,483,648
Unsigned long	32 bits	0 to 4,294,967,295
Enum	16 bits	-2,147,483,648 to 2,147,483,648
Long	32 bits	-2,147,483,648 to 2,147,483,648
Float	32 bits	3.4*10E-38 to 3.4*10E38
Double	64 bits	1.7*10E-308 to 1.7*10E308
Long double	80 bits	3.4*10E-4932 to 1.1*10E4932

DERIVED DATA TYPES:

Derived data types are used in ‘C’ to store a set of data values. Arrays and Structures are examples for derived data types.

Ex: int a[10];
Char name[20];

USER DEFINED DATATYPES:

C Provides a facility called typedef for creating new data type names defined by the user. For Example , the declaration :
typedef int Integer;

makes the name Integer a synonym of int.Now the type Integer can be used in declarations ,casts, etc, like,

Integer num1,num2;

Which will be treated by the C compiler as the declaration of num1,num2as int variables. “typedef” ia more useful with structures and pointers.

POINTER DATA TYPES:

Pointer data type is necessary to store the address of a variable.

1.9 VARIABLES:

A quantity that can vary during the execution of a program is known as a variable. To identify a quantity we name the variable for example if we are calculating a sum of two numbers we will name the variable that will hold the value of sum of two numbers as 'sum'.

IDENTIFIERS :

- Names of the variables and other program elements such as functions, array,etc,are known as identifiers.
- There are few rules that govern the way variable are named(identifiers).
- Identifiers can be named from the combination of A-Z, a-z, 0-9, _(Underscore).
- The first alphabet of the identifier should be either an alphabet or an underscore. digit are not allowed.
- It should not be a keyword. Eg: name,ptr,sum After naming a variable we need to declare it to compiler of what data type it is . The format of declaring a variable is

Data-type id1, id2,.....idn;

where data type could be float, int, char or any of the data id1, id2, id3 are the names of variable we use. In case of single variable no commas are required.

```
eg      float a, b, c;
          Int  e, f, grand total;
          char present_or_absent;
```

ASSIGNING VALUES :

When we name and declare variables we need to assign value to the variable. In some cases we assign value to the variable directly like

```
a=10;
```

in our program

In some cases we need to assign values to variable after the user has given input for that.
Eg: we ask user to enter any no and input it.

```
/* write a program to show assigning of values to variables */

#include<stdio.h>
main()
{
int a; float b;
printf("Enter any number\n"); b=190.5;
scanf("%d",&a); printf("user entered %d", a); printf("B's values is %f", b);
}
```

CONSTANTS :
A quantity that does not vary during the execution of a program is known as a constant supports two types of constants namely Numeric constants and character constants.
NUMERIC CONSTANTS:

- Example for an integer constant is 786,-127
- Long constant is written with a terminal 'l'or 'L',for example 1234567899L is a Long constant.
- Unsigned constants are written with a terminal 'u' or 'U',and the suffix 'ul' and 'UL' indicates unsigned long. for example 123456789u is a Unsigned constant and 1234567891ul is an unsigned long constant.
- The advantage of declaring an unsigned constant is to increase the range of storage.
- Floating point constants contain a decimal point or an exponent or both. For Eg : 123.4,1e-2,1.4E-4,etc.The suffixes f or F indicate a float constant while the absence of f or F indicate the double, l or L indicate long double.

CHARACTER CONSTANTS:

A character constant is written as one character with in single quotes such as 'a'. The value of a character constant is the numerical value of the character in the machines character set. certain character constants can be represented by escape sequences like '\n'. These sequences look like two characters but represent only one.
The following are the some of the examples of escape sequences:

Escape sequence	**Description**
\a	Alert
\b	Backspace
\f	Form feed
\n	New Line
\r	Carriage return
\t	Horizontal Tab
\v	Vertical Tab

String constants or string literal is a sequence of zero or more characters surrounded by a double quote. Example , “ I am a little boy”. quotes are not a part of the string.
To distinguish between a character constant and a string that contains a single character ex: ‘a’ is not same as “a”. ‘a’ is an integer used to produce the numeric value of letter a in the machine character set, while “a” is an array of characters containing one character and a ‘\0’ as a string in C is an array of characters terminated by NULL.
There is one another kind of constant i.e Enumeration constant , it is a list of constant integer values.

1.10 Syntax and Logical Errors in Compilation:

Syntax error: A **syntax error** is an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language.

For compiled languages, syntax errors are detected at compile-time. A program will not compile until all syntax errors are corrected.

For example, consider the statement,

1 int a,b:

The above statement will produce syntax error as the statement is terminated with : rather than ;

Logical Error: A **logic error** is a bug in a program that causes it to operate incorrectly, but not to terminate abnormally (or crash). A logic error produces unintended or undesired output or other behaviour, although it may not immediately be recognized as such.

Logic errors occur in both compiled and interpreted languages. Unlike a program with a syntax error, a program with a logic error is a valid program in the language, though it does not behave as intended.

For example,

```
a=5/0;
```

1.11 **Object and Executable code:**

The programming language code is called as Source code or object code and is obtained after compilation. Source code extension is .c file and object code extension is .obj and executable file is .exe

1.12 OPERATORS :

An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations. They form expressions.

C operators can be classified as

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment or Decrement operators
- Conditional operator
- Bit wise operators
- Special operators

1. ARITHMETIC OPERATORS: All basic arithmetic operators are present in C.

operator	**meaning**
+	add
-	subtract
*	multiplication
/	division
%	modulo division(remainder)

An arithmetic operation involving only real operands(or integer operands) is called real arithmetic(or integer arithmetic). If a combination of arithmetic and real is called mixed mode arithmetic.

2. RELATIONAL OPERATORS: We often compare two quantities and depending on their relation take certain decisions for that comparison we use relational operators.

operator	**meaning**
<	is less than
➢	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equal to
!=	is not equal to

3. LOGICAL OPERATORS: An expression of this kind which combines two or more relational expressions is termed as a logical expressions or a compound relational expression. The operators and truth values are

op-1	**op-2**	**op-1 && op-2 op-1 || op-2**	
non-zero	non-zero	1	1
non-zero	0	0	1
0	non-zero	0	1
0	0	0	0
op-1	!op-1		
non-zero	Zero		
zero	non-zero		

4. ASSIGNMENT OPERATORS : They are used to assign the result of an expression to a variable. The assignment operator is '='.

Syntax: op=exp is variable
op binary operator exp expression
op= short hand assignment operator

short hand assignment operators

use of simple assignment operators	use of short hand assignment operators
a=a+1	a+=1
a=a-1	a-=1
a=a%b	a%=b

5. INCREMENT AND DECREMENT OPERATORS:

and == are called increment and decrement operators used to add or subtract. Both are unary and as follows

Syntax: ++m or m++ --m or m--

The difference between ++m and m++ is

if m=5; y=++m then it is equal to m=5;m++;y=m; if m=5; y=m++ then it is equal to m=5;y=m;m++;

6. CONDITIONAL OPERATOR: A ternary operator pair "?:" is available in C to construct conditional expressions of the form

Syntax: exp1 ? exp2 : exp3;

It work as

if exp1 is true then exp2 else exp3

7. BIT WISE OPERATORS: C supports special operators known as bit wise operators for manipulation of data at bit level. They are not applied to float or double.

operator	**meaning**
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement

8. SPECIAL OPERATORS:

These operators which do not fit in any of the above classification are , (comma), sizeof, Pointer operators(& and *) and member selection operators (. and ->). The comma operator is used to link related expressions together. sizeof operator is used to know the sizeof operand.

```
/* programs to exhibit the use of operators  */

#include<stdio.h>
main()
{
int sum, mul, modu; float sub, divi;
int i,j; float l, m;
printf("Enter two integers "); scanf("%d%d",&i,&j); printf("Enter two real numbers");
scanf("%f%f",&l,&m);
sum=i+j;
mul=i*j;
modu=i%j; sub=l-m; divi=l/m;
printf("sum is %d", sum); printf("mul is %d", mul); printf("Remainder is %d", modu);
printf("subtraction of float is %f", sub); printf("division of float is %f", divi);
}

/* program to implement relational and logical  */
#include<stdio.h>
main()
{
int i, j, k;
printf("Enter any three numbers "); scanf("%d%d%d", &i, &j, &k); if((i<j)&&(j<k))
printf("k is largest"); else if i<j || j>k
{
if i<j && j >k
printf("j is largest");
else
printf("j is not largest of all");
}
}
```

```
/* program to implement increment and decrement operators */

#include<stdio.h>
main()
{
int i;
printf("Enter a number"); scanf("%d", &i);
i++;
printf("after incrementing %d ", i); i--;
printf("after decrement %d", i); }
/* program using ternary operator and assignment */ #include<stdio.h>

main()
{
int i,j,large;
printf("Enter two numbers "); scanf("%d%d",&i,&j); large=(i>j)?i:j;
printf("largest of two is %d",large);
}
```

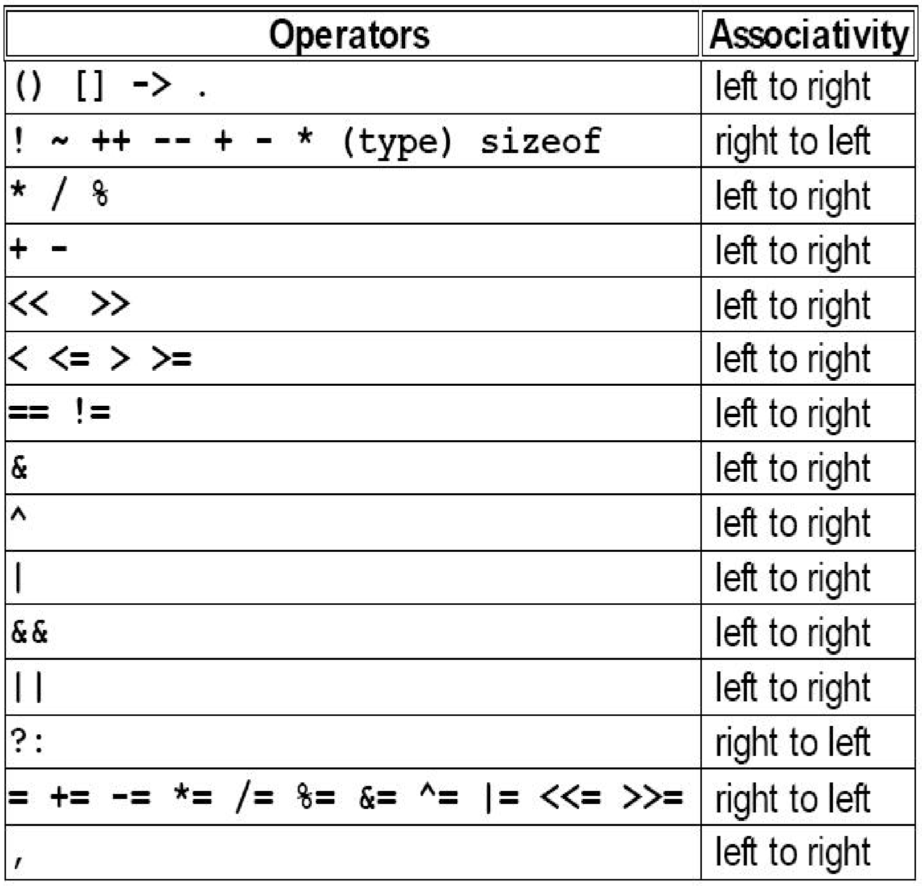
1.13 EXPRESSIONS AND PRECEDENCE:

An expression is a sequence of operands and operators that reduces to a single value. Expression can be simple or complex. An **operator** is a syntactical token that requires an action be taken. An **operand** is an object on which an operation is performed.

A simple expression contains only one operator. E.g: 2 + 3 is a simple expression whose value is 5.A complex expression contains more that one operator. E.g: 2 + 3 * 2. To evaluate a complex expression we reduce it to a series of simple expressions. In this first we will evaluate the simple expression 3 * 2 (6)and then the expression 2 + 6,giving a result of 8.

The order in which the operators in a complex expression are evaluated is determined by a set of priorities known as **precedence,** the higher the precedence ,the earlier the expression containing the operator is evaluated. If two operators with the same precedence occur in a complex expression ,another attribute of an operator ,its associativity ,takes control. **Associativity** is the parsing direction used to evaluate the expression. It can be either left-to-right or right-to-left. When two operators with the same precedence occur in an expression and their associativity is left-to-right ,the left operator is evaluated first. For example ,in the expression 3*4/6 ,there are two operators multiplication and division ,with the same precedence and left-to-right associativity. Therefore the multiplication is evaluated before the division.

The following table shows the precedence and associativity of operators:



Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

1.14 EXPRESSION EVALUATION:

ARITHMETIC EXPRESSIONS:

It is a combination of variables, constants and operators arranged according to the syntax of C language.

Some examples A * B – C
(M + N) * (X + Y)

Evaluation of expressions: Expressions are evaluated using an assignment statement of the form

Syntax:

Variable = expression

Eg:

x = x*y + z-3 *(z *y)

Precedence of arithmetic expressions is used to evaluate a expression to provide un ambiguous way of solving an expression. The highest precedence operator is evaluated then next highest precedence operator until no operator is present.

The precedence or priorities are as follows High * / %

Low + -

An **expression is evaluated** in left to right and value is assigned to variable in left side of assignment operator.

```
/* program to demonstrate evaluation of expressions */ #include<stdio.h>
main()
{
float a,b,c,x,y,z; a=9;b=23;c=3; x=a-b/3+c*2-1; y=a-b/(3+c)*(2-1); z=a-(b/(3+c)*2)-1;
printf("values of x,y,z are %d%d%d",x,y,z);
}
```

1.15 STORAGE CLASSES:

Storage Classes:- Not only data type is required to declare a variable but its storage class also has to be mentioned. **(or)**

The variables declared in C programs are totally different from other languages. We can use the same variable names in the C program in separate blocks. When we declare a variable it is available only to specific part or block of the program. Remaining block or other function cannot get access to the variable.

The area or block of the C program from where the variable can be accessed is known as the **scope of variable.** The area or scope of the variable depends on its **storage class** i.e. where and how it is declared. There are four scope variables in C.

1. Function
2. File
3. Block
4. Function prototype

A storage class of variable tells us four things:

(i) Where the variable would be stores.

(ii) The Scope of the variable i.e., in which region of the program the value of variable is actually available for us active.

(iii) Life of the variable i.e., how long the variable i.e., how long the variable would be active in the program(longevity or alive).

(iv) The initial value of the variable if it is not initialized.

Any variable declared in C can have any one of the four storage classes:

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

The variables may also be broadly categorized, depending on the place of their declaration, as

1. Internal variables (*local)*
2. External variables (*global)*

1*) Automatic variables* are defined inside a function. A variable declared inside a function without storage class name, by default is an auto variable.

The features of automatic variables are:-

(i)	**Storage**	**:**Memory
(ii)	**Initial value**	**:**Garbage (or) unpredictable
(iii)	**Scope**	**:**Within the function
iv)	**Life time**	**:**Till the control remains in the function.

These variables are created when the function is called and destroyed automatically when the function is exited.

Automatic variables are local to the function in which they are declared. These values cannot be accessed by any other function. The keyword used is „**auto**‟.

Eg:

```
void main ( )

{

auto int a =100;

clrscr ( );

{

auto int a = 300;

{

auto int a = 500;

printf ("\n\n\t a=%d",a);

}

printf ("\n\n\t a=%d",a);

}

printf ("\n\n\t a=%d", a);

getch ( );

}
```

O/p: a = 500 a = 300 a = 100

Eg 4:

```
void show ( );

void main ( )

{

int I;

clrscr ( );

for (i=1; i<5; i++);

show ( )

getch ( );

}
```

```
void show ( )
{
int a= 10;
printf (“\n\n\t a=%d”,a);
a++;
}
```

2)External variables are also known as global variables. These variables are declared outside the function and the values of these variables are available to all the functions of the program.

Unlike Local Variables, Global Variables can be accessed by any function in the program.

If same name is given to both the global and local variables priority is given to the local variable. The keyword “ extern” is used to declare these variables.

The features of external variables are:

(i)	**Storage**	**:**	memory
(ii)	**Initial value**	**:**	zero
(iii)	**Scope**	**:**	Global
(iv)	**Life time**	**:**	Till the program comes to an end.

Example Program:

```
#include<stdio.h>

#include<conio.h>
int a=20;
void main( )
{
clrscr( );
fun1( );
fun2( );
fun3( );
printf(“\n In main function a=%d”, a);
}
fun1( )
{
printf("\n In fun1 a = %d", a);
}
fun2( )
{
int a = 10;
printf("\n In fun2 a = %d",a);
}
fun3( )
{
printf("\n In fun3 a = %d", a);
}
```

Explanation: In this program local variable and global variables are declared with the same name in fun2( ). In this case, when fun2( ) is called, the local variable „a‟ of fun2( ) hides the global variable „a‟.

Note:

We can declare external variables by using **extern** key word inside the function body. **Ex:** extern int a;

Eg 1:

```
extern int a = 100; // int a;

void main ( )

{
clrscr ( );
printf ("\n\n a=%d", a);
getch ( );
}
```

O/p: a = 100 //a = 0

3) ***Static variables*** may be of Local (or) global depending upon where it is declared. If it is declared outside the function, it is static global otherwise if it declared inside a function block, it is static local.

A static variable is initialized only once and can never be re-initialized. The value of static variable persists at each call and last change made in the variable remains throughout the program execution.

The features of a static variable are:-

(i)	**Storage**	:memory
(ii)	**Initial value**	:zero
(iii)	**Scope**	**:**Local to the block in which variable is defined.
(iv)	**Life time**	:persists till the end of program execution.

Example Programs:-

Eg 1:

```
void main( )

{

incr( );

incr( );

incr( );

}

void incr( )

{

static int x;

x=x+1;

printf("%d", x);

}
```

The keyword used to declare these variables is "**static**".

4*)* Register Variables*:* Instead of storing in memory, variables can also be stored in register of CPU. The advantage of storing in registers is register access is faster than memory access, so, generally frequently accessed variables are kept in registers for faster execution of the program.

Syntax : register int count;

The keyword register tells the compiler that the variable list followed by it is kept on the CPU registers. If the CPU fails to keep the variables in CPU registers, in that case the variables are assured as auto and stored in the memory.

Note:

1. CPU registers are limited in number. So, we cannot declare more variables as register variables. Compiler automatically converts the register variables into non-register variables once the limit is reached.
2. We cannot use register class for all types of variables. The CPU registers in microcomputer are 16 bit registers. The data types *float* and *double* needs space of more than 16 bits. If we define any variable of these types with register class, no errors will be shown. The compiler treats them as auto variables.

The features of register variables are:-

(i)	**Storage**	**:**Registers
(ii)	**Initial value**	**:**Garbage
(iii)	**Scope**	**:**Local
(iv)	**Life time**	**:**until the control remains in that function block.

Example Programs:-

Eg 1:

```
void main( )

{

register int i;

for (i=1; i<=5; i++)

printf (“ %d/t”, i);
```

}

The keyword used to declare these variables is “**register**”.

1.16 TYPE CONVERSION:

In an expression that involves two different data types ,such as multiplying an integer and a floating point number to perform these evaluations ,one of the types must be converted.
We have two types of conversions

1. Implicit Type Conversion
2. Explicit Type Conversion

IMPLICIT TYPE CONVERSION :
When the types of the two operands in a binary expression are different automatically converts one type to another .This is known as implicit type conversion .

EXPLICIT TYPE CONVERSION :
Explicit type conversion uses the unary cast operator ,to convert data from one type to another. To cast data from one type to another ,we specify the new type in parentheses before the value we want converted.

For example ,to convert an integer ,a , to a float, we code the expression like (float) a

1.17 Conditional Branching and Loops:

1.17.1 Writing and evaluation of conditionals

STATEMENTS AND BLOCKS :

A statement causes an action to be performed by the program. It translates directly in to one or more executable computer instructions.

STATEMENT TYPES:

NULL STATEMENT :

The null statement is just a semicolon (the terminator).

Eg:
//null statement

Although they do not arise often, there are syntactical situations where we must have a statement but no action is required .In these situations we use the null statement.

EXPRESSION STATEMENT :

An expression is turned in to a statement by placing a semicolon(;)after it.

Syntax:
expression;   //expression statement

Eg: a=2;

RETURN STATEMENT :

A return statement terminates a function. All functions ,including main, must have a return statement. Where there is no return statement at the end of the function ,the system inserts one with a void return value.

Syntax: return expression;   //return statement
The return statement can return a value to the calling function. In case of main ,it returns a value to the operating system rather than to another function. A return value of zero tells the operating system that the program executed successfully.

COMPOUND STATEMENTS:

A compound statement is a unit of code consisting of zero or more statements .It is also known as a **block**. The compound statement allows a group of statements to become one single entity. A compound statement consists of an opening brace, an optional declaration and definition section ,and an optional statement section ,followed by a closing brace.

Eg:
```
{
//Local Declarations int x;
int y; int z;

//Statements
x=1;
y=2;
…
} //End BlocK
```

1.18 IF AND SWITCH STATEMENTS :

We have a number of situations where we may have to change the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

The if statement is a two way decision statement and is used in conjunction with an expression. It takes the following form

If(test expression)

If the test expression is true then the statement block after if is executed otherwise it is not executed

Syntax:

```
if (test expression)
{
        statement block;


}
statement–x ;
```

only statement–x is executed.

```
/* program for if */
#include<stdio.h>
main()

{

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

The if –else statement:

If your have another set of statement to be executed if condition is false then if-else is used

Syntax: if (test expression)

```
{
        statement block1;
}
else
{
        statement block2;
}
statement –x ;
```

Example: /* program for if-else */

```
#include<stdio.h>
main()
{
        int a,b;
        printf("Enter two numbers");
        scanf("%d%d",&a,&b):
        if a>b
                printf(" a is greater")
        else
                printf("b is greater");
}
```

Nesting of if..else statement :

If more than one if else statement exists in the program then it is said as Nesting of statement

Syntax:

```
 if(text cond1)
{

if (test expression2
        {
                statement block1;
        }
        else
        {
                statement block 2;
        }
}
else
{
        statement block2;
}
statement-x ;
```

if else ladder

```
        if(condition1)
                statement1; else
                   if(condition2)
               statement 2; else
                   if(condition3)
                     statement n;
        else
                default statement.
        statement-x;
```

The nesting of if-else depends upon the conditions with which we have to deal.

THE SWITCH STATEMENT: If for suppose we have more than one valid choices to choose from then we can use switch statement in place of if statements.

Syntax: switch(expression)

```
        {.
                case value-1
                                block- 1
                                break;

                case value-2
                                block- 2
                                break;


                        ----------------
                default:
                                default block;
                                break; }
        statement–x
```

In case of
Syntax:

```
        if(cond1)
                {
                statement-1
                }
                if (cond2)
                {
                        statement 2
                }
```

```
/* program to implement switch */
#include<stdio.h>
main()
{
         int marks,index;
        char grade[10];
        printf("Enter your marks");
        scanf("%d",&marks);
        index=marks/10; switch(index)

        {
                case 10 :
                case 9: case
                8: case 7:

                case 6: grade="first";
                        break;
                case 5 : grade="second";
                        break;
                case 4 : grade="third";
                        break;

                default : grade ="fail";
                        break;
        }
        printf("%s",grade);
}
```

LOOPING :

Some times we require a set of statements to be executed repeatedly until a\condition is met.

We have two types of looping structures. One in which condition is tested before entering the statement block called entry control.

The other in which condition is checked at exit called exit controlled loop.

WHILE STATEMENT :

It helps to execute the statements only if the condition is true.

Syntax:

```
        While(test condition)
        {
                body of the loop
        }
```

It is an entry controlled loop. The condition is evaluated and if it is true then body of loop is executed. After execution of body the condition is once again evaluated and if is true body is executed once again. This goes on until test condition becomes false.

```
Example: /* program for
while */ #include<stdio.h>
main()
{
        int count,n;
        float x,y;
        printf("Enter the values of x and n");
        scanf("%f%d",&x,&n);
        y=1.0;
        count=1;
        while(count<=n)
        {
                y=y*x;
                count++;
        }
        printf("x=%f; n=%d; x to power n = %f",x,n,y);
}
```

DO WHILE STATEMENT:

The while loop does not allow body to be executed if test condition is false. The do while is an exit controlled loop and its body is executed at least once.

Syntax:

```
do
        {
                body }while(test
        condition);
```

Example: /* printing multiplication table */

```
#include<stdio.h>
#define COL 10
#define ROW 12
main()
```

```
{
        int row,col,y;
        row=1;

        do
        {
                col=1; do
                {
                        y=row*col;
                        printf("%d",y);
                        col=col+1;

                }while(col<=COL);
                printf("\n");
                row=row+1;
        }while(row<=ROW);
}
```

THE FOR LOOP:

It is also an entry control loop that provides a more concise structure

Syntax:

```
        for(initialization; test control; increment)
        {
                        body of loop
        }
```

Example: /* program of for loop */ #include<stdio.h>

```
main()

{
        long int p; int
        n; double q;
        printf("2 to power n "); p=1;
        for(n=0;n<21;++n)

        {
                if(n==0)
                        p=1;
                else
                        p=p*2;
                q=1.0/(double)p;
                printf("%101d%10d",p,n);
        }
}
```

BREAK STATEMENT:

This is a simple statement. It only makes sense if it occurs in the body of a switch, do, while or for statement. When it is executed the control of flow jumps to the statement immediately following the body of the statement containing the break. Its use is widespread in switch statements, where it is more or less essential to get the control .

The use of the break within loops is of dubious legitimacy. It has its moments, but is really only justifiable when exceptional circumstances have happened and the loop has to be abandoned. It would be nice if more than one loop could be abandoned with a single break but that isn't how it works. Here is an example.

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

        for(i = 0; i < 10000; i++){
         if(getchar() == 's') break;

              printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

It reads a single character from the program's input before printing the next in a sequence of numbers. If an ‘s’ is typed, the break causes an exit from the loop.
If you want to exit from more than one level of loop, the break is the wrong thing to use.

CONTINUE STATEMENT:

This statement has only a limited number of uses. The rules for its use are the same as for break, with the exception that it doesn't apply to switch statements. Executing a continue starts the next iteration of the smallest enclosing do, while or for statement immediately. The use of continue is largely restricted to the top of loops, where a decision has to be made whether or not to execute the rest of the body of the loop. In this example it ensures that division by zero (which gives undefined behaviour) doesn't happen.

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

      for(i = -10; i < 10; i++){ if(i
              == 0)
                      continue;
              printf("%f\n", 15.0/i); /*
               * Lots of other statements .....
               */
      }
      exit(EXIT_SUCCESS);
}
```

The continue can be used in other parts of a loop, too, where it may occasionally help to simplify the logic of the code and improve readability. continue has no special meaning to a switch statement, where break does have. Inside a switch, continue is only valid if there is a loop that encloses the switch, in which case the next iteration of the loop will be started.

There is an important difference between loops written with while and for. In a while, a continue will go immediately to the test of the controlling expression. The same thing in a for will do two things: first the update expression is evaluated, then the controlling expression is evaluated.

GOTO AND LABELS:

In C, it is used to escape from multiple nested loops, or to go to an error handling exit at the end of a function. You will need a *label* when you use a goto; this example shows both.

```
goto L1;
/* whatever you like here */
L1: /* anything else */
```

A label is an identifier followed by a colon. Labels have their own 'name space' so they can't clash with the names of variables or functions. The name space only exists for the function containing the label, so label names can be re-used in different functions. The label can be used before it is declared, too, simply by mentioning it in a goto statement.

Labels must be part of a full statement, even if it's an empty one. This usually only matters when you're trying to put a label at the end of a compound statement—like this.

```
label_at_end: ; /* empty statement */
}
```

The goto works in an obvious way, jumping to the labelled statements. Because the name of the label is only visible inside its own function, you can't jump from one function to another one.

It's hard to give rigid rules about the use of gotos but, as with the do, continue and the break (except in switch statements), over-use should be avoided. More than one goto every 3–5 functions is a symptom that should be viewed with deep suspicion.

1.19 & 1.20 FORMATTED INPUT AND OUTPUT STATEMENTS:

The simplest of input operator is getchar to read a single character from the input device.

Syntax: varname=getchar(); //you need to declare varname.

The simplest of output operator is putchar to output a single character on the output device.

putchar(varname)

The getchar() is used only for one input and is not formatted. Formatted input refers to an input data that has been arranged in a particular format, for that we have scanf.
scanf("control string", arg1, arg2,...argn);
Control string specifies field format in which data is to be entered.
arg1, arg2... argn specifies address of location or variable where data is stored.

eg scanf("%d%d",&a,&b);

%d	used for integers
%f	floats
%l	long
%c	character

for formatted output you use printf

printf("control string", arg1, arg2,...argn);

```
/* program to exhibit i/o */
 #include<stdio.h>
main()
{
int a,b; float c;
printf("Enter any number"); a=getchar();
printf("the char is "); putchar(a);
printf("Exhibiting the use of scanf");
printf("Enter three numbers");
scanf("%d%d%f",&a,&b,&c);
printf("%d%d%f",a,b,c);
}
```

1.20 INTRODUCTION TO STDIN, STDOUT AND STDERR

Standard Input (stdin) :

Standard input is stream data (often text) going into a program. The program requests data transfers by use of the read operation. Not all programs require stream input. For example, the dir and ls programs (which display file names contained in a directory) may take command-line arguments, but perform their operations without any stream data input.

Unless redirected, standard input is inherited from the parent process. In the case of an interactive shell, that is usually associated with the keyboard.

The file descriptor for standard input is 0 (zero); <stdio.h> variable is FILE* stdin;

Standard Output(stdout):

Standard output is the stream where a program writes its output data. The program requests data transfer with the write operation. Not all programs generate output. For example, the file rename command (variously called mv, move, or ren) is silent on success.
Unless redirected, standard output is inherited from the parent process. In the case of an interactive shell, that is usually the text terminal which initiated the program.

The file descriptor for standard output is 1 (one); the POSIX <*unistd.h*> definition is `STDOUT_FILENO`; the corresponding C <*stdio.h*> variable is `FILE* stdout`

1.21 COMMAND LINE ARGUMENTS IN C:

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

Syntax:

```
int main() { /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

Syntax:

```
int main(int argc, char *argv[]) { /* ... */ }
or
int main(int argc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

TUTORIAL QUESTIONS

<u>This Tutorial corresponds to Unit No: 1</u>

Find the output of the following Program:

1.
```
#include <stdio.h>
  int main(void) {
    int i, j, k;
    i = -1;
    j = 1;
    if(i)
       j--;
    if(j)
       i++;
    k = i * j;
    printf("%d",k);
    return 0;
  }
```

Ans:______________________

2.
```
#include <stdio.h>
  int main(void) {
    float x,y;
    int i,j;
    x = 1.5; y = 2.0;
    i = 2; j = 4;
    x = x * y + (float)i / j;
    printf("%f",x);
    return 0;
  }
```

Ans:______________________

3.
```
#include <stdio.h>
  int main(void) {
    int i = 7, j = i - i;
    while(i) {
       i /= 2;
       j++;
    }
    printf("%d",j);
    return 0;
  }
```

Ans:______________________

4.
```
#include <stdio.h>
    int main(void) {
        int i, j = 1;
        for(i = 11; i > 0; i /= 3)
            j++;
        printf("%d",j);
        return 0;
    }
```

Ans:______________________

5.
```
#include <stdio.h>
    int main(void) {
        int i = 1, j = -2, k;
        k = (i >= 0) || (j >= 00) && (i <= 0) || (j <= 0);
        printf("%d",k);
        return 0;
    }
```

Ans:______________________

ASSIGNMENT QUESTIONS

SET-1:

1. Design an algorithm and flow chart to find maximum among three numbers.
2. Implement a C program to find factors of given number.

SET-2:

1. Design an algorithm and flow chart to check given number is positive or negative or zero
2. Implement a C program to print multiplication table for 1 to n numbers.

SET-3:

1. Implement a C program to check given number is Armstrong or not.
2. Design an algorithm and flow chart to convert Celsius to Fahrenheit and Fahrenheit to Celsius.

SET-4:

1. Implement a C program to check given number is perfect number or not.
2. Design a structure how to create and run C program.

SET-5:

1. Implement a C program to reverse a given number.
2. Design program Development steps.

SET-6:

1. Implement a C program to convert decimal number into binary.
2. Discuss about data types.
3. Implement a C program to add two numbers using command line argument.

IMPORTANT QUESTIONS UNIT – I

1. What is a flowchart? Explain the different symbols used in a flowchart.

2. (a) Define an Algorithm?

(b) What is the use of Flowchart?

(c) What are the different steps followed in the program development?

. 3. Explain the different computing environments?

4. Explain the steps for the creation and running c program?
5. Draw a flow chart to find the biggest among three numbers.
6. Write the various steps involved in executing a C program and illustrate with the help of flow chart?
7. What is the difference between break and continue statements ? Explain with examples.
8. What is the purpose of goto statement? How is the associated target statement identified?
9. (a) What are constants?

   B.Name the different data types that C supports and explain them in detail.
10. What is meant by looping? Describe any two different forms of looping with Examples.

(b) Write a program to print the following outputs using for loop.

i)
```
1
2  2
3  3 3
4  4 4 4
```

ii)
```
            1
      2           2
```

```
        3         3           3
    4         4          4        4
```

10 What are the logical operators used in C and illustrate with examples?

11. Whar is the purpose of switch statement ? How does this statement differ from the other statements?

12. State and explain various identifiers in C program. And also discuss about operator precedence in expression evaluation with a suitable example.

13. Explain with a sample program about while, for, do-while and switch statements in C programming

14. Explain the difference between break, goto and continue statements with an example

15. Describe the for loop statement in 'C'.

16. Explain the difference between while and do-while statements with suitable examples.

17. Write a C program to print digits in reverse order for a given number.

18. Describe type casting with an example.

19. Explain the logical operators with suitable examples.

20. Write C program to print prime numbers in a given series of numbers. For example: numbers from 1 to 100.

21. What is a flowchart? Discuss various symbols used in flowchart. Illustrate with an example.

22. Explain the steps in program development.

23. What is meant by assembly language? Give example. Differentiate between pre test and post test loops.

24. What is the need of type conversion? Discuss type casting. b) List the demerits of go to statement. c) Why is switch statement known as multi way selection?

25. Explain switch statement. Explain its usage with a sample C-program. What are the storage classes in C? Explain their usage with a sample C-program.

26. Explain scope of a variable with an example. Write brief notes on computer languages

OBJECTIVE QUESTIONS

1. The C language consist of ____ number of keywords.

2. What will be sum of the binary numbers 1111 and 11001………………

3. Which one of the following is known as the language of the computer [ ]

(A) Programming language

(B) Machine language

(C) High level language

(D) Assembly level language

4. Which of the following is syntactically correct [ ]

(A) for(;); (B) for(); (C) for(,); (D) for(;;);

5. which one of the following is not a translator program [ ]

(A) Assembler (B) Interpreter (C) Linker (D) Compiler

6. . What will be the ASCII Octal value of A [ ]

(A) 100 (B) 101 (C) 110 (D) 111

7. a<<1 IS EQUAL TO ______________

8. The process of repeating a group of statements in an algorithm is known as___________

9. Extend the term CPU ________________________________

10 Monitor, keyboard, mouse and printers are _________devices

14. C was developed by ____________________

15. _________is used to compile your c program

16. Short Integer size is ____ bytes

17. The while loop repeats a statement until the test at the top proves _______

18. The __________statement transfers control to a statement within its body

19. The _______is a unconditional branching statement used to transfer control of the program from one statement to another

20. ANSI stands for _________________

UNIT-II

ARRAYS

2.1 Arrays:

An array is a group of related data items that share a common name.
Ex:- Students
The complete set of students is represented using an array name students. A particular value is indicated by writing a number called index number or subscript in brackets after array name. The complete set of value is referred to as an array, the individual values are called elements.

2.2 ONE – DIMENSIONAL ARRAYS :

A list of items can be given one variable index is called single subscripted variable or a one-dimensional array.
Declaration of One - Dimensional Arrays :

Syntax: Type variable _name [sizes];

Type – data type of all elements
Ex: int, float etc.,
Variable – name – is an identifier
Size – is the maximum no of elements that can be stored.

Ex:- float avg[50]

This array is of type float. Its name is avg. and it can contains 50 elements only. The range starting from 0 – 49 elements.

The subscript value starts from 0. If we want 5 elements the declaration will be

int number[5];

The elements will be number[0], number[1], number[2], number[3], number[4] There will not be number[5]

2.3 Initialization of Arrays :

Initialization of elements of arrays can be done in same way as ordinary variables are done when they are declared.

Type array name[size] = {List of Value};

Ex:- int number[3]={0,0,0};
If the number of values in the list is less than number of elements then only that elements will be initialized. The remaining elements will be set to zero automatically.

Ex:- float total[5]= {0.0,15.75,-10};

The size may be omitted. In such cases, Compiler allocates enough space for all initialized elements.

```
int counter[ ]= {1,1,1,1};
```

```
/* Program Showing one dimensional array */

#include<stdio.h>
main(){
int i;
printf("Enter 10  numbers\n");
for(i=0;i<10;i++)
{
scanf("%d",&a[i]);
}
printf("the entered numbers are\n");
for(i=0;i<10;i++)
{
printf("%d\t",a[i]);
}
getch();
}
```

```
/* Program to add one dimensional array */
#include<stdio.h>
main(){
int i,sum=0;
printf("Enter 10  numbers\n");
for(i=0;i<10;i++)
{
scanf("%d",&a[i]);
sum=sum+a[i];
}
printf("the sum of array elements are \n");
for(i=0;i<10;i++)
{
printf("sum=%d\t",sum);
}
getch();
}
```

TWO – DIMENSIONAL ARRAYS:

To store tables we need two dimensional arrays. Each table consists of rows and columns. Two dimensional arrays are declare as

Syntax: type array name [row-size][col-size];

```
/* Write a program Showing 2-DIMENSIONAL ARRAY */

/* SHOWING MULTIPLICATION TABLE */

#include<stdio.h>

#include<math.h>
#define ROWS 5
#define COLS 5
main()

{

        int row,cols,prod[ROWS][COLS]; int
        i,j;


        printf("Multiplication table");
        for(j=1;j< =COLS;j++)

                printf("%d",j);

        for(i=0;i<ROWS;i++)

        {

                row = i+1;
                printf("%2d|",row);


                for(j=1;j < = COLS;j++)

                {

                        COLS=j;

                        prod[i][j]= row * cols;
                        printf("%4d",prod[i][j]);


                }

        }

}
```

INITIALIZING TWO DIMENSIONAL ARRAYS:

They can be initialized by following their declaration with a list of initial values enclosed in braces.

Ex:- int table[2][3] = {0,0,0,1,1,1};

Initializes the elements of first row to zero and second row to one. The initialization is done by row by row. The above statement can be written as

```
int table[2][3] = {{0,0,0},{1,1,1}};
```

When all elements are to be initialized to zero, following short-cut method may be used.

```
int m[3][5] = {{0},{0},{0}};
```

MULTI-DIMENSIONAL ARRAYS:

C allows arrays of three or more dimensions. The exact limit is determined by Compiler.

Syntax: type array- names[s1][s2][s3] - - - - - [sn];
where si is size of dimension.

Ex:- int Survey[3][5][2];

```
#include <stdio.h>
void printArray( const int a[][ 3 ] );
int main()
{
  int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
  int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
  printf( "Values in array1 by row are:\n" );
  printArray( array1 );
  printf( "Values in array2 by row are:\n" );
  printArray( array2 );
  printf( "Values in array3 by row are:\n);
  printArray( array3 );
  return 0;
}
void printArray( const int a[][ 3 ] )
{
  int i; int
  j;
  for ( i = 0; i <= 1; i++ ) { for (
    j = 0; j <= 2; j++ ) {

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

}
```

APPLICATIONS OF ARRAYS:

1. Frequency arrays with their graphical representations.
2. Random number permutations.

FREQUENCY ARRAYS:

Two common statistical applications that use arrays are frequency distributions and histograms. A frequency array shows the number of elements with an identical value found in a series of numbers.

For example ,suppose we have taken a sample of 100 values between 0 and 19.We want to know how many of the values are 0,how many are 1,how many are 2,and so forth up through 19.

We can read these numbers into an array called numbers. Then we create an array of 20 elements that will show the frequency of each number in the series.

One way to do it is to assign the value from the data array to an index and then use the index to access the frequency array.

```
F=numbers[i];
Frequency[F]++;
```

Since an index is an expression ,however ,we can simply use the value from our data array to index us in to the frequency array .The value of numbers[i] is determined first ,and then that value is used to index in to frequency.

```
Frequency[numbers[i]]++
```

HISTOGRAMS:

A histogram is a pictorial representation of a frequency array .Instead of printing the values of the elements to show the frequency of each number, we print a histogram in the form of a bar chart. For example ,the following figure is a histogram for a set of numbers in the range 0 …19.In this example, asterisks (*) are used to build the bar. Each asterisk represents one occurrence of the data value.

```
0     0
1     4 * * * *  ————————(four 1s)
2     7 * * * * * * *
3     7 * * * * * * *————————(seven 3s)
.
.
.
18    2 * *
19    0 ————————————————(zero 19s)
```

RANDOM NUMBER PERMUTATIONS:

A random number permutation is a set of random numbers in which no numbers are repeated. For example, given a random number permutation of 10 numbers, the values from 0 to 9 would all be included with no duplicates.

Generating random numbers:

To generate a random integral in a range x to y, we must first scale the number and then ,if x is greater than 0 ,shift the number within the range. We scale the number using the modulus operator.

Ex: To produce a random number in the range 0 …50,we simply scale the random number and scaling factor must be one greater than the highest number needed.

rand ( ) %51

modulus works well when our range starts at 0.for example ,suppose we want a random number between 10 and 20.for this we will use one formula

range = (20 - 10) +1;

randno = rand ( ) % range + 10;

To generate a permutation ,we need to eliminate the duplicates. Using histogram concept we can solve the problem most efficiently by using two arrays. The first array contains the random numbers. The second array contains a logical value that indicates whether or not the number represented by its index has been placed in the random number array.

Only the first five random numbers have been placed in the permutation. For each random number in the random number array ,its corresponding location in the have-random array is set to 1. Those locations representing numbers that have not yet been generated are still set to 0.

2.4 STRINGS

The group of characters, digits, and symbols enclosed with in double quotation marks are called as strings.

Ex: "MRECW"

"1234"

Every string terminates with "**\0**" (**NULL**) character. The decimal equivalent value of null is zero. We use strings generally to manipulate text such as words and sentences. The common operations that we can perform on strings are:

1. Reading and writing strings.
2. Combining strings together.
3. Coping one string to another.
4. Extracting a portion of a string.
5. Comparing string for equality.

2.5 Declaration and Initialization:

Every string is always declared as character array.

Syntax: char string_name [size];

Ex: char name[20];

We can initialize string as follows:

1. Char name[ ]={"H‟,‟E‟,‟L‟,‟L‟,‟O‟,‟\0‟};

The last character of string is always "\0‟ (NULL). But it is not necessary to write "\0‟ character at the end of the string. The compiler automatically puts "\0" at the end of the string / character array.

The characters or elements of the string are stored in contiguous memory locations.

Memory map for above example:

H	E	L	L	O	\0
2000	2001	2002	2003	2004	2005

2. Char name [ ] = "HELLO".

In this type also C compiler automatically inserts NULL character at the end of the string.

3. Char name[9]={"H‟},{‟E‟},{‟L‟},{‟L‟},{‟O‟}};

In the above three formats we can initialize character arrays. We can also initialize character array by using standard input functions like scanf( ), gets ( ) etc and we can display strings by using puts ( ), printf ( ) functions.

The size of the character array: ***The argument / size of the character array or string = Number of characters in the string + NULL character.***

Note:

1. If NULL character is not taken in account then the successive string followed by the first string will be displayed.
2. If we declare size of a string equal to the number of characters. i.e., without taking the NULL character in to account and we print that will display some garbage value followed by the string.

READING STRINGS FROM TERMINAL:

1. Using scanf ( ) function

To read strings from terminal we can use scanf ( ) function with "%s" format specifier.

Ex: char name [20];

scanf("%s", name);

Limitation: If we use scanf ( ) function for reading string in runtime it terminates its input on the first white space it finds. The scanf ( ) function automatically append NULL character at the end of the input string, so we have to declare a (size of) character array which can accommodate input string and NULL character.

(**White space:** blanks, tab space, carriage returns, new lines etc.)

2. By using gets ( ) function

By using gets ( ) function we can read a text which consists of more than one word which i not possible with scanf ( ) function. This function is available in **<stdio.h>** header file.

Ex: gets (string);

Here string is a character array.

Ex: gets ("GOOD MORNING"):

We can pass string in double quotes as an argument.

3. By using getchar ( ) function

We know that getchar ( ) function is used to read single character from the terminal. By using this function repeatedly we can read successive single characters from the input and place them into a character array (string). The reading is terminated when the user enters new line character (\n) or press enter key. After reading the new line character this function automatically appends NULL character at the end of the string in the place of new line character.

Ex: char ch[20];
int i=0;
while ((ch[i]=getchar ( ))!= „\n\)
{
i = i+1;
}

WRITING STRINGS TO SCREEN

1. BY USING PRINTF ( ) FUNCTION

Using printf ( ) function we can display the string on output screen with the help of format specifier „%s".

Ex: printf("%s",name);
We can use different format specifications to display strings in various formats according to requirements.

2. By using puts ( ) and putchar ( ) functions

Using puts ( ) function we can display the string without using any format specifier. This function is available in the **<stdio.h>** header file in C library.

Ex: puts ("GOOD MORNING");

We can use this function to display messages as shown above.

Using putchar ( ) function we can display one character at a time. So, by using this function repeatedly with the help of a loop statement we can display total string.

Ex: char name [20];

    for (i=0;i<5;i++)

    putchar (name[i]);

2.6 String standard functions

The following table provides frequently used string handling functions which are supported by C compiler.

FUNCTIONS	DESCRIPTION
strlen ( )	Determines Length of a string
strcpy ( )	Copies a string from source to destination
strncpy ( )	Copies specified number of characters of a string to another string
strcmp ( )	Compares two strings (Discriminates between small and capital letters)
stricmp ( )	Compares two strings (Doesn"t Discriminates between small and capital letters)
strncmp ( )	Compares characters of two strings upto the specified length
strnicmp ( )	Compares characters of two strings upto the specified length. Ignores case.
strlwr ( )	Converts upper case characters of a string to lower case
strupr ( )	Converts lower case characters of a string to upper case
strdup ( )	Duplicates a string
strchr ( )	Determines first occurrence of a given character in a string
strrchr ( )	Determines last occurrence of a given character in a string
strstr ( )	Determines first occurrence of a given string in another string
strcat ( )	Appends source string to destination string
strncat ( )	Appends source string to destination string upto specified length
strrev ( )	Reverses all characters of a string
strset ( )	Sets all characters of string with a given argument or symbol
strnset ( )	Sets specified number of characters of string with a given argument or Symbol
strspn ( )	Finds up to what length two strings are identical
strpbrk ( )	Searches the first occurrence of the character in a given string and then it display the string from that character.

1. **strlen ( ) function:** This function counts the number of characters in a given string.

Syntax: strlen(string); (or) strlen(" string");

Example Program: To implement the purpose of strlen():

```
void main( )
```

```
{
char a[80];
int  len;
clrscr( );
printf(“enter string:”);

scanf(“%s”,a);    //    (or)           get (a);
len=strlen(a);
printf(“\n Name = %s”, a);

printf(“\n Total no of char‟s = %d”, len);
getch( );
}


(OR )
void main()
{
int s;

clrscr( );
s=strlen(“MRECW”);

printf(“\nTotal no of char‟s =%d”, s);
getch( );
}
```

Output:

```
Enter string: MRECW

 Name = MRECW
```

Output:

```
Total no of char‟s = 5
Total no of char‟s = 5
```

2. **strcpy ( ) function:** This function copies the content of one string to another. Here ***string1 is source string*** and ***string2 destination string.*** String1 and string2 are character arrays. String1 is copied into string2.

Syntax: strcpy(string2, string1); or strcpy(destination, source);

Example Program: To implement the purpose string copy( ).

```
 void main( )

{

char s1[80]="sachin", s2[80];

clrscr( );

printf ("\n\n \t given string =%s", s1);

strcpy(s2,s1);

printf("\n\n \t copied string = %s ", s2);

getch( );

}
```

Output:-

```
given string =sachin

copied string =sachin
```

3. **strncpy ( ) function:** This function performs the same task as strcpy ( ). The difference between them is that the strcpy( ) function copies the whole string to destination string. Whereas ***strncpy( )*** function copies specified length of characters from source to destination string.

Syntax: strncpy(string2, string1,n);

or

strcpy(destination, source,n);

Example Program: To copy source string to destination string up to a specified length.

Length is to be entered through the keyboard.

```
void main()
```

```
{
char s1[15], s2[15];
int n;
clrscr( );
printf ("\nEnter Source String:");
gets(s1);
printf ("\nEnter Destination String:");
gets(s2);
printf("\n Enter no. of characters to replace in Destination string:");
scanf("%d",&n);
strncpy(s2,s1,n);
printf("\n Source string = %s ", s1);
printf("\n Destination string = %s ", s2);
getch( );
}
```

Output:

```
Enter Source String: wonderful
Enter Destination String: beautiful
Enter no. of characters to replace in Destination string: 6
Source string = wonderful
Destination string=wonderful
```

4. **strcmp ( ) function:** This function is used to compare two strings identified by the arguments and it returns a value "0" if they are equal. If they are not equal it will return numeric difference (ASCII) between the first non-matching characters.

Syntax: strcmp(string1, string2);

Example Program1: To compare any strings using strcmp( ).

```
void main( )

{

char m1[30]=”amar”, m2[30] = “balu”;

int s;

clrscr( );
s=strcmp(m1, m2);

printf(“ %d ”, s);

getch( );

}
```

Output:-

```
-1
```

Example Program2: To compare any strings using strcmp( ).

```
void main( )

{

int m;

char s1[30], s2[30];

clrscr( );

printf(“enter 2 string \n”);

scanf(“%s%s”, &s1, &S2);

printf(“\n\n\n\t %s \t %s”, s1, s2);

m = strcmp(s1, s2);

printf(“\n\n \t %d”, m);

if(m==0)

printf(“\n\n both the strings are equal”);

else if(m<0)
printf("\n\n first string is less than second string \n");
else

printf("\n\n first string is greater than second string \n");

getch( );

}
```

Output:-

```
enter 2 strings

mrecw

computers

mrecw computers

1

First string is greater than second string
```

5. **stricmp ( ) function:** This function is used to compare two strings. The characters of the strings may be in lower to upper case. This function does not determinate between the cases.

This function returns zero when two strings are same and it returns non-zero value if they are not equal

Syntax: stricmp(string1, string2);

Example Program: To compare two strings using stricmp( ).

```
void main( )

{

char sr[30], tr[30];
int s;
clrscr( );
printf ("\nEnter Source String:");
gets(sr);
printf ("\nEnter Target String:");
gets(tr);
s=stricmp(sr,tr);
if(s==0)
    puts("The two strings are Identical.");
else
    puts("The two strings are Different");
}
```

Output:-

```
Enter Source String: HELLO

Enter Target String: hello

The two strings are Identical.
```

6. **strncmp ( ) function:** This function is used to compare two strings. This function is same as strcmp( ) but it compares the character of the string to a specified length

Syntax: strncmp(string1, string2,n);

or

strncmp(source, target, arguments);

Example Program: To compare two strings using strncmp( ).

```
void main( )

{
char sr[30], tr[30];
int s,n;
clrscr( );
printf ("\nEnter Source String:");
gets(sr);
printf ("\nEnter Target String:");
gets(tr);
printf("\n Enter length up to which comparison is to be made:"); scanf("%d",&n);
s=strncmp(sr,tr,n);
if(s==0)
puts("The two strings are Identical up to %d characters.",n); else
puts("The two strings are Different");

getche( );}
```

Output:-

```
Enter Source String: goodmorning

Enter Target String: goODNIGHT

Enter length up to which comparison is to be made:2

The two strings are Identical up to 2 characters.
```

7. **strnicmp ( ) function:** This function is used to compare two strings. This function is same as strcmp( ) but it compares the character of the string to a specified length.

Syntax: **strnicmp(string1, string2,n);**

(or)

strnicmp(source, target, arguments);

Example Program: To compare two strings using strnicmp( ).

```
void main( )

{

char sr[30], tr[30];

int s,n;

clrscr( );

printf ("\nEnter Source String:");

gets(sr);

printf ("\nEnter Target String:");

gets(tr);
printf("\n Enter length up to which comparison is to be made:");

scanf("%d",&n);

s=strnicmp(sr,tr,n);

if(s==0)

puts("The two strings are Identical up to %d characters.",n);
else

puts("The two strings are Different");

getche( );

 }
```

Output:-

```
Enter Source String: goodmorning

Enter Target String: GOODNIGHT

Enter length upto which comparison is to be made5 The two strings are different.Enter
Source String: goodmorning

Enter Target String: GOODNIGHT

Enter length upto which comparison is to be made4

The two strings are identical up to 4 characters.
```

8. **strlwr ( ) function:** This function can be used to convert any string to a lower case. When you are passing any uppercase string to this function it converts into lower case.

Syntax: **strlwr(string);**

strupr ( ) function: This function is the same as strlwr( ) but the difference is that strupr( ) converts lower case strings to upper case.

Syntax: **strupr(string);**

Example Program: To implment the purpose of string upper()& string lower().

```
void main ( )

{

char a[80]=”SACHIN”, b[80] = “Karan”;

clrscr( );

printf(“\n\n %s \t %s”, a,b);

strlwr(a);

strupr(b);

printf (“\n\n %s \t %s ”,a,b);

getch( );

}
```

Output:-

```
SACHIN      Karan

sachin      KARAN
```

9. strdup( ) function: This function is used for duplicating a given string at the allocated memory which is pointed by a pointer variable.

Syntax: string2 = strdup(string1);

Example Program: To enter the string and get it‟s duplicate.

```
void main( )

{

char s1[10], *s2;

clrscr( );

printf(“Enter text:”);

gets(s1);
s2 = strdup(s1);

printf("\n original string = %s \n duplicate string = %s",s1,s2);

getch();

}
```

Output:

```
Enter text: Engineering

original string = Engineering

duplicate string = Engineering
```

10. strchr( ) function: This function returns the pointer to a position in the first occurrence of the character in the given string.

Where, **string** is character array, **ch** is character variable & **chp** is a pointer which collects address returned by **strchr( )** function.

Example Program: To find first occurrence of a given character in a given string.

```
void main( )

{

char s[30], ch, *chp;

clrscr( );

printf("Enter text:");

gets(s);

printf("\n Character to find:");

ch = getchar( );

chp = strchr(string,ch);

if(chp)
printf("\n character %c found in string.",ch);

else

printf("\n character %c not found in string.",ch);

getch( );}
```

Output:

```
Enter text: Hello how are you

Character to find: r

Character r found in string.
```

11. **strrchr( ) function:** In place of strchr( ) one can use strrchr( ). The difference between them is that the strchr( ) searches occurrence of character from the beginning of the string where as strrchr( ) searches occurrence of character from the end (reverse).

Syntax: chp = strrchr(string, ch);

12. **strstr( ) function:** This function finds second string in the first string. It returns the pointer location from where the second string starts in the first string. In case the first occurrence in the string is not observed, the function returns a **NULL** character.

Syntax: strstr(string1,string2);

Example Program: To implement strstr( ) function for occurrence of second string in the first string.

```
void main( )

{

char s1[30], s2[30], *chp;

clrscr( );

printf("Enter text:");

gets(s1);
printf("\nEnter text:");

gets(s2);

chp = strstr(s1,s2);

if(chp)

printf("\n „%s‟ string is present in given
string.",s2); else

printf("\n „%s‟ string is not present in given string.",s2);

getch( );}
```

Output:

```
Enter text: INDIA IS MY COUNTRY

Enter text: INDIA

„INDIA‟ string is present in given string.
```

13. strcat ( ) function: This function appends the target string to the source string. Concatenation of two strings can be done using this function.

Syntax: **strcat(string1, string2);**

Example Program: To implement the purpose of string concat().

```
#include<string.h>

void main ( )

{

char s[80]="Hello", a[80] = "Mrecw";

clrscr( );

printf("\n %s \t %s", s,a);
strcat(s,a);    // strcat (s, “mrecw”);

printf (“\n %s ”,s);

getch( );

}
```

Output:-

```
Hello   Mrecw

HelloMrecw
```

14. strncat( ) function: This function is the same as that of strcat( ) function. The difference between them is that the former does the concatenation of two strings with another up to the specified length. Here, n is the number of characters to append.

Syntax: strncat(string1, string2, n);

Example Program: To append 2nd string with specified no. of characters at the end of the string using strncat( ) function.

```
#include<string.h>

void main ( )

{

char s[80]="Hello", a[80] = "Mrecw";

int n;

clrscr( );

printf("\n s=%s \t a=%s", s,a);

printf("\n Enter no. of characters to add:");

scanf("%d",&n);

strcat(s," ");

strncat(s,a,n);
printf ("\n %s ",s);

getch( );

}
```

Output:-

```
s= Hello        a = Mrecw

Enter no. of characters to add: 2

Hello Mr
```

15. Strrev ( ) function: This function simply reverses the given string.

Syntax: strrev(string);

Example Program1: To implement the purpose of string reverse().

```
#include<string.h>

{

char s[30]=”hello”;

clrscr( );

printf("\n\n original text = %s",s);

strrev(s);

printf ("\n\n reverse of text = %s",s);

getch( );

}
```

Output:-

```
        s1=hello

        s2=olleh
```

Example Program2: To implement the purpose of string reverse().

```
void main( )
{
char t[30];
clrscr( );
printf("enter string:");
scanf("%s", &t);    // gets(t);
printf("\n\n given string = %s",t);
strrev(t);
printf ("\n\n reversed string = %s",t);
getch( );
}
```

Output:

```
enter string: abcdefgh
given string = abcdefgh
reversed string = hgfedcba
```

16. strset( ) function: This function replaces every character of a string with the symbol given by the programmer i.e. the elements of the strings are replaced with the arguments given by the programmer.

Syntax: **strset(string,symbol);**

Example:

```
void main( )
{
```

```
char st[15], symbol;

clrscr( );

puts("Enter string:");

gets(st);

puts("Enter symbol for replacement:");

scanf("%c",&symbol);

printf("\n\n Before strset( ): %s",st);

strset(st,symbol);

printf("\n After strset( ): %s",st);

getch( );

}
```

Output:

```
Enter string: LEARN C

Enter symbol for replacement: Y

Before strset( ): LEARN C

After strset( ):   YYYYYY
```

17. strnset( ) function: This function is the same as that of **strset( )**. Here the specified length is provided. Where, **n** is the no. of characters to be replaced.

Syntax: **strnset(string,symbol,n);**

Example:

```
void main( )

{

char st[15], symbol;
int n;

clrscr( );

puts("Enter string:");

gets(st);

puts("\nEnter symbol for replacement:");

scanf("%c",&symbol);

puts("\nHow many string characters to be replaced:");
scanf("%d",&n);

printf("\n\n Before strset( ): %s",st);

strnset(st,symbol,n);

printf("\n After strset( ): %s",st);

getch( );

}
```

Output:

```
Enter string: ABCDEFGHIJ

Enter symbol for replacement: +

How many string characters to be replaced: 5

Before strset( ): ABCDEFGHIJ

After strset( ):   +++++FGHIJ
```

18. strspn( ): This function returns the position of the string from where the source array does not match with the target one.

Syntax: strspn(string1,string2);

Example Program: To indicate after what character the lengths of the 2 strings have no match.

```
void main( )

{

char s1[10], s2[10];

int len;

clrscr( );

printf("Enter string1:");

gets(s1);

printf("\nEnter string2:");

gets(s2);

len = strspn(s1,s2);

printf("\n\n After %d characters there is no match.\n", len);

getch( );

}
```

Output:

```
Enter string1: GOOD MORNING

Enter string2: GOOD BYE

After 5 characters there is no match.
```

19. strpbrk( ) function: This function searches the first occurrence of the character in a given string and then it display the string starting from that character.

Syntax: strpbrk(string1, string2);

Example Program: To print given string from first occurrence of given character.

```
void main( )

{
```

```
char string1[20], string2[10];

char *ptr;

clrscr( );

printf(“Enter string:”);

gets(string1);

printf(“\n Enter a character:”);

gets(string2);

ptr = strpbrk(string1,string2);

puts(“\n String from given character:”);

printf(ptr);

getch();

}
```

Output:

```
Enter string1: Good morning

Enter a character: d

String from given character: d morning.
```

2.7 ARITHMETIC OPERATIONS ON CHARACTERS

Observe the following example:

```
char x = “a“;

printf (“%d”,x);
```

The above statement will display integer value 97 on the screen even though x is a character variable, because when a character variable or character constant is used in an expression, it is automatically converted in to integer value by the system. The integer value is equivalent to ASCII code.

We can perform arithmetic operations on character constants and variables.

Ex: int x;

```
x=‟z‟-1;

printf(“%d”,x);
```

The above statement is valid statement and that will display 121 one as result, because ASCII value of “z‟ is 122 and therefore the resultant value is 122 – 1 = 121.

atoi ( ) function:

This library function converts a string of digits into their integer values.

Syntax: atoi (string);

Ex: int x;

```
char no[5] = “2012”;

 x = atoi(no);

 printf(“%d”,x);
```

The above statement will display x value as 2012 (which is an integer).

2.8 Structures:

Definition: In the C language structures are used to group together different types of variables under the same name.

Structure is a collection of variables of different data types under single name. A structure provides a convenience to group of related data types. A structure can contain variables, pointers, other structures, arrays or pointers. All these variables may contain data items of similar or dissimilar data types. Using these variables each item of a structure can be selected. Each variable in the structure represents an item and is called member or field of the structure. Each field has a type.

General format: struct tag_name

```
{
```

```
                    type1 member1;

                    type2 member2;

                    ……..

                    ……..

                   };
```

A structure is defined with the keyword „struct‟. Where,

“struct” is the keyword which tells the compiler that a structure is being defined member1, member2 … are called members of the structure.

The members are declared with curly braces. The members can be any of the data types such as int, char, float etc.There should be semicolon at the end of closing brace.

Example:

```
struct lib_books

{

char title[20];

char author[15];

int pages;

float price;

};
```

The complete structure declaration might look like this

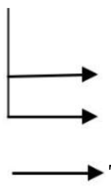
```
struct lib_books

{

char title[20];
char author[15];


int pages;

float price;

};struct lib_books, book1, book2, book3;
```

To holds the details of four fields namely title, author pages and price,the keyword struct declares a structure. These are the members of the structures. Each member may belong to same or different data type. The tag name can be used to define the objects that have the tag names structure. The structure we just declared is not a variable by itself but a template for the structure. We can declare the structure variables using the tag name anywhere in the program. For example the statement, struct lib_books book1,book2,book3; declares the book1,book2,book3 as variables of type struct lib_books each declaration has four elements of the structure lib_books.

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

2.9 Declaration of structure:

As variables are defined before they are used in the function, the structures are also defined and declared before they are used. A structure can be declared using three different ways



Tagged structures

Structure Variables

Type defined structures

Tagged structure means the structure definition associated with structure name is called tagged structure (i.e. tag_name is the name of the structure).

The tagged structure definition is shown
below; struct student

```
{
        char name[10];
        int roll_number;
        float avg_marks;
      };
```

Here, student is an identifier representing the structure name. It is also called tagname.

The complete structure definition along with structure declaration is shown below;

```
     struct student

     {
        char name[10];
        int roll_number;
        float avg_marks;
      };

     /* structure deflaration */ struct student ece, cse;
          truct tag_name
          {
             type1 member1;
            type2 member2;
```

… } v1, v2,v3; // **Structure variables**

Structure Variables:

```
struct student
    {
        char name[10];
        int roll_number;
        float avg_marks;

} ece, cse;
```

1. name, roll_number and average_marks are members of a structure and are not variables. So, they themselves do not occupy any memory.

But, once the structure definition is associated with variables such as ece and cse the compiler allocates memory for the structure variables

Type – Defined Structure

typedef struct

{

type1 member1;

type1 member1;

………}typeid;

- typedef is keyword added to the beginning of the definition.
- Using typedef it is not possible to declare a variable. Here, TYPE_ID can be treated as the new date type.

- Normally all typedef statements are defined at the beginning of the file immediately after #include and #define statements in a file.

```
typedef struct
{
    char name[10];
    int roll_number;
    float avg_marks;
 } student;
```

In the above definition, **student** is the user-defined data type.

Structure declaration using type-defined structure.

i.e. **student cse, ece;**

Initialization of structures

Consider the structure definition for an employee with three fields name, salary and is as shown below,

```
struct employee
        {
            char name[10];
            int id;
            float salary;
         } a = {“Shivashankar”,1686,20000.28};
```

(OR)

```
struct emp
        {
            char name[10];

            int id;

            float salary;

        };
```

Struct emp a = {“Shivashankar”,1686,20000.28};

Struct emp b = {“Shiva”,1886,18898.28}; //structure declaration with more than one value

Accessing structure members

We can access and assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. The link between a member and a variable is established using the ***member operator*** „.‟ Which is also known as “***dot operator”*** or ***“period operator”.***

Example:

```
struct book

{

char title[20];

char author[20];

int pages;

float price;

}book1, book2, book3;

strcpy(book1.title, “basic”);

strcpy(book1.author, “balagurusamy”);

book1.pages=250;

book1.price=120.50;
```

we can also use **scanf** to give values through the keyboard.

```
    scanf(“%s \n”,book1.title);
```

```
        scanf("%s \n",&book1.pages);
```

Example2:

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

The keyword struct informs the compiler for holding fields (title, author, pages and price in the above example). All these are the members of the structure. Each member can be of same or different data type.

The tag name followed by the keyword struct defines the data type of struct. The tag name library_books in the above example is not the variable but can be visualized as template for the structure. The tag name is used to declare struct variables.

Ex: struct library_books book1,book2,book3;

The memory space is not occupied soon after declaring the structure, being it a template. Memory is allocated only at the time of declaring struct variables, like book1 in the above example. The members of the structure are referred as - book1.title, book1.author, book1.pages, book1.price.

Copying and comparing structure variables

Two variables of the same structure type can be copied the same way as ordinary variables. If **person1** and **person2** belong to the same structure, then the following statements are valid:

book2=book1;

C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

book1 == book2

book1 != book2

Operations on individual members

The individual members are identified using the members are identified using the member operator. A member with the *dot operator* along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operator.

Example: float sum = student1.marks + student2.marks; Student2.marks *= 0.5;

We can also apply increment and decrement operators to numeric type members.

```
student1.number++;

++student1.number;
```

Three ways to access members

We have used the dot operator to access the members of structure variables. In fact, there are two other ways. The identifier **ptr** is known as **pointer** that has been assigned the address of the structure variable n. now members can be accessed in three ways:

Using dot notation	:	n.x
Using indirection notation	:	(*ptr).x
Using selection notation	:	Ptr->x

2.10 ARRAY OF STRUCTURES

We may declare an array of structures, each element of the array representing a structure variable. For example:

struct class student[100];

define an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class**.

Write a c program to calculate the subject wise and student wise totals and store them as a part of the structure.

```
struct marks
{
        int sub1;
        int sub2;
        int sub3;
        int total;
};
void main( )
{
        int i;
        struct marks student [3]={{45,67,81,0}, {75,53,69,0},{57,36,71,0}};
        struct marks total;
        for(i=0;i<=2;i++)
        {
                student[i].total=student[i].sub1+ student[i].sub2+ student[i].sub3;
                total.sub1=total.sub1+student[i].sub1;
                total.sub2=total.sub2+student[i].sub2;
                total.sub3=total.sub3+student[i].sub3;
                total.total=total.total+student[i].total;
        }
        printf("Student                         Total\n\n");
        for(i=0;i<=2;i++)
        printf("student[%d]          %d\n", i+1,student[i].total);

        printf("\n Subject              Total\n\n");

        printf("%s            %d\n%s              %d\n%s              %d\n",

        "subject1    ",total.sub1, "subject2         ",total.sub2, "subject3      ",total.sub3);

        printf("\n grand total  = %d \n", total.total);

        getch();

        }
```

Output:

```
        Student              Total

        student[1]           193

        student[2]           197

        student[3]           164

        Subject      Total

        subject1     177

        subject2     156

        subject3     221

        grand total   = 554
```

Arrays within structure

C permits the use of arrays as structure numbers. We have already used arrays of characters inside a structure. Similarly, we can use single dimension or multidimensional arrays of type **int** or **float**.

```
struct marks

        {
          int number;

          float subject[3];

        } student[2];
```

Here, the member **subject** contains three elements, **subject[0], subject[1]** and **subject[2].**

These elements can be accessed using appropriate subscripts.

Example program:

```
void main( )

{

struct marks

{

        int sub[3];

        int total;

};

struct marks student [3]={45,67,81,0,75,53,69,0,57,36,71,0};

struct marks total;

int i, j;

        for(i=0;i<=2;i++)

        {

          for(j=0;j<=2;j++)

           {

                student[i].total += student[i].sub[j];

                total.sub[j] += student[i].sub[j];

        }
```

```
                total.total += student[i].total;

        }

        printf("Student                          Total\n\n");

        for(i=0;i<=2;i++)

        printf("student[%d]             %d\n", i+1,student[i].total);

        printf("\n Subject              Totla\n\n");

        for(j=0; j<=2; j++)

        printf("subject - %d             %d\n", j+1,total.sub[j]);

        printf("\n grand total  = %d \n", total.total);

        getch();

        }
```

Output:

```
        Student                 Total

        student[1]              193

        student[2]              197

        student[3]              164

        Subject         Total

        subject1        177

        subject2        156

        subject3        221

        grand total   = 554
```

Structures within Structures

A structure within a structure means *nesting of structures.* Nesting of structures is permitted in C.

Write a C program to read and display car number, starting time and reaching time.

Use structure within structure.

```
void main( )
{
struct time
{
int second;
int minute;
int hour;
};
struct tt
{
int carno;
struct time st;
struct time rt;
};
struct tt r1;
clrscr( );
printf("\n car no. \t starting time \t reaching time:");
scanf("%d",&r1.carno);
scanf("%d%d%d",&r1.st.hour, &r1.st.minute, &r1.st.second);
scanf("%d%d%d",&r1.rt.hour, &r1.rt.minute, &r1.rt.second);

printf("\n \t carno. \t starting time \t reaching time \n");

printf("\t%d\t",r1.carno);

printf("\t%d:%d:%d\t",r1.st.hour, r1.st.minute, r1.st.second);

printf("\t%d:%d:%d",r1.rt.hour, r1.rt.minute, r1.rt.second);

getch();   }
```

Output:

```
car no.          starting time    reaching time: 4321 2 50 30 3 50 25

carno   starting time    reaching time

4321    2:50:30          3:50:25
```

Structures and Functions

Like variables of standard data type structure variables also can be passed to the function by value or address.

```
struct book {

char n[30];

char author[30];

int pages;

};

void main()

{

struct book b1={"JAVA COMPLETE REFERENCE","P.NAUGHTON", 886};

show(&b1);

getch();
}
show(struct book *b2)
{
clrscr();
printf("\n %s by %s of %d pages", b2->n, b2->author, b2->pages);
}
```

Output: JAVA COMPLETE REFERENCE by P.NAUGHTON of 886 pages

2.11 POINTER TO STRUCTURE

If you want a pointer to a structure you have to use the -> (infix operator) instead of a dot.

Take a look at the following example:

```
        #include<stdio.h>
        typedef struct telephone
        {
                char *name;
                int number;
        }TELEPHONE;
        void main()
        {
                TELEPHONE index;
                TELEPHONE *ptr;
                ptr = &index;
                ptr->name = "Jane Doe";
                ptr->number = 12345;
```

```
                    printf("Name: %s\n", ptr->name);

                    printf("Telephone number: %d\n", ptr->number);

                    getch();

          }
```

Note: The -> (infix operator) is also used in the printf statement.

2.12 Self-Referential Structures

A structure definition which includes at least one member as a pointer to the same structure is known as self-referential structure.

Syntax:

```
struct struct_name

{

    datatype datatypename;

    struct_name * pointer_name;

};
```

Example:

```
Struct student

{

Char n[50];

Int rollno;

Struct student *ptr;

};
```

Self-referential structures are mainly used in the implementation of data structures.

Example: Linked List, trees etc.

2.13 UNIONS

Unions are like structures, in which the individual data types may differ from each other. All the members of the union share the same memory / storage area in the memory. Every member has unique storage area in structures. In this context, unions are utilized to observe

the memory space. When all the values need not assign at a time to all members, unions are efficient. Unions are declared by using the keyword union, just like structures.

```
union tag_name {

type1 member1;

type1 member2;


};
```

```
Ex: union item {

int code;
```

floa the memory space. When all the values need not assign at a time to all members, unions are efficient. Unions are declared by using the keyword union, just like structures.

```
Ex: union item {

int code;

float price;


};
```

The members of the unions are referred as - book1.title, book1.author, book1.pages, book1.price.

What are the properties of Union?

A union is utilized to use same memory space for all different members of union. Union offers a memory section to be treated for one variable type , for all members of the union. Union allocates the memory space which is equivalent to the member of the union, of large memory occupancy.

A union is like a structure in which **all members** are stored at the **same** address. Members of a union can only be accessed one at a time. The union data type was invented to prevent memory fragmentation. The union data type prevents fragmentation by creating a standard size for certain data. Just like with structures, the members of unions can be accessed with the . and -> operators. For example:

```
#include<stdio.h>

typedef union myunion

{

        double PI;

        int B;

}MYUNION;

void main()

{

        MYUNION numbers;

        numbers.PI = 3.14;

        numbers.B = 50;

getch();

}
```

t price;};

UNION OF STRUCTURE

We know that one structure can be nested within another structure. It the same way a union can be nested another union. We can also create structure in a union or vice versa.

Write a program to use structure within union. Display the contents of structure elements.

```
void main( )
{
      struct x
      {
        float f;
        char p[2];
      };
      union z
      {
        struct x set;
      };
      union z st;
      st.set.f = 5.5;
      st.set.p[0] = 65;
      st.set.p[1] = 66;
      clrscr();
      printf("\n%g", st.set.f);
      printf("\n%c",st.set.p[0]);
      printf("\t%c",st.set.p[1]);
}
```

output: 5.5

```
A       B
```

Difference between Structure and Union

Structure	**Union**
1. The keyword struct is used to define a structure.	**1.** The keyword union is used to define a structure.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of the structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused	**2.** When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member. So, a size of union is equal to the size of largest member.

slack bytes.	
3. Each member within a structure is assigned unique storage area.	**3.** Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order. This indicates that memory for each member will start at different offset values.	**4.** The address is same for all the members of a union. This indicates that every member begins at offset values.
5. Altering the value of a member will not affect other members of the structure.	**5.** Altering the value of any of the member will alter other member values.
6. Individual members can be accessed at a time.	**6.** Only one member can be accessed at a time.
7. Several members of a structure can be initialized at once.	**7.** Only the first member of a union can be initialized.

2.14 POINTERS :

One of the powerful features of C is ability to access the memory variables by their memory address. This can be done by using Pointers. The real power of C lies in the proper use of Pointers.

A pointer is a variable that can store an address of a variable (i.e., 112300).We say that a pointer points to a variable that is stored at that address. A pointer itself usually occupies 4 bytes of memory (then it can address cells from 0 to 232-1).

Advantages of Pointers :

1. A pointer enables us to access a variable that is defined out side the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.

2.15 Definition :

A variable that holds a physical memory address is called a pointer variable or Pointer.

Declaration :

```
          Datatype * Variable-name;
Eg:-   int *ad;              /* pointer to int */
          char *s;             /* pointer to char */
          float *fp;           /* pointer to float */
          char **s;            /* pointer to variable that is a pointer to char */
```

A pointer is a variable that contains an address which is a location of another variable in memory.

Consider the Statement

```
p=&i;
```

Here '&' is called address of a variable. 'p' contains the address of a variable i

The operator & returns the memory address of variable on which it is operated, this is called Referencing.

The * operator is called an indirection operator or dereferencing operator which is used to display the contents of the Pointer Variable.

Consider the following Statements :

```
int *p,x; x
=5; p= &x;
```

Assume that x is stored at the memory address 2000. Then the output for the following printf statements is :

```
                                                Output
Printf("The Value of x is %d",x);                    5
Printf("The Address of x is %u",&x);              2000
Printf("The Address of x is %u",p);               2000
Printf("The Value of x is %d",*p);                   5
Printf("The Value of x is %d",*(&x));                5
```

POINTER FUNCTION ARGUMENTS

Function arguments in C are strictly pass-by-value. However, we can simulate pass-by-reference by passing a pointer. This is very useful when you need to Support in/out(bi-directional) parameters (e.g. swap, find replace) Return multiple outputs (one return value isn't enough) Pass around large objects (arrays and structures).

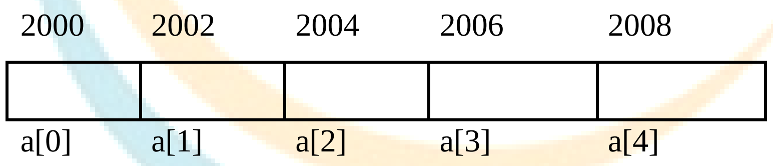
```
/* Example of swapping a function can't change parameters */
void bad_swap(int x, int y)
{
int      temp;
temp = x; x =
y;
y = temp;
}

/* Example of swapping - a function can't change parameters, but if a parameter is a pointer it
can change the value it points to */
void good_swap(int *px, int *py)
{
int temp;
temp = *px; *px
=  *py;  *py  =
temp;
  }
#include <stdio.h>
```

```
void bad_swap(int x, int y);
void good_swap(int *p1, int *p2);
main() {
int a = 1, b = 999;
printf("a = %d, b = %d\n", a, b);
bad_swap(a, b);
printf("a = %d, b = %d\n", a, b);
good_swap(&a, &b);
printf("a = %d, b = %d\n", a, b);
}
```

POINTERS AND ARRAYS :

When an array is declared, elements of array are stored in contiguous locations. The address of the first element of an array is called its base address.

Consider the array



The name of the array is called its base address.

i.e., a and k& a[20] are equal.

Now both a and a[0] points to location 2000. If we declare p as an integer pointer, then we can make the pointer P to point to the array a by following assignment

P = a;

We can access every value of array a by moving P from one element to another.

i.e., P	points to 0th element
P+1	points to 1st element
P+2	points to 2nd element

P+3	points to 3rd Element
P +4	points to 4th Element

Reading and Printing an array using Pointers :

```
main()
        {
                int *a,i;
                printf("Enter five elements:"); for
                (i=0;i<5;i++)
                        scanf("%d",a+i);
                        printf("The array elements are:"); for
                        (i=o;i<5;i++)
                                printf("%d", *(a+i));
        }
```

In one dimensional array, a[i] element is referred by (a+i) is
the address of ith element.
* (a+i) is the value at the ith element.

In two-dimensional array, a[i][j] element is represented as
((a+i)+j)

POINTERS AND FUNCTIONS:

Parameter passing mechanism in 'C' is of two types.

1.Call by Value 2.Call
by Reference.

The process of passing the actual value of variables is known as Call by Value.The process of calling a function using pointers to pass the addresses of variables is known as Call by Reference. The function which is called by reference can change the value of the variable used in the call.

Example of Call by Value:

```
#include      <stdio.h>
void      swap(int,int);
main()

{
        int a,b;
        printf(“Enter the Values of a and b:”);
        scanf(“%d%d”,&a,&b); printf(“Before
        Swapping \n”); printf(“a = %d \t b = %d”,
        a,b); swap(a,b);
        printf(“After Swapping \n”);
        printf(“a = %d \t b = %d”, a,b);
}
void swap(int a, int b)
{
        int      temp;
        temp = a; a =
        b;
        b = temp;
}
```

Example of Call by Reference:

```
#include<stdio.h>
main()
{
        int a,b; a =
        10; b = 20;
        swap (&a, &b); printf(“After
        Swapping \n”);

        printf(“a = %d \t b = %d”, a,b);
}
void swap(int *x, int *y)
{
        int temp; temp
        = *x; *x = *y;
        *y = temp;

}
```

ADDRESS ARITHIMETIC :

Incrementing/Decrementing a pointer variable ,adding and subtracting an integer from pointer variable are all legal and known as pointer arithmetic. Pointers are valid operands in arithmetic expressions ,assignment expressions ,and comparison expressions.
However not all the operators normally used in these expressions are valid in conjunction with pointer variable.

A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented(+ +) or decremented(--) ,an integer may be added to a pointer
(+ or +=),an integer may be subtracted from a pointer(- or -=),or one pointer may be subtracted from another.

We can add and subtract integers to/from pointers – the result is a pointer to another element of this type

Ex : int *pa; char *s;

s-1 →points to char before s (1 subtracted)
pa+1→ points to next int (4 added!)
s+9 →points to 9th char after s (9 added)
++pa→ increments pa to point to next int

NULL POINTER :

‘Zero’ is a special value we can assign to a pointer which does not point to anything most frequently, a symbolic constant NULL is used. It is guaranteed, that no valid address is equal to 0.The bit pattern of the NULL pointer does not have to contain all zeros usually it does or it depends on the processor architecture. On many machines, dereferencing a NULL pointer causes a segmentation violation.

NULL ptr is not the same as an EMPTY string.

```
const  char*  psz1  =  0;
const  char*  psz2  =  "";
assert(psz1 != psz2);
```

Always check for NULL before dereferencing a pointer.

```
        if (psz1)
        /* use psz1 */
        sizeof(psz1) // doesn't give you the number of elements in psz1. Need
        additional size variable.
```

VOID POINTER :

In C ,an additional type void *(void pointer) is defined as a proper type for generic pointer. Any pointer to an object may be converted to type void * without loss of information. If the result is converted back to the original type ,the original pointer is recovered .

Ex:

```
        main()
        {
                void *a; int
                n=2,*m;
                double d=2.3,*c;
                a=&n;
                m=a;
                printf("\n%d %d %d",a,*m,m);
                a=&d;
                c=a;
                printf("\n%d %3.1f %d",a,*c,c);
        }
```

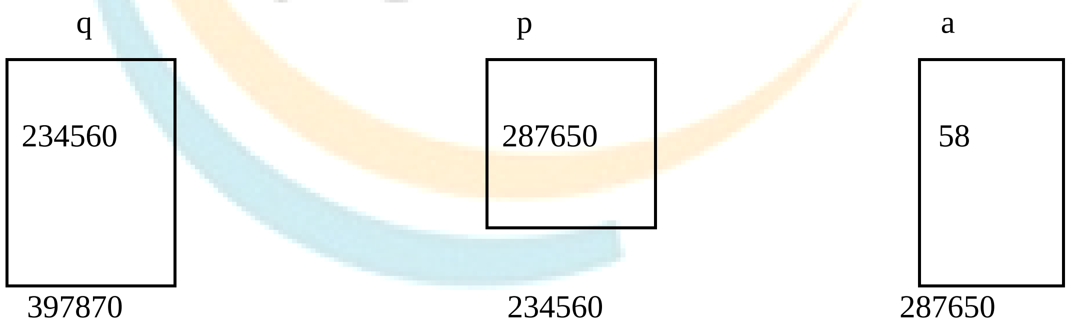
In the above program a is declared as a pointer to void which is used to carry the address of an int(a=&n)and to carry the address of a double(a=&d) and the original pointers are recovered with out any loss of information.

POINTERS TO POINTERS :

So far ,all pointers have been pointing directely to data.It is possible and with advanced data structures often necessary to use pointers to that point to other pointers. For example,we can have a pointer pointing to a pointer to an integer.This two level indirection is seen as below:

```
 //Local declarations
int a;
int* p; int
**q;
```

Ex:



pointer to pointer to integer **pointer to integer** **integer variable**

```
//statements
a=58;
p=&a;
q=&p;
printf("%3d",a);
printf("%3d",*p);
printf("%3d",**q);
```

There is no limit as to how many level of indirection we can use but practically we seldom use morethan two.Each level of pointer indirection requires a separate indirection operator when it is dereferenced .

In the above figure to refer to 'a' using the pointer 'p', we have to dereference it as shown below.

*p

To refer to the variable 'a' using the pointer 'q' ,we have to dereference it twice toget to the integer 'a' because there are two levels of indirection(pointers) involved.If we dereference it only once we are referring 'p' which is a pointer to an integer .Another way to say this is that 'q' is a pointer to a pointer to an integer.The doule dereference is shown below:

**q

In above example all the three references in the printf statement refer to the variable 'a'. The first printf statement prints the value of the variable 'a' directly,second uses the pointer 'p',third uses the pointer 'q'.The result is the value 58 printed 3 times as below

```
58      58       58
```

2.16 User-defined data types:

The data types defined by the user are known as the user-defined data types. C provides two identifiers ***typedef*** and ***enum*** to create new data type names.

Typedef:

It allows the user to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables.

Syntax: typedef type identifier;

Type: refers to an existing data type

Identifier: refers to new name given to the data type.

Typedef cannot create a new type but it creates a new name to the existing type.

The main *advantage* of typedef is that we can create meaningful data type names for increasing the ***readability of the program***.

Ex: 1. **typedef int marks;**

Here marks are later used to declare variables as:

marks sub1[60], marks sub2[60];

sub1[60], sub2[60] are declared as 60 elements **integer** variables.

Example:

```
#define H 60
void main()
{
typedef int hours;
hours hr;
printf("enter hours:\n");
scanf("%d",&hr);
printf("\n minutes=%d \t
        Seconds=%d",hr*H,hr*H*H);
}//OP:hrs=2 mins=120 secs = 7200
```

Enumeration:

The enumeration data type is defined as follows:

Syntax: **enum identifier {value$_1$, value$_2$, …….,value$_n$} ;**

Identifier: it is a user defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (*enumeration constants*).

Declaration of variables of this new type:

enum identifier v$_1$, v$_2$, v$_3$,…….,v$_n$;

The enumerated variables **v$_1$, v$_2$, v$_3$,…….,v$_n$** can only have one of the values **value$_1$, value$_2$, …….,value$_n$.**

Assignments: ex: v$_1$ = value3;

v$_4$ = value2;

Ex: enum day {Monday, Tuesday, Wednesday, …. Sunday}; (definition of identifier „day‟)

enum day week_st, week_end;   (declaring variable of new type day)

week_st = Monday;

week_end= Friday;

Note:

2. *The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants* .i.e. 0 is assigned to value1, 1 is assigned to value2, 2 is assigned to value3, and soon. User can also assign values **explicitly** to the enumeration constants.

   Ex**: enum day** {Monday=1, Tuesday, Wednesday, …. Sunday};

3. The definition and declaration of enumeration variables can be combined in one statement.

   Ex: **enum day** {Monday, Tuesday, Wednesday, …. Sunday} week_st, week_end;

Tutorial Questions

1.
```
include <stdio.h>
struct student
{
    char *name;
};
void main()
{
    struct student s, m;
    s.name = "st";
    m = s;
    printf("%s%s", s.name, m.name);
} Ans_______________________
```

2.
```
#include <stdio.h>
struct student
{
    char *name;
};
struct student s[2];
void main()
{
    s[0].name = "alan";
    s[1] = s[0];
    printf("%s%s", s[0].name, s[1].name);
    s[1].name = "turing";
    printf("%s%s", s[0].name, s[1].name);
}  Ans___________________________
```

3. **Identify the error**

```
      #include <stdio.h>
                    struct point
                  {
                   int x;
                   int y;
                  };
             int main()
               {
                   struct point p = {1};
                   struct point p1 = {1};
                  if(p == p1)
                   printf("equal\n");
                  else
        printf("not equal\n");
}
```

4. **Find the output of the following Program:**

```
#include <stdio.h>
int main(void) {
    int i,t[3];
 for(i = 2; i >=0 ; i--)
         t[i] = i - 1;
    printf("%d",t[1] - t[t[0] + t[2]]);
    return 0;
}
```

Ans________________________

5. Implement a c program to print 1 to 100 numbers without using loops.
6. Implement a C program to multiply three numbers by passing it as arguments to a function.

7.
   ```
   include<stdio.h>
       #include<string.h>
     int main()
     {
          char str1[20] = "Hello", str2[20] = " World";
            printf("%s\n", strcpy(str2, strcat(str1, str2)));
           return 0;
   } Ans______________________________
   ```

8.
   ```
        #include<stdio.h>
           #include<string.h>
      int main()
       char sentence[80];
      int i;
     printf("Enter a line of text\n");
        gets(sentence);
      for(i=strlen(sentence)-1; i >=0; i--)
   putchar(sentence[i]);return o;}
   Ans____________________
   ```

Assignment Questions

SET 1:

1. Implement a c program to transpose the given matrix.
2. Implement a C program to add elements of array by passing it as an argument.

SET 2:

1. Implement a C program to find maximum and minimum elements in an array.

2. Explain various way to initialize array?

SET 3:

1. Implement a c program to find the largest from given strings.
2. Discuss call by value and call by reference.

SET 4

1. Implement a c program to show the difference between gets() and scanf().
2. Explain the importance of reaaloc() and free().

SET 5:

1. Implement a C program to read and print an employee's detail using structure
2. Discuss about bit field.

SET-6

3. Implement a C program using array of structure.
4. Can we add two structure variables?If yes, justify?

Important Questions

1. Write a program to demonstrate passing an array argument to a function? Consider the problem of finding largest of N numbers defined in an array.
2. Define an array. What are the different types of arrays? Explain
3. (a) Write a C program to do matrix multiplications.
   (b) Write in detail about one dimensional and multidimensional arrays. Also write about how initial values can be specified for each type of array.

   What is meant by structure? Discuss with a C-program about operations on structures.
4. Discuss about bit fields.
5. Give an example for self-referential structure
6. Differentiate between structure and union. Give examples for each.With examples discuss Array of Structures and Structure of Arrays.
7. Write a program to handle students profile data using structures.
8. Write about enumerated data types.
9. Define structure.Distinguish between structures and functions.
10. Explain enumerated type, Structure and Union types with examples.
11. Write a short notes on unions within structures.
12. Discuss about self referential structures with examples.
13. Write a program to print 60 student’s total marks and grades.
14. What are the advantages of structures and unions? Discuss.
15. How to access structure elements? Discuss.
16. Explain about declaration, initialization and accessing of structures.

17. Discuss about complex structures.
18. How to declare a union in ‘C’ explain with an example.Write a ‘C’ program to compute the monthly pay of 100 employees using each employee name and basic pay. The DA is computed as 2.5% of the basic pay, Gross salary (Basic pay + DA). Display the employees name and gross salary.
19. Explain about pointers to pointers with an example.
20. How pointers can be used for declaring multi dimensional arrays?
21. What is character pointer? How to initialize pointer variables?
22. What is dynamic memory allocation? Discuss with examples.
23. Define a pointer. What is a function pointer with an example?
24. Explain various string manipulation functions in ‘C’ programming?
25. What is the use of strcat() function?
26. Write a C program to add two numbers using call by pointers method.
27. Write a C program that implements string concatenate operation STRCAT (str1, str2) that combines a string str1 to another string str2 without using library function.
28. Explain pointer arithmetic with a sample C-program.
29. What are the string manipulation functions? Explain their usage.
30. What is null pointer? What is a void pointer? Explain when null pointer and void pointer are used.
31. Explain any five string manipulation functions with examples.
32. Write a program using pointers to compute the sum of all elements stored in an array.
33. Explain the concept of passing strings to functions as dynamic arrays with a programs.
34. Explain unformatted functions in c?
35. How to declare a pointer to a function? What is its use?
36. What is the difference between calloc and malloc functions?
37. What is the difrence between gets() and scanf()
38. Write a program to display the location of a character ‘T’ in a given string. Give the signatures of getch, puts functions.

FILL IN BLANKS:-

1. Array is a group of ________________data items.
2. Array is a _______________________ data type.
3. Arrays can be of __________________data types.
4. The elements or the members of the Array are accessed by _______________ names. (same/different).
5. Elements of an array are differentiated by a single subscript value called _______________.

6. The index value of Array Starts from ________________ number.

7. Arrays are of _______ dimensional and ___________dimensional and ___________ Dimensional.

8. main() { int i=1; while (i<=5) { printf("%d",i); if (i>2) goto here; i++; } } fun() { here: printf("PP");

9. standard function to find square root___________

10. Function is a _________________

11. In array memory is allocated ____________

12. Predict the output of below program:

```
#include <stdio.h>


int main()

{

   int arr[5];


   // Assume that base address of arr is 2000 and size of integer

      // is 32 bit

   arr++;

   printf("%u", arr);


   return 0;

}
```

13. What is output?

```
# include <stdio.h>


void print(int arr[])
```

```
{
  int n = sizeof(arr)/sizeof(arr[0]);
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", arr[i]);
}


int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
  print(arr);
  return 0;
}
```

14. Predict output of following program

```
int main()
{
   int i;
   int arr[5] = {1};
   for (i = 0; i < 5; i++)
       printf("%d ", arr[i]);
   return 0;
}
```

15. n C, parameters are always--------------
16. Are the expression *ptr++ and ++*ptr are same?

17. The following program reports an error on compilation.

#include<**stdio.h**>

```
int main()
{
   float i=10, *j;
   void *k;
   k=&i;
   j=k;
   printf("%f\n", *j);
   return 0;
}
```

18. Are the three declarations char **apple, char *apple[], and char apple[][] same?
19. What is the output of this C code?

```
int main()
{
char *p = NULL;
char *q = 0;
if (p)
printf(" p ");
else
printf("nullp");
if (q)
printf("q\n");
else
printf(" nullq\n");
}
```

20. #include<stdio.h>

```
#include<string.h>

int main()
{
   char str1[20] = "Hello", str2[20] = " World";
   printf("%s\n", strcpy(str2, strcat(str1, str2)));
   return 0;
}
```

21. #include<stdio.h>

```
int main()
{
   char p[] = "%d\n";
   p[1] = 'c';
   printf(p, 65);
   return 0;
}
```

22. String manipulation functions use __________________library function.

23 Getchar() is ______________ function.

24 Int **p is ____________________ declaration.

26. Is String is data type?_____

26. __________________are themselves a collection of different data types?

27. User-defined data type can be derived by___________.

28. Which operator connects the structure name to its member name______

29. _________cannot be a structure member.
30. What is the output of this C code?

```
void main()
{
struct student
{
int no;
char name[20];
};
struct student s;
s.no = 8;
printf("%d", s.no);
}
```

Number of bytes in memory taken by the below structure is?

```
struct test
{
int k;
char c;
};
```

31. Size of a union is determined by size of the____________

32. Point out the error in the program?

```
typedef struct data mystruct;
 struct data
     {
     int x;
       mystruct *b;
      };
```

33. #include<stdio.h>

```
int main()
{
    struct a
    {
        float category:5;
        char scheme:4;
    };
    printf("size=%d", sizeof(struct a));
    return 0;
}
```

34. #include<stdio.h>

```
int main()
{
   struct emp
   {
      char name[20];
      float sal;
   };
   struct emp e[10];
   int i;
   for(i=0; i<=9; i++)
      scanf("%s %f", e[i].name, &e[i].sal);
   return 0;
}
```

PREPROCESSOR

3.1 Preprocessor

Before a C program is compiled in a compiler, source code is processed by a program called **preprocessor**. This process is called preprocessing.

Commands used in preprocessor are called preprocessor directives.

- It is placed before the main().
- It begins with a # symbol.
- They are never terminated with a semicolon.

Commonly used Preprocessor commands

1. #include

#include is used to insert specific header file into C program.

Syntax:

#include filename

- The content that is included in the filename will be replaced at the point where the directive is written.
- By using the file inclusive directive, we can include the header files in the programs.
- Macros, function declarations, declaration of the external variables can all be combined in the header file instead of repeating them in each of the program.
- The stdio.h header file contains the function declarations and all the information regarding the input and output.

There are two ways of the file inclusion statement:

i) #include "file-name"
ii) #include <file-name>

- If the first way is used, the file and then the filename in the current working directory and the specified list of directories would be searched.
- If the second way, is used the file and then the filename in the specified list of directories would be searched.

2. #define

#define is used to create symbolic constants (known as macros) in C programming language. This preprocessor command can also be used with parameterized macros.

Syntax:

#define name replacement text

Where,
name – it is known as the micro template.
replacement text – it is known as the macro expansion.

- A macro name is generally written in capital letters.
- If suitable and relevant names are given macros increase the readability.
- If a macro is used in any program and we need to make some changes throughout the program we can just change the macro and the changes will be reflected everywhere in the program.

Example : Simple macro

```
#define LOWER 30
void main()
{
    int i;
    for (i=1;i<=LOWER; i++)
    {
        printf("\n%d", i);
    }
}
```

Example : Macros with arguments

```
#define AREA(a) (3.14 * a * a)
void main()
{
    float r = 3.5, x;
    x = AREA (r);
    printf ("\n Area of circle = %f", x);
}
```

3. #undef:

The #undef directive tells the preprocessor to remove all definitions for the specified macro. A macro can be redefined after it has been removed by the #undef directive.

Syntax:

#undef macro_definition

Example:

```
#include <stdio.h>
#define PI 3.14
```

```
#undef PI
void main()
 {
   printf("%f",PI);
 }
```

Output: Compile Time Error: 'PI' undeclared

4. #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code.

Syntax:

```
#if expression
//code
#endif
```

Example:

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 0
void main() {
#if (NUMBER==0)
printf("Value of Number is: %d",NUMBER);
#endif
getch();
}
```

Output: Value of Number is: 0

5. #ifdef:

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifdef MACRO
//code
#endif
```

Example:

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

```
int a=0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output: Value of a: 2

6. #ifndef:

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO
//code
#endif
```

Example:

```
#include <stdio.h>
#include <conio.h>
#define INPUT
void main() {
int a=0;
#ifndef INPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Enter a:5
Value of a: 5
```

3.2 FILES

File is a collection of bytes that is stored on secondary storage devices like disk. There are two kinds of files in a system. They are,

1. Text files (ASCII)
2. Binary files

- Text files contain ASCII codes of digits, alphabetic and symbols.
- Binary file contains collection of bytes (0's and 1's). Binary files are compiled version of text files.

Text Files

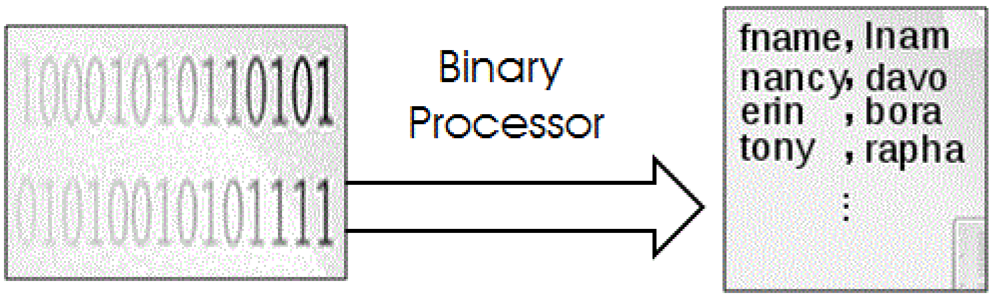
- Text File is Also Called as "**Flat File**".
- Text File Format is **Basic File Format** in C Programming.
- Text File is simple **Sequence of ASCII Characters**.
- Each Line is Characterized by **EOL Character** (End of Line).

Text file formats

1. Text File have **.txt** Extension.
2. The precise definition of the .txt format is not specified, but typically **matches the format accepted by the system terminal** or simple text editor.
3. Files with the .txt extension can **easily be read or opened by any program that reads text** and, for that reason, are considered universal (or platform independent).
4. Text Format Contain **Mostly English Characters.**

Binary Files

1. Binary Files Contain Information Coded Mostly in Binary Format.
2. Binary Files are **difficult to read for human**.
3. Binary Files can be processed by **certain applications or processors**.
4. Only **Binary File Processors can understood Complex Formatting**Information Stored in Binary Format.
5. Humans can read binary **files only after processing**.
6. All Executable Files are **Binary Files**.
7. *I/O operations are much faster with binary data.*
8. *Binary files are much smaller in size than text files.*
9. *Some data cannot be converted to character formats.*



3.3 Creating and Reading and writing text and binary files

File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE *fptr;

Opening a file - for creation and edit

Opening a file is performed using the library function in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

ptr=fopen("fileopen","mode");

For Example:
fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file

R	Open for reading.	If the file does not exist, fopen() returns NULL.
Rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
W	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
Wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
A	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
Ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function **fclose().**

fclose(fptr);

Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

Writing to a text file

Example

```
#include <stdio.h>
int main()
{
   int num;
   FILE *fptr;
   fptr = fopen("C:\\program.txt","w");
   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }
   printf("Enter num: ");
   scanf("%d",&num);
   fprintf(fptr,"%d",num);
   fclose(fptr);
   return 0;
}
```

Reading from a text file

Example

```
#include <stdio.h>
int main()
{
   int num;
   FILE *fptr;
   if ((fptr = fopen("C:\\program.txt","r")) == NULL){
       printf("Error! opening file");  // Program exits if the file pointer returns NULL.
       exit(1);
   }
   fscanf(fptr,"%d", &num);
   printf("Value of n=%d", num);
   fclose(fptr);
   return 0;
}
```

Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the function fwrite(). The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

fwrite(address_data,size_data,numbers_data,pointer_to_file);

Example:

```
#include <stdio.h>
struct threeNum
{
  int n1, n2, n3;
};

int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.bin","wb")) == NULL)
  {
     printf("Error! opening file");   // Program exits if the file pointer returns NULL.
     exit(1);
  }
  for(n = 1; n < 5; ++n)
  {
    num.n1 = n;
    num.n2 = 5n;
    num.n3 = 5n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, fptr);
  }
  fclose(fptr);
  return 0;
}
```

Reading from a binary file

Function fread() also take 4 arguments similar to fwrite() function as above.

fread(address_data,size_data,numbers_data,pointer_to_file);

Example:

```
#include <stdio.h>
struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.bin","rb")) == NULL)
  {
       printf("Error! opening file");      // Program exits if the file pointer returns NULL.
       exit(1);
   }
   for(n = 1; n < 5; ++n)
   {
      fread(&num, sizeof(struct threeNum), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
   }
   fclose(fptr);
   return 0;
}
```

3.4 APPENDING DATA TO EXISTING FILES

C programming supports different file open mode to perform different operations on file. To append data into a file you can use a file open mode.

Step by step descriptive logic to append data into a file.

- Input file path from user to append data, store it in some variable say filePath.
- Declare a FILE type pointer variable say, fPtr.
- Open file in a (append file) mode and store reference to fPtr using fPtr = fopen(filePath, "a");.
- Input data to append to file from user, store it to some variable say dataToAppend.
- Write data to append into file using fputs(dataToAppend, fPtr);.
- Finally close file to save all changes. Use fclose(fPtr);.

Example

```
#include <stdio.h>
#include <stdlib.h>
#define BUFFER_SIZE 1000
void readFile(FILE * fPtr);
int main()
{
    FILE *fPtr;
    char filePath[100];
    char dataToAppend[BUFFER_SIZE];
    printf("Enter file path: ");
    scanf("%s", filePath);
    fPtr = fopen(filePath, "a");
    if (fPtr == NULL)
    {
        printf("\nUnable to open '%s' file.\n", filePath);
        printf("Please check whether file exists and you have write privilege.\n");
        exit(EXIT_FAILURE);
    }
    printf("\nEnter data to append: ");
    fflush(stdin);          // To clear extra white space characters in stdin
    fgets(dataToAppend, BUFFER_SIZE, stdin);
    fputs(dataToAppend, fPtr);
    fPtr = freopen(filePath, "r", fPtr);
    printf("\nSuccessfully appended data to file. \n");
    printf("Changed file contents:\n\n");
    readFile(fPtr);
    fclose(fPtr);
    return 0;
}
void readFile(FILE * fPtr)
{
    char ch;
    do
    {
        ch = fgetc(fPtr);
        putchar(ch);
    } while (ch != EOF);
}
```

3.5 WRITING AND READING STRUCTURES USING BINARY FILES

fwrite() function used to write structure on to the file in a binary format

Example:

```
#include<stdio.h>
#include<conio.h>
struct person
{
    int id;
    char fname[20];
    char lname[20];
};

int main ()
{
    FILE *outfile;
    outfile = fopen ("person.dat", "w");
    if (outfile == NULL)
    {
        fprintf(stderr, "\nError opend file\n");
        exit (1);
    }

    struct person input1 = {1, "rohan", "sharma"};
    struct person input2 = {2, "mahendra", "dhoni"};
    fwrite (&input1, sizeof(struct person), 1, outfile);
    fwrite (&input2, sizeof(struct person), 1, outfile);
        if(fwrite != 0)
        printf("contents to file written successfully !\n");
    else
        printf("error writing file !\n");
    return 0;
}
```

fread() function used to read a structure from a file in binary format.

Example:

```
#include<stdio.h>
#include<conio.h>
struct person
{
    int id;
    char fname[20];
    char lname[20];
};
 int main ()
{
    FILE *infile;
    struct person input;
     infile = fopen ("person.dat", "r");
    if (infile == NULL)
    {
        fprintf(stderr, "\nError opening file\n");
        exit (1);
    }
     while(fread(&input, sizeof(struct person), 1, infile))
        printf ("id = %d name = %s %s\n", input.id,
        input.fname, input.lname);
    return 0;
}
```

3.6 RANDOM ACCESS USING FSEEK, FTELL AND REWIND FUNCTIONS

There is no need to read each record sequentially, if we want to access a particular record. C supports these functions for random access file processing.

1. fseek()
2. ftell()
3. rewind()

The fseek function

We use the `fseek()` function to move the file position to a desired location.

Syntax:

fseek(fptr,offset,position);

Where, `fptr` is the file pointer. `offset` which is of type `long`, specifies the number of positions (in bytes) to move in the file from the location specified by the `position`.

The `position` can take the following values.

- 0 - The beginning of the file
- 1 - The current position in the file
- 2 - End of the file

Following are the list of operations we can perform using the `fseek()` function.

Operation	Description
fseek(fptr, 0, 0)	This will take us to the beginning of the file.
fseek(fptr, 0, 2)	This will take us to the end of the file.
fseek(fptr, N, 0)	This will take us to (N + 1)th bytes in the file.
fseek(fptr, N, 1)	This will take us N bytes forward from the current position in the file.
fseek(fptr, -N, 1)	This will take us N bytes backward from the current position in the file.

fseek(fptr, -N, 2)	This will take us N bytes backward from the end position in the file.

Example:

```
#include <stdio.h>
int main ()
 {
   FILE *fp;
   fp = fopen("file.txt","w+");
   fputs("This is tutorialspoint.com", fp);
     fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Language", fp);
   fclose(fp);
      return(0);
}
```

Now the content in the file is changed to “This is C Programming Language”

The ftell function

The `ftell()` function tells us about the current position in the file (in bytes).

Syntax:

pos=ftell(fptr);

Where, `fptr` is a file pointer. `pos` holds the current position i.e., total bytes read (or written).

Example:

If a file has 10 bytes of data and if the `ftell()` function returns 4 then, it means that 4 bytes has already been read (or written).

```
#include<stdio.h>
 int main()
{
     FILE *fp = fopen("test.txt","r");
     char string[20];
    fscanf(fp,"%s",string);
     printf("%ld", ftell(fp));
    return 0;
}
```

The rewind function

We use the `rewind()` function to return back to the starting point in the file.

Syntax:

rewind(fptr);

Where, `fptr` is a file pointer.

Example

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("file.txt","r");
while((c=fgetc(fp))!=EOF)
{
printf("%c",c);
}
rewind(fp);//moves the file pointer at beginning of the file
while((c=fgetc(fp))!=EOF)
{
printf("%c",c);
}
fclose(fp);
getch();
}
```

The contents in file.txt is " this is a simple text"
The output can be displayed as "this is a simple textthis is a simple text".

TUTORIAL QUESTIONS

1.
```
#include <stdio.h>
   int main(void) {
      FILE f;
      f = fopen("file","wb");
      printf("%d",f != NULL);
      fclose(f);
      return 0;
   }
```
2.
```
#include <stdio.h>
   int main(void) {
      int i;
      i = fprintf(stdin,"Hello!");
      printf("%d",i == EOF);
      return 0;
   }
```
3.
```
#include <stdio.h>
   int main(void) {
      FILE *f = fopen("file","w");
      char c;
      fputs("12A",f);
      fclose(f);
      f = fopen("file","r");
      fscanf(f,"%c",&c);
      fclose(f);
      printf("%c",c);
      return 0;
   }
```
4. Implement a c program to read the content of five students in a file.

ASSIGNMENT QUESTIONS

SET-1

1. Define file. Expalin how to open and read a file.
2. Discuss about the different modes available for opening a file.

SET-2

1. Explain about fseek( ).
2. Write a ‘C’ program to count the number of characters in a file.

SET-3

1. Explain various standard library functions for handling files.
2. Write a ‘C’ program to create a file contains a series of integer numbers and then reads all numbers of this file and write all odd numbers to other file called odd and write all even numbers to a file called even.

SET-4

1. Discuss file positioning functions in C.
2. What are file streams? Discuss about state of file, opening and closing file with a sample C program.

SET-5

1. Write a program to merge two given files and store in a target file.
2. How to handle errors in file management?

SET-6

1. Expalin about fprintf and fscanf functions.
2. Implement a C program to display the content of file.

SET-7

1. **What Are The Preprocessor Categories?Explain in detail?**
2. **What Are Types Of Preprocessor In C**

IMPORTANT QUESTIONS

1. Define file. Expalin how to open and read a file.
2. Discuss about the different modes available for opening a file.
3. Explain about fseek( ).
4. Write a ‘C’ program to count the number of characters in a file.
5. Explain various standard library functions for handling files.
6. Write a ‘C’ program to create a file contains a series of integer numbers and then reads all numbers of this file and write all odd numbers to other file called odd and write all even numbers to a file called even.
7. Discuss file positioning functions in C.
8. What are file streams? Discuss about state of file, opening and closing file with a sample C program.
9. Write a program to merge two given files and store in a target file.
10. How to handle errors in file management?
11. Expalin about fprintf and fscanf functions.
12. Implement a C program to display the content of file.

13. Define preprocessor commands ? Explain its advantages?
14. #define is used for?
15. **What Are Types Of Preprocessor In C**

OBJECTIVE QUESTIONS

1. The first and second arguments of fopen are____________
2. For binary files, a ___ must be appended to the mode string.
3. If there is any error while opening a file, fopen will return_______
4. When a C program is started, O.S environment is responsible for opening file and providing pointer for that file_____________
5. The value of EOF is_____
6. What is the return value of putchar()___________
7. What is the use of getchar()____________
8. **#include<stdio.h>**i

```
int main()
  {
      FILE *fp;
      fp=fopen("trial", "r");
      return 0;
          }
```

   What does fp point to in the program ?

9. Out of fgets() and gets() which function is safe to use?

10 #include<stdio.h>

```
  int main()
   {
        FILE *fp1, *fp2;
       fp1=fopen("file.c", "w");
       fp2=fopen("file.c", "w");
      fputc('A', fp1);
      fputc('B', fp2);
     fclose(fp1);
     fclose(fp2);
   return 0;
}
```

UNIT-IV

FUNCTIONS

4.1 Designing Structured Programs:

A function is a self contained program segment that carries out a specific, well-defined task.

Every c program can be thought of as a collection of these functions. A large problem has to be split into smaller segments so that it can be efficiently solved. This is where, functions come into the picture. They are actually the smaller segments, which help solve the large problems.

The C language supports **two types** of functions:

1. **Library Functions**
2. **User defined Functions**
    - **The library functions** are pre-defined set of functions. Their task is limited. A user cannot understand the internal working of these functions. The user can only use the functions but can"t change or modify them.

Ex: **sqrt(81)** gives result **9**. Here the user need not worry about its source code, but the result should be provided by the function.

- **The User defined functions** are totally different. The functions defined by the user according to his requirement are called as User defined functions. The user can modify the function according to the requirement. The user certainly understands the internal working of the function. The user has full scope to implement his own ideas in the function. Thus the set of such user defined functions can be useful to another programmer. One should include the file in which the user-defined functions are stored to call the functions in the program.

Ex: Let **square(9)** is user-defined function which gives the result **81**.

Here the user knows the internal working of the square() function, as its source code is visible. This is the major difference between the two types of functions.

Why we use functions?

- If we want to perform a task repetitively then it is not necessary to re-write the particular block of the program again and again. Shift the particular block of statements in a user-defined function. The function defined can be used for any number of times to perform the task.
- Suppose a section of code in a program calculates the simple interest for some specified amount, time and rate of interest. Consider a scenario, wherein later on in the same program the same calculation has to be done for a different amount, rate and time.
- Functions come to rescue here. Rather than writing the same instructions all over again, a function can be written to calculate the simple interest for any specified amount, time and rate. The program control is then transferred to the function, the calculations are performed and the control is transferred back to the place from where it was transferred.
- Using functions large programs can be reduced to smaller ones.

- It is easy to debug and find out the errors in it. It also increases readability.
- Programs containing functions are also easier to maintain, because modifications, if required, can be confined to certain functions within the program. In addition to this program, functions can be put in a library of related functions and used by many programs, thus saving on coding time.
- Facilitates top-down modular programming. In this programming style, the high level of the overall program is solved first while the details of each lower level functions are addressed later.

4.2 User-defined functions-Declaring a function

We know that functions are classified as one of the derived data types in C. So, we can define functions and use them like any other variable in C programs. Therefore we can observe the following similarities between the functions and variables in C.

1. Both function names and variable names are considered as identifiers.
2. Like variables, functions also have types (data types) associated with them.

3. Like variables, function names and their types must be declared and defined before they are used in a program.
   In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. **Function declaration**
   2.Function call
   3.Function defining
   ✓

4.3Signature of function

1. **A function declaration** in C tells the compiler about function name, function parameters and return value of a function. The actual body of the function can be defined separately.Like variable in C, we have to declare functions before their first use in program.

Syntax of Function Declaration:

return type function name(type arg1, type arg2 .....);

For Example

- Declaration of a function with no input parameters
  void printMatrix();
- Declaration of a function with one input parameter and integer return type
  int isEvenNumber(int num);
- Declaration of a function with multiple input parameter and integer return type.
  int getSum(int num1, int num2);
- Declaration of a function with no input parameters and integer return type.
  int getRandomNumber();

Important Points about Function Declaration:

- ✓ Function declaration in C always ends with a semicolon.
- ✓ By default the return type of a function is integer(int) data type.
- ✓ Function declaration is also known as function prototype.
- ✓ Name of parameters are not compulsory in function declaration only their type is required. Hence following declaration is also valid.
  int getSum(int, int);

If function definition is written before main function then function declaration is not required whereas, If function definition is written after main function then we must write function declaration before main function.

2. Function call: A compiler executes the function when a semi-colon is followed by function name. A function can be called simply by using its name like other C statement, terminated by semicolon. A function can be called any number of times. A function can be called from other function, but a function cannot be defined in another function. We can call main ( ) function from other functions. But always a function must and should at least call by main once.

3. Function definition (Function Implementation): It is an independent program module that is specifically written to implement the requirements of the function. It includes the following elements:

- ✓ Function type
- ✓ function name,
- ✓ list of parameters (arguments)
- ✓ local variable declarations
- ✓ body of the functions (list of statements)
- ✓ a return statement.

All the six elements are grouped into two parts, namely,

1. Function header (First three elements)
2. Function body (Last three statements)

The function structure: The general syntax of a function in C is:

```
type_specifier function_name (arguments)
{
```

```
            local variable declaration;
            statement-1;
            statement-2;
            ...................
            return (value);
    }
```

- **The type_specifier** specifies the data type of the value, which the function will return. If no data type is specified, the function is assumed to return an **integer** result. The arguments are separated by commas. A function is not returning anything; then we need to specify the return type as void. A pair of empty parentheses must follow the function name if the function definition does not include any arguments.
- **The parameters list / arguments** list declares the variables that will receive the data sent by the calling program. They serve as input data to the function o carry out the specific task.

4.4Parameters and return type of a function:

4.4.1 Parameters:

1. **Actual parameters:** The arguments of calling functions are called as actual parameters.
2. **Formal parameters:** The arguments of called functions are called as formal parameters.
   Note:A semicolon is not used at the end of the function header.

The **body of the function** may consist of one or many statements necessary for performing the required task. The body consists of three parts:

1. Local variable declaration
2. Function statements that perform the task of the function
3. A return statement that returns the value evaluated by the function. If a function is not returning any value then we can omit the return statement.

EX: 1. /* program to call main function from another user defined function*/

```
void main()
{
message();
}
void message ( )
{
printf("\n welcome to C lab");
main();

}
```

EX: 2/* Program to show how user defined function is called */

```
#include<stdio.h>
int add(int,int);//declaration
void main()
{
int x=1,y=2,z;
z=add (x,y);//calling
printf("z=%d",z);
}
add (int a,int b)  //defining
{
return (a+b);
}
```

4.4.2 Return values and their types:

The value evaluated by any function is send back to the calling function by using return statement. The called function can only return only one value per call. The return statement has the following forms:

- return;
- return(expression);

- The first form does not return any value to the calling function; it acts as the closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function.

- A function may have more than one return statement. This situation arises when the value returned is based on certain conditions:

EX:

```
if(a>b)
return(1);
else
return(0);
```

Note:

- All functions by default return integer type data. But, we can force a function to return a particular type of data by using a type specifier in the function header as discussed earlier.

- Absence of return statement in called function indicates that no value is returned to the calling function, such functions are called as void.

How function works?

- Once a function is defined and called, it takes some data from the calling function and returns a value to the called function.

- Whenever a function is called, control passes to the called function and working of the calling function is stopped. When the execution of the called function is completed, control returns back to the calling function and execute the next statement.

- The values of actual arguments passed by the calling function are received by the formal arguments of the called function. The number of actual and formal arguments should be the same.

- Extra arguments are discarded if they are defined. If the formal arguments are more than the actual arguments then the extra arguments appear as garbage. Any mismatch in data type will produce the unexpected result.

- The function operates on formal arguments and sends back the result to the calling function. The return ( ) statement performs this task.

4.5 Passing parameters to function:

Depending upon the arguments present, return value send the result back to the calling function. Based on this, the functions are divided into 4 types:

1. Without arguments and return values.
2. Without arguments and but with return values.
3. With arguments but without return values.
4. With arguments and return values.

Without arguments and return values

Calling function	Analysis	Called function

main()
{
....
....
add ( );
....
....
}
	No arguments are passed.
	
	No values are sent back
		add ( )
		{
		....
		....
		....
		....
		}

In this type neither the data is passed through the calling function nor is the data sent back from the called function.

There is no data transfer between calling function and called function.

If the functions are used to perform any operations, they act independently. They read data values and print result in the same block.

This type of functions may be useful to print some messages, draw a line or split the line etc.

Ex: /* program to display message using user-defined function*/ void main ( )

```
{
void message ( );        /*FUNCTION PROTOTYPE*/
message ( );     /*FUNCTION CALL*/
}
void message ( )         /*FUNCTION DEFINITION*/
{
printf("Have a nice day");       }
```

Explanation: This program contains a user-defined function named message ( ). It requires no arguments and returns nothing. It displays only a message when called.

Without arguments and but with return values:
/* program to receive values from the user defined function without passing any value through main ( )*/

```
void main( )
{
```

```
int z;
z=add ( );        /* FUNCTION CALL*/
print("sum = %d",z);
}
add ( )  /* FUNCTION DEFINITION*/
{

int a=2, b=3, y;
y=a+b;
return (y);
}
```

Calling function	Analysis	Called function

main()
{
int z;
 	
z=add();
printf("sum = %d",z);
}
	
	No arguments are passed.
	No values are sent back
		add ( )
		{
		int a=2,b=3,y;
		
		y=a+b;
		return(y);
		}

In this type of function no arguments are passed through the main ( ) function (calling function). But the called function (add ( )) returns the values. Here the called function is independent. It re data from the keyboard or generates form the initialization and returns the values. In this type both the calling function and called function are partly communicated with each other.

With arguments but without return values

In this type of functions arguments are passed to the calling function. The called function operates on the values. But no result is sent back. This called function is also partly dependent on the calling function. The result obtained is utilized in the called function only.

Ex: /* program to calculate sum of two numbers using user-defined function*/

```
void main ( )

{
void add (int, int); /* FUNCTION PROTOTYPE */
int a, b;            add (int x,int y) /* FUN DEFINITION*/
clrscr( );           {
printf("enter the values of a and b");  int z; /*Local variable declaration*/
scanf("%d%d", &a,&b);        z = x+y;
add(a,b);           /* FUNCTION CALL */        printf("sum = %d",z);
getch( );            }
}
```

Explanation: in this program two values are passed to add ( ) function. The add ( ) function receives argument from main ( ) and displays sum of a and b. But it returns nothing.

Calling function	Analysis	Called function

void main()
{
int a,b;
printf("enter the value of a & b");
scanf("%d %d", &a, &b);
add (a,b);
}
	□Arguments are passed.
	No values are sent back □
		add (int x, int y)
		{
		
		int z; //Local variable
		z = x + y;
		printf(" sum = %d",z);
		}

With arguments and return values:

In this type of functions a copy of actual argument is passed to the formal argument from the calling function to the called function. Called function operates on those data values and it will return the result to the calling function. Here data is transferred between calling and called functions.

```
Ex: /* program to calculate sum of two numbers using user-defined function */
void main()
{
void add(int, int); /*Function prototype declaration */
int a, b, z;
clrscr( );
printf("enter the values of a and b");
scanf("%d %d", &a, &b);
z=add (a,b);  /* Function CALL*/
printf("sum = %d", z);
getch();
}
add (int x, int y)   /* Function Definition */
{
return (x+y);
}
```

Calling function	Analysis	Called function

void main()
{
int a, b, z;
printf("enter the value of a & b");
scanf("%d %d", &a, &b);
z = add (a,b);
printf(" sum = %d", z);
getch( );
}
	□Arguments are passed.
	values are send back □
		add (int x, int y)
		{
		
		return (x + y);
		
		}

Data transfer between the functions / Parameters Passing:

The technique of passing data from one function to another is known as parameters passing. Parameter passing can be done in two ways:

Call by value
Call by reference

4.6 Call by value:

In this type value of actual arguments are passed to the formal arguments and the operation is done on the formal arguments. *Any change made in the formal arguments does not affect the actual arguments because formal arguments are photocopy of actual arguments.*

Changes made in the formal arguments are local to the block of the called function. Once the control returns back to the calling function the changes made vanish.

```
/* Example program to send values by call by value*/
void main( )
{
int a=2,b=3;
void display(int, int);
clrscr( );
display(a, b);
printf("\n\n In main \t a=%d \t b=%d", a, b);
getch( );
}

void display (int x, int y)
{
printf("\n In display( ) before change \t a=%d \t b=%d\n", x,y);
x=10;
y=12;
printf("\n In display( ) after change \t a=%d \t b=%d \n", x, y);
}
```

```
/* Example program on swapping of 2 numbers by using call by value */ void swap(int,int);

void main( )
{
int a,b;
printf("enter any two values\n");
scanf("%d%d",&a,&b);
swap(a,b);
}
void swap(int x,int y)
{
int t;
t=x;
x=y;
y=t;
printf("after interchange\n");
printf("%d %d",x,y);
}
```

4.7 Call by reference:

In this type instead of passing values, addresses (reference) are passed. Function operates on addresses rather than values. Here the formal arguments are pointers to the actual arguments. Formal arguments point to the actual arguments. Hence the changes made in the arguments are permanent. i.e., *the changes made in formal arguments will affect the actual arguments.*

```
/* Example program to send values by call by reference*/
void main( )
{
int a=2,b=3;
void display(int*, int*);
clrscr();
display(&a, &b);
printf("\n\n in main \t a=%d \t b=%d", a, b);
getch();
}
void display (int *x, int *y)
{
printf("\n in display( ) before change \t a=%d \t b=%d\n", *x, *y);
*x=10;
*y=12;
printf("\n in display( ) after change \t a=%d \t b=%d \n", *x, *y);
}

/* Ex 2: Swapping of 2 numbers */
#include<stdio.h>

#include<conio.h>
void swap(int *,int *);
void main( )
{
int a,b;
printf("enter any two values\n");
scanf("%d%d",&a,&b);
swap(&a,&b);
printf("after interchange\n");
printf("%d %d",a,b);
getch( );
}
void swap(int *x,int *y)
{
int t;
t=*x;
*x=*y;
*y=t;
}
```

Function returning more values:

We know that a function can return only one value per call. But we can force the function to return more than one value per call by using call by reference.

```
void main( )
{
int x,y, add, sub, change (int*,int*,int*,int*);
clrscr();
printf("\n enter values of x&y");
scanf("%d %d", &x, &y);
change(&x, &y, &add, &sub);
printf("\n Addition=%d \n Subtraction=%d", add, sub);
getch();
}
change(int *a, int *b, int *c, int *d)
{
*c=*a+*b;
*d=*a-*b;
}
```

Explanation: In this program return statement is not used. Still function returns more than one value. Actually, no values are returned. Once the addresses of the variables are available, we can directly access them and modify their contents.

Note:
The memory address of any variable is unique.

If we declare the same variable for actual and formal arguments, their memory addresses will be different from each other.

4.8 Standard functions and libraries

In „c‟, a number of pre-defined functions are available to perform various tasks. To use these functions, we have to include the corresponding header file in which the function is available.

stdio.h:- (standard input output library functions)

When any of the functions getchar ( ), qets ( ), putchar ( ), puts ( ), scanf( ), printf ( ) is used the header file stdio.h has to be included.

math.h:- (mathematical functions)

pow ( )*:*-This function returns xn value. Syntax:- pow (x,n);

Eg:- pow (5,3)=5^3=125.

sqrt ( )*:*-This function performs square root of the given number. Syntax:- sqrt (n);

Eg:- sqrt (81)=9.

log ( ):-This function returns natural logarithm of the given number. Syntax:- log (n);

Eg:- log (8);

log10 ( ):-This functions returns logarithm value of the given number to the base 10.

Syntax:- log10 (n);
Eg:- log10(10)=1

exp( ):-This function returns er value.
Syntax:- exp (x);
Eg:- exp(3);

ceil ( ):-This function returns the next higher integer value of the given number. Syntax:- ceil(n);
Eg:- ceil (17.7)=18 ceil (16.1)=17.

floor( ):-This function returns the integer value less than or equal to the given number.

Syntax:- floor (n);

Eg:- floor (17.7) = 17 floor(16.1) = 16.

Cos, acos, cosh, sin, asin, sinh, tan, atan, tanh are also the functions under this header file.

Stdlib.h:- (standard Library functions header file)

abs( ):-This functions returns the absolute value of a given, integer. Syntax:- abs (integer value);

Eg:- abs (-17); =17.

***fabs( )**:-*This functions returns absolute value (modulus) of a given floating point number.

Syntax:- fabs (float value);
Eg:- fabs (-17.6) = 17.6.

atoi ( ):-This function converts the given string to an integer value. Syntax:- atoi(string);

Eg:- atoi ("123") =123.

atof ( ):-This function converts the given string into floating point value. Syntax:- atof (string);

Eg:- atof (" 123.56") = 123.560000.

ctype. h:- (character testing and conversion functions)

(i*) isalpha( ):-*This function checks whether the given character is an alphabet (or) not. If it is an alphabet, it returns a non-zero value and otherwise a zero value.

Syntax:- is alpha(„a‟); True
Eg:- isalpha (“a‟) True (non-zero)
isalpha(“2‟) Flase (zero)

isalnum( ):- This function checks whether the given character is an alphabet or a number. If true it returns a non-zero value otherwise a zero value.
Eg:- isalnum(“1‟) True (non-zero)
isalnum(“q‟) False (zero)

isdigit ( ):-This function checks whether the given character is a digit or not. If true it returns a non-zero value otherwise a zero value.
Eg:- isdigit (“a‟) True (non zero)
isdigit(“*‟) False (zero)

islower ( ):-This function checks whether the given character is a Lower case alphabet or not. If it is a small letter it returns a non-zero value otherwise a zero value.
Eg:- islower (“b‟) True (non-zero)
islower(“A‟) False (zero)

isupper ( ):-This function checks whether the given character is a upper case alphabet or not. If it is a capital letter is returns a non-zero value otherwise a zero value.
Eg: isupper (“B‟) True (non-zero)
Isupper(”q‟) False (zero)
toupper ( ):-This function converts the given small letters to an upper case letter.

Eg:- toupper (“b‟)=B
toupper(“q‟)=Q

tolower ( ):- This function converts the given capital letters to a Lower case letter.

Eg;- tolower (“B‟) =b
tolower (“Q‟)=q

toascii( ):-This function returns the equivalent ASCII value for the given character.

Eg:- toascii (“a‟) = 97
toascii(“B‟) = 66

COMMONLY USED LIBRARY FUNCTIONS:
clrscr( ):

This function clears the previous output from the screen and displays the output of the current program from the first line of the screen. This function is defined in **conio.h** header file.

Syntax: clrscr();

exit( ):

This function terminates the program. It is defined in the **process.h** header file.

Syntax: exit( );

4.9 Recursion

Recursion is a special case of process, where a function calls itself. A function is called recursive if a statement within the body of a function calls the same function.

```
factorial(x)
int x;
{
if (x = =1)
return(1);
else
return(x * factorial(x-1));
}
```

When writing recursive functions, you must have an If stmt somewhere in the recursive function to force the function to return without recursive call being executed. If you do not do this and you call the function, you will fall in an indefinite loop, and will never return from the called function.

```
/*program to find factorial of a given number using a Recursive function*/ #include<stdio.h>
int factorial(x)
int x;
{
if (x<=1)
return(1);
else
return(x*factorial(x-1));
}
main()
{
int n,fn;
clrscr();
printf("enter n");
scanf("%d",&n);
fn=factorial(n);          /* Function Call */
printf("the factorial %d is %d\n",n,fn);
getch();          }
```

In case the value of n is 4, main() would call factorial() with 4 as its actual argument, and factorial() will send back the computed value. But before sending the computed value, factorial() calls factorial() and waits for a value to be returned.

factorial(4) returns 4*factorial(3)	4*6	= 24
factorial(3) returns 3*factorial(2)	3*2	= 6
factorial(2) returns 2*factorial(1)	2*1	= 2
factorial(1) returns1	Back substitution	

```
/* A Program to find factorial of a given number using recursion. */
#include<stdio.h>
#include<conio.h>
int factorial(int);
void main( )
{
int n,k;
printf("enter any number\n");
scanf("%d",&n);
k=factorial(n);
printf("factorial of %d is %d",n,k);
getch( );
}
int factorial(int x)
{
int fact;
if(x==0||x==1)
return(1);
else
{
fact=(x*factorial(x-1));
return(fact);
}
}
/* A Program to print Fibonacci series using recursion. */
#include<stdio.h>
#include<conio.h>
int fib(int);
void main()
{
int i,n;
printf("enter limit\n");
scanf("%d",&n);
for(i=1;i<=n;i++)
printf("%d ",fib(i));
getch( );
}

int fib(int x)
```

```
{
if(x==1)
return 0;
else if(x==2)
return 1;
else
return(fib(x-1)+fib(x-2));
}
```

4.9.1 Limitations of Recursion:

Advantages

1. Reduce unnecessary calling of function.
2. Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex.

Disdvantages

1. Recursive solution is always logical and it is very difficult to trace.(debug and understand).
2. In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
3. Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
4. Recursion uses more processor time.

Dynamic memory allocation: Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is running. It effectively postpones the data definition ,but not the declaration to run time.

To use dynamic memory allocation ,we use either standard data types or derived types .To access data in dynamic memory we need pointers.

4.10 MEMORY ALLOCATION FUNCTIONS:

Four memory management functions are used with dynamic memory. Three of them,malloc,calloc,and realloc,are used for memory allocation. The fourth ,free is used to return memory when it is no longer needed. All the memory management functions are found in standard library file(stdlib.h).

1. **BLOCK MEMORY ALLOCATION(MALLOC) :**

The malloc function allocates a block of memory that contains the number of bytes specified in its parameter. It returns a void pointer to the first byte of the allocated memory. The allocated memory is not initialized.

Declaration:

```
void *malloc (size_t size);
```

The type size_t is defined in several header files including Stdio.h. The type is usually an unsigned integer and by the standard it is guaranteed to be large enough to hold the maximum address of the computer. To provide portability the size specification in malloc's actual parameter is generally computed using the sizeof operator. For example if we want to allocate an integer in the heap we will write like this:

```
Pint=malloc(sizeof(int));
```

Malloc returns the address of the first byte in the memory space allocated. If it is not successful malloc returns null pointer. An attempt to allocate memory from heap when memory is insufficient is known as overflow.

The malloc function has one or more potential error if we call malloc with a zero size, the results are unpredictable. It may return a null pointer or it may return someother implementation dependant value.

Ex:

```
If(!(Pint=malloc(sizeof(int)))) //
        no memory available
   exit(100);
        //memory available
…
```

In this example we are allocating one integer object. If the memory is allocated successfully,ptr contains a value. If does not there is no memory and we exit the program with error code 100.

2. **CONTIGIOUS MEMORY ALLOCATION(calloc) :**

Calloc is primarily used to allocate memory for arrys.It differs from malloc only in that it sets memory to null characters. The calloc function declaration:

```
Void *calloc(size_t element_count, size_t element_size);
```

The result is the same for both malloc and calloc.

calloc returns the address of the first byte in the memory space allocated. If it is not successful calloc returns null pointer.

Example:

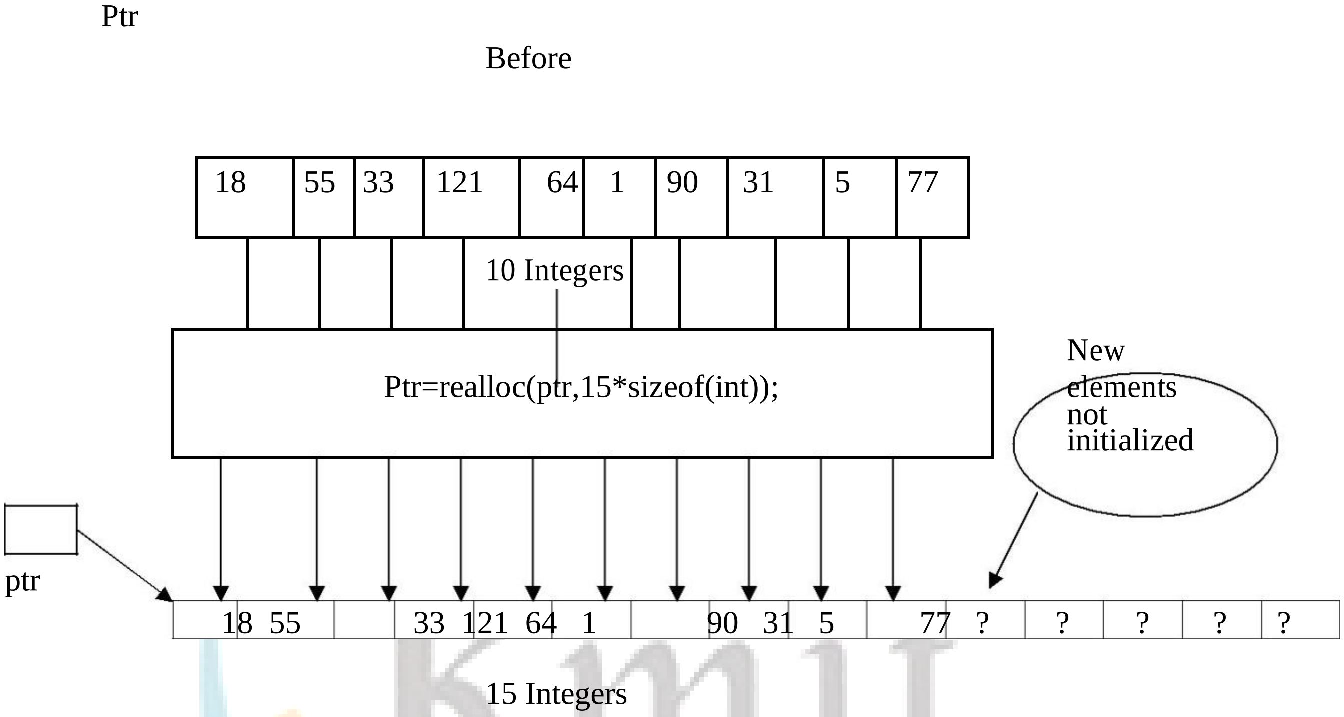
```
If(!(ptr=(int*)calloc(200,sizeof
        (int)))) //no memory
        available
   exit(100);
        //memory available
…
```

In this example we allocate memory for an array of 200 integers.

3. **REALLOCATION OF MEMORY(realloc):**

The realloc function can be highly inefficient and therefore should be used advisedly. When given a pointer to a previously allocated block of memory realloc changes the size of the block by deleting or extending the memory at the end of the block. If the memory can not be extended because of other allocations realloc allocates completely new block,copies the existing memory allocation to the new location,and deletes the old allocation.

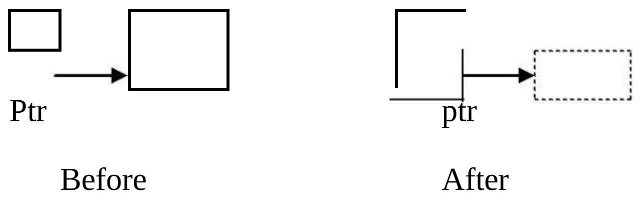
Void *realloc(void*ptr,size_t newsize);



After

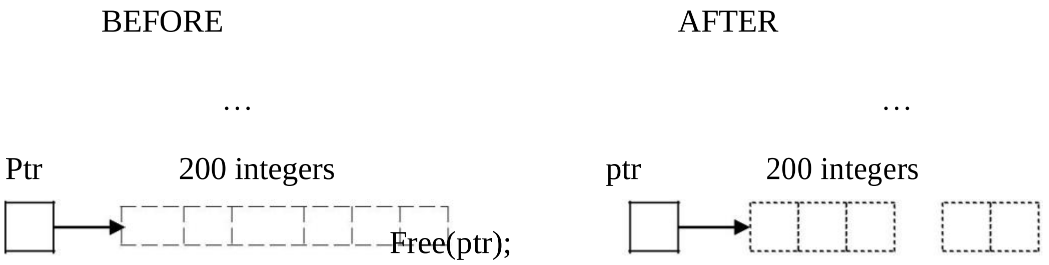
Releasing Memory(free):When memory locations allocated by malloc,calloc or realloc are no longer needed, they should be freed using the predefined function free. It is an error to free memory with a null pointer, a pointer to other than the first element of an allocated block, a pointer that is a different type then the pointer that allocated the memory, it is also a potential error to refer to memory after it has been released.

Void free(void *ptr);



Free(ptr);

In above example it releases a single element to allocated with a malloc,back to heap.



In the above example the 200 elements were allocated with calloc. When we free the pointer in this case, all 200 elements are return to heap. First, it is not the pointers that are being released but rather what they point to. Second , To release an array of memory that was allocated by calloc , we need only release the pointer once. It is an error to attempt to release each element individually.

Releasing memory does not change the value in a pointer. Still contains the address in the heap. It is a logic error to use the pointer after memory has been released.

4.11 Array of pointers:

Array are closely related to pointers in C programming but the important difference between them is that, a pointer variable takes different addresses as value whereas, in case of array it is fixed.

This can be demonstrated by an example:

```
#include <stdio.h>
int main()
{
  char charArr[4];
  int i;

  for(i = 0; i < 4; ++i)
  {
    printf("Address of charArr[%d] = %u\n", i, &charArr[i]);
  }

  return 0;
}
```

When you run the program, the output will be:

```
Address of charArr[0] = 28ff44
Address of charArr[1] = 28ff45
Address of charArr[2] = 28ff46
Address of charArr[3] = 28ff47
```

Note: You may get different address of an array.

Notice, that there is an equal difference (difference of 1 byte) between any two consecutive elements of array *charArr*.

But, since pointers just point at the location of another variable, it can store any address.

Relation between Arrays and Pointers

Consider an array:

```
int arr[4];
```

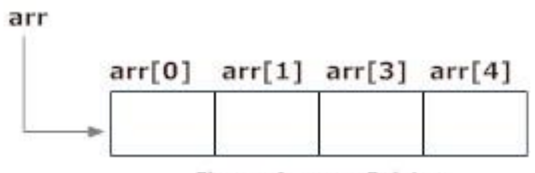


Figure: Array as Pointer

- In C programming, name of the array always points to address of the first element of an array.
- In the above example, *arr* and *&arr[0]* points to the address of the first element. &arr[0] is equivalent to arr
- Since, the addresses of both are the same, the values of arr and &arr[0] are also the same.
- arr[0] is equivalent to *arr (value of an address of the pointer)
- Similarly, &arr[1] is equivalent to (arr + 1) AND, arr[1] is equivalent to *(arr + 1).
- &arr[2] is equivalent to (arr + 2) AND, arr[2] is equivalent to *(arr + 2).
- &arr[3] is equivalent to (arr + 3) AND, arr[3] is equivalent to *(arr + 3)..
- &arr[i] is equivalent to (arr + i) AND, arr[i] is equivalent to *(arr + i).

In C, you can declare an array and can use pointer to alter the data of an array.

Example: Program to find the sum of six numbers with arrays and pointers

```
#include <stdio.h>
int main()
{
  int i, classes[6],sum = 0;
  printf("Enter 6 numbers:\n");
  for(i = 0; i < 6; ++i)
  {
      // (classes + i) is equivalent to &classes[i]
      scanf("%d",(classes + i));

      // *(classes + i) is equivalent to classes[i]
      sum += *(classes + i);
  }
  printf("Sum = %d", sum);
  return 0;
}
```

TUTORIAL QUESTIONS

1. WACP to Check given number is Prime and Armstrong Number or not using function.
2. Implement a c program to print 1 to 100 numbers without using loops.
3. Implement a C program to multiply three numbers by passing it as arguments to a function.
4. Implement a c program to reverse string using function
5. Implement a c program to find GCD using recursion

ASSIGNMENT QUESTIONS

SET 1:

1. Differentiate call by value and call by reference.
2. Implement a C program to print factorial of given number using with arguments and with return type.

SET 2:

1. Discuss recursive function with an example.
2. Implement a C program to find power and square root of any number using standard functions.

SET 3:

3. Implement a c program to transpose the given matrix.
4. Implement a C program to factorial using recursive functions.

SET 4:

1. Discuss about storage classes.
2. Implement a C program to allocate memory for 10 integers dynamically.

SET 5:

3. Discuss about inter function communications.
4. List out the difference between malloc() and calloc()?

SET 6:

1. Explain the difference between static allocation and dynamic allocation?
2. Implement Fibonacci series using recursion.

Important questions

1. Write a program to demonstrate passing an array argument to a function? Consider the problem of finding largest of N numbers defined in an array.

   (b) Write a recursive function power(base, exponent) that when invoked returns base exponent?

2. What do you mean by functions? Give the structure of the functions and explain about

the arguments and their return values?

3. (a) Distinguish between the following.

(i) Actual and Formal arguments?

(ii) Global and Local variables?

(iii) Automatic and Static variables?

(c) Explain in detail about pass by value and pass by reference. Explain with a sample program?

4. (a) Distinguish between formal variable and actual variable.

(b) Distinguish between Local and Global variable.

(c) Distinguish between Call by value and Call by reference.

5. (a) What are the advantages and disadvantages of recursion?

(b) Write a C program to find the factors of a given integer using a function.

6. (a) Give some important points while using return statement.

(b) Write short notes on scope of a variable.

7. (a) Write short notes on auto and static storage classes.

(b) Write short notes on call by reference.

8. (a) Distinguish between user defined and built-in functions.

(i) What is meant by function prototype? Give an example for function prototype.

9. (a) Distinguish between getchar and scanf functions for reading strings.

(i) Write a program to count the number of words, lines and characters in a text.

10. (a) What do you mean by functions? Give the structure of the functions and explain about the arguments and their return values.

(i) Write a C program that uses a function to sort an array of integers.

UNIT-V

INTRODUCTION TO ALGORITHMS

5.1 ALGORITHMS

5.1 .1 Algorithm for finding roots of quadratic equations:

```
Step 1: Start
Step 2: Read a,b,c
Step3: calculate disc=b*b-4*a*c
Step 4: if(disc>0)
Begin
Step 5:root1=(-b+sqrt(disc))/(2*a)
Step 6:root2=(-b-sqrt(disc))/(2*a)
Step7:Print―Root1‖, ―Root2‖
End
Step 8: else if(disc=0)Begin
Step 9:root1=-b/(2*a)
Step 10:root2=root1;
Step11:Print―Root1‖,―Root2‖
End Step 12: else
Step 13: Print Roots are imaginary
Step 14: Stop
```

5.1.2 Algorithm for finding maximum numbers of a given set:

```
Step 1: Start

Step 2: Read n elements a[1:n]

Step 3: Assign min=max=a[0]

Step 4: for i=1 to n do {if i reaches to n goto step 7}

Step 5:if (a[i]>max) goto step 6 otherwise goto step 4

Step 6:max=a[i] and goto step 4

Step 7: Print max.

Step 8: stop
```

5.1.3 Algorithm to check given number is prime or not:

```
Step 1: Start
Step 2: Declare variables n,i,flag.
Step 3: Initialize variables
        flag←1
        i←2
```

```
Step 4: Read n from user.
Step 5: Repeat the steps until i<(n/2)
    5.1 If remainder of n÷i equals 0
           flag←0
           Go to step 6
    5.2 i←i+1
Step 6: If flag=0
          Display n is not prime
       else
          Display n is prime
Step 7: Stop
```

5.2 Basic searching in an array of elements

Searching:
Searching refers to an operation of finding the location of an item in a table or a file. Depending on the location of the records to be searched, searching techniques are classified into two types.

External Searching:
When the records are stored in files, disk, tape any secondary storage, then the searching is known as external searching.

Internal Searching:
When the records are stored in main memory, then it is known as internal searching. *In C language we have 2 types of searching techniques,*

5.2.1 Linear Search

This is simplest search technique also known as sequential search. In this method, the array is searched for the required element from the first element onward till either the list is exhausted or the required element is found. It doesn't required sorted list.

The key which is to be searched is compared with each element of the list. If a match exists, the search is terminated. If the end of the list is reached, it means that the search has failed and the key has no matching element in the list.

Algorithm: linearsearch (a, n, key)

Step 1: for (i=0;i<n;i++)
Step 2: Take the first element in the list. If the element in the list is equal to the desired searching element, then it returns element position. Go to step 5.
Step 3: If it is not end of list, take the next element in the list. Go to step 2.
Step 4: If the element not found then return negative value.

Step 5: End of algorithm.

```
/*program using linear search*/
#include<stdio.h>
#include<conio.h>
int linear(int a[],int,int);
void main()
```

```
{
int a[20],n,i,key,pos;
clrscr();
printf("enter the no. of elements u want in array\n");
scanf("%d",&n);
printf("enter %d elements\n",n);
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter the key which u wnt to search\n");
scanf("%d",&key);
pos=linear(a,n,key);
if(pos==-1)
printf("element is not found");
else
printf("element is found at %d position",pos+1);
getch();
}
int linear(int a[],int n,int key)
{
int i,k=0;
for(i=0;i<n;i++)
{
if(key==a[i])
{
k=i;
}
}
if(k==0)
return(-1);
else
return(k);
}
```

5.2.2 Binary Search

- To implement binary search method, the elements must be in sorted order. This method is implemented as given below,
- The key is compared with item in the middle position of array.
- If the key matches with item, return it and stop.
- If the key is less than mid positioned item, then the item to be found must be in half of array, otherwise it must be in second half of array.

- Repeat the procedure for lower half or upper half of array until the element is found.

Algorithm: binarysearch (a, n, key) [non-recursive]

```
Step 1: Initialize low = 0 and high = n-1.

Step 2: repeat thru step 4 while low is less than or equal to high i.e., while(low<=high) do

Step 3: Find the mid value using mid=(low+high)/2.

Step 4: [ Compare to search for item ]

if key < a[mid] then

high = mid – 1

otherwise, if key > a[mid] then

low = mid + 1

otherwise if key == a[mid] then return mid+1.

Step 5: Unsuccessful search return -1.

Step 6: End of algorithm.
```

/*program using binary search*/

```
#include<stdio.h>
#include<conio.h>
int binary(int a[],int,int);
void main()
{
int a[20],i,n,key,pos;
clrscr();
printf("enter the no. of elements u want in the list\n");
scanf("%d",&n);
printf("enter the %d elements of array\n",n);
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter the key u want to search\n");
scanf("%d",&key);
pos=binary(a,n,key);
if(pos==-1)
printf("key is not found");
else
printf("key is found at %d position",pos+1);
getch();
}
```

```
int binary(int a[],int n,int key)
{
int low=0,high=n-1,mid;
while(high>=low)
{
mid=(low+high)/2;
if(key>a[mid])
low=mid+1;
else if(key<a[mid])
high=mid-1;
else
return(mid);
}
return(-1);
}
```

5.3 Sorting:

Sorting is a method of arranging data in a file in ascending or descending order. Sorting makes handling of records in a file is easier.

5.3.1 Bubble Sort:

- Bubble sorting is a simple sorting technique in sorting algorithm. It is also known as exchange sort.
- The name bubble sort is because after each pass, largest element will bubble up. In bubble sorting algorithm, we arrange the elements of the list by forming pairs of adjacent elements. It means we repeatedly step through the list which we want to sort, compare two items at a time and swap them if they are not in the ascending order.
- Here are basic steps of Bubble Sort algorithm:

- Compare each pair of adjacent elements from the beginning of an array and, if they are in descending order, swap them.

- If at least one swap has been done, repeat step 1.
- This process will be continued until all the elements in an array are sorted.

Algorithm

```
1. begin
2. for i := 0 to array_size - 1 do
3. begin
4. for j := 0  array_size–1-i do
5. if a [j] > a [j+1] then
6. begin
7. temp := a [j];
8. a [j] := a[j+1];
9. a[j+1] := temp;
10. end
11. end
12. end
```

Working of bubble sort algorithm:

- Bubble sort (to sort in ascending order) is performed as given below, Consider an array of n elements a[0] to a[n-1]
- In first pass, compare a[0] with a[1]. If a[0] > a[1], swap the numbers otherwise leave it.
- In second pass, Compare a[1] with a[2] and swap the numbers if a[1] > a[2] and so on and compare a[n-2] with a[n-1] and swap the number if a[n-2] > a[n-1].
- By this process the first largest element placed in nth position. This element is not further compared in the remaining passes.
- Now consider first n-1 elements in the list and repeat the above process to place the next largest element in the (n-1)th position. Repeat this process until all the elements are placed in proper positions.

Advantages of Bubble Sort:

- It is relatively easy to write and understand.
- It is not used for sorting the list of larger size.
- The performance is good for nearly sorted list.
- Works well for smaller list of elements.
- Disadvantages of Bubble Sort:It is inefficient algorithm because the number of iterations increases with the increase in number of elements to be sorted.
- The time taken for sorting is more than selection and insertion sort.

```
/*program using Bubble Sort*/
#include<stdio.h>
#include<conio.h>
void bubble(int a[20],int);
void main()
{
int a[20],n,i;
clrscr();
printf("enter the no. of elements u want to give in an array\n");
scanf("%d",&n);
printf("enter %d elements\n",n);
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("before sorting,the elements are\n");
for(i=0;i<n;i++)
{
printf("%3d",a[i]);
}
bubble(a,n);
printf("after sorting,the elements are\n");
for(i=0;i<n;i++)
{
printf("%3d",a[i]);
}
getch();
}
void bubble(int a[20],int n)
{
int i,j,temp;
for(i=0;i<n-1;i++)
{
for(j=i;j<n;j++)
{

if(a[i]>a[j])

{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
}
```

5.3.2 Insertion sort

- Insertion sort is implemented by inserting a particular element at the appropriate position.
- In this method, the first iteration starts with comparison of 1st element with the 0th element.
- In the second iteration 2nd element is compared with the 0th and 1st element.
- In general, in every iteration an element is compared with all elements before it.
- During comparison if it is found that the element in question can be inserted at a suitable position then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
- This procedure is repeated for all the elements in the array.

Suppose the sequence is: 25, 17, 31, 13, 2
Step by Step Iteration

I	Present Sequence
I = 0	17 25 31 13 2
I = 1	17 25 31 13 2
I = 2	13 17 25 31 2
I = 3	2 13 17 25 31

Explanation:

a) In the first iteration the 1st element 17 is compared with the 0th element 25. Since 17 is smaller than 25, 17 is inserted at 0th place. The 0th element 25 is shifted one position to the right.
b) In the second iteration, the 2nd element 31 and 0th element 17 is compared. Since, 31 is greater than 17, nothing is done. Then the 2nd element 31 is compared with the 1st element 25.Again no action is taken as 25 is less than 31.
c) In the third iteration, the 3rd element 13 is compared with the 0th element 17.Since, 13 is smaller than 17, 13 is inserted at the 0th place in the array and all the elements from 0th till 2nd position are shifted to right by one position.
d) In the fourth iteration the 4th element 2 is compared with the 0th element 13. Since, 2 is smaller than 13, the 4th element is inserted at the 0th place in the array and all the elements from 0th till 3rd are shifted right by one position. As a result, the array now becomes a sorted array.

/*program using Insertion Sort*/

```
#include<stdio.h>
#include<conio.h>
void insertion(int [], int );
int main()
{
 int arr[30];
 int i,size;
 printf("Enter total no. of elements : ");
 scanf("%d",&size);
printf("\n Enter the elements to sort:");
 for(i=0; i<size; i++)
   scanf("%d",&arr[i]);
 insertion(arr,size);
 printf("\nAfter sorting\n");
 for(i=0; i<size; i++)
   printf(" %d",arr[i]);
 getch();
 return 0;
}
void insertion(int arr[], int size)
{
 int i,j,tmp;
 for(i=0; i<size; i++)
 {
   for(j=i-1; j>=0; j--)
   {
    if(arr[j]>arr[j+1])
    {
      tmp=arr[j];
      arr[j]=arr[j+1];
      arr[j+1]=tmp;
    }
    else
      break;
   }
 }
}
```

5.3.3 Selection Sort

- Suppose that items in a list are to be sorted in their correct sequence. Using Selection Sort, the first item is compared with the remaining n-1 items, and whichever of all is lowest, is put in the first position.
- Then the second item from the list is taken and compared with the remaining (n-2) items, if an item with a value less than that of the second item is found on the (n-2) items, it is swapped (Interchanged) with the second item of the list and so on.

Algorithm for Selection Sort:

```
Procedure Selection_Sort(A,N)
For Index = 1 to N-1 do
begin
 Index←MinPostion
 For j = Index+1 to N do
 begin if A[j] < A[MinPosition] then
  j←MinPosition
 end swap (A[Index], A[MinPosition])
end
Return
```

/*program using Selection Sort*/

```
#include <stdio.h>
void selection_sort();
int a[30], n;
void main()
{
    int i;
    printf("\nEnter size of an array: ");
    scanf("%d", &n);
    printf("\nEnter elements of an array:\n");
    for(i=0; i<n; i++)
    scanf("%d", &a[i]);
    selection_sort();
    printf("\n\nAfter sorting:\n");
    for(i=0; i<n; i++)
    printf("\n%d", a[i]);
    getch();
}
void selection_sort()
{
    int i, j, min, temp;
    for (i=0; i<n; i++)
    {
        min = i;
        for (j=i+1; j<n; j++)
        {
            if (a[j] > a[min])
                min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

5.4 Order of complexity

Algorithm	**Time Complexity**		
	Best	**Average**	**Worst**
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)

TUTORIAL QUESTIONS

1. Write a program to explain bubble sort. Which type of technique does it belong. (b) What is the worst case and best case time complexity of bubble sort?
2. Explain the algorithm for bubble sort and give a suitable example. (OR) Explain the algorithm for exchange sort with a suitable example.
3. Write a program to explain insertion sort . Which type of technique does it belong. (or) Write a C-program for sorting integers in ascending order using insertion sort.
4. Explain the algorithm for insertion sort and give a suitable example
5. Demonstrate the insertion sort results for each insertion for the following initial array of elements . 25 6 15 12 8 34 9 18 2
6. Demonstrate the selection sort results for each pass for the following initial array of elements . 21 6 3 57 13 9 14 18 2

ASSIGNMENT QUESTIONS

SET-1:

1. Design an algorithm and flow chart to find maximum among three numbers.
2. Implement a C program to find roots of quadratic equation

SET-2:

1. Design an algorithm and flow chart to check given number is positive or negative or zero
2. Implement a C program to check given number is prime or not.

SET-3:

1. Implement a C program to check given number is Armstrong or not.
2. Design an algorithm and flow chart to convert Celsius to Fahrenheit and Fahrenheit to Celsius.

SET-4:

1. Implement a C program to check given number is perfect number or not.
2. Explain selection sort with example.

SET-5:

1. Implement a C program to search a key using binary search.
2. Illustrate the complexity of bubble sort , selection sort and insertion sort.

IMPORTANT QUESTIONS

1. What is a flowchart? Explain the different symbols used in a flowchart.

2. (a) Define an Algorithm?

(d) What is the use of Flowchart?

(e) What are the different steps followed in the program development?

3. Write a program to explain selection sort. Which type of technique does it belong.
4. Explain the algorithm for selection sort and give a suitable example.
5. Formulate recursive algorithm for binary search with its timing analysis
6. Write a C program that searches a value in a stored array using linear search.
7. Implement a C program to find prime number from 1 to n.
8. Implement Fibonacci series using recursion
9. WACP to sort elements using insertion sort.
10. Compute and analysis time and space complexity.

.

OBJECTIVE QUESTIONS

1) Finding the location of a given item in a collection of items is called ……

A. Discovering

B. Finding

C. Searching

D. Mining

2) Which of the following is an external sorting?

A. Insertion Sort

B. Bubble Sort

C. Merge Sort

D. Tree Sort

3). Selection sort first finds the .... element in the list and put it in the first position.

A. Middle element

B. Largest element

C. Last element

D. Smallest element

4) The total number of comparisons in a bubble sort is ....

A. O(n logn)

B. O(2n)

C. O(n2)

D. O(n)

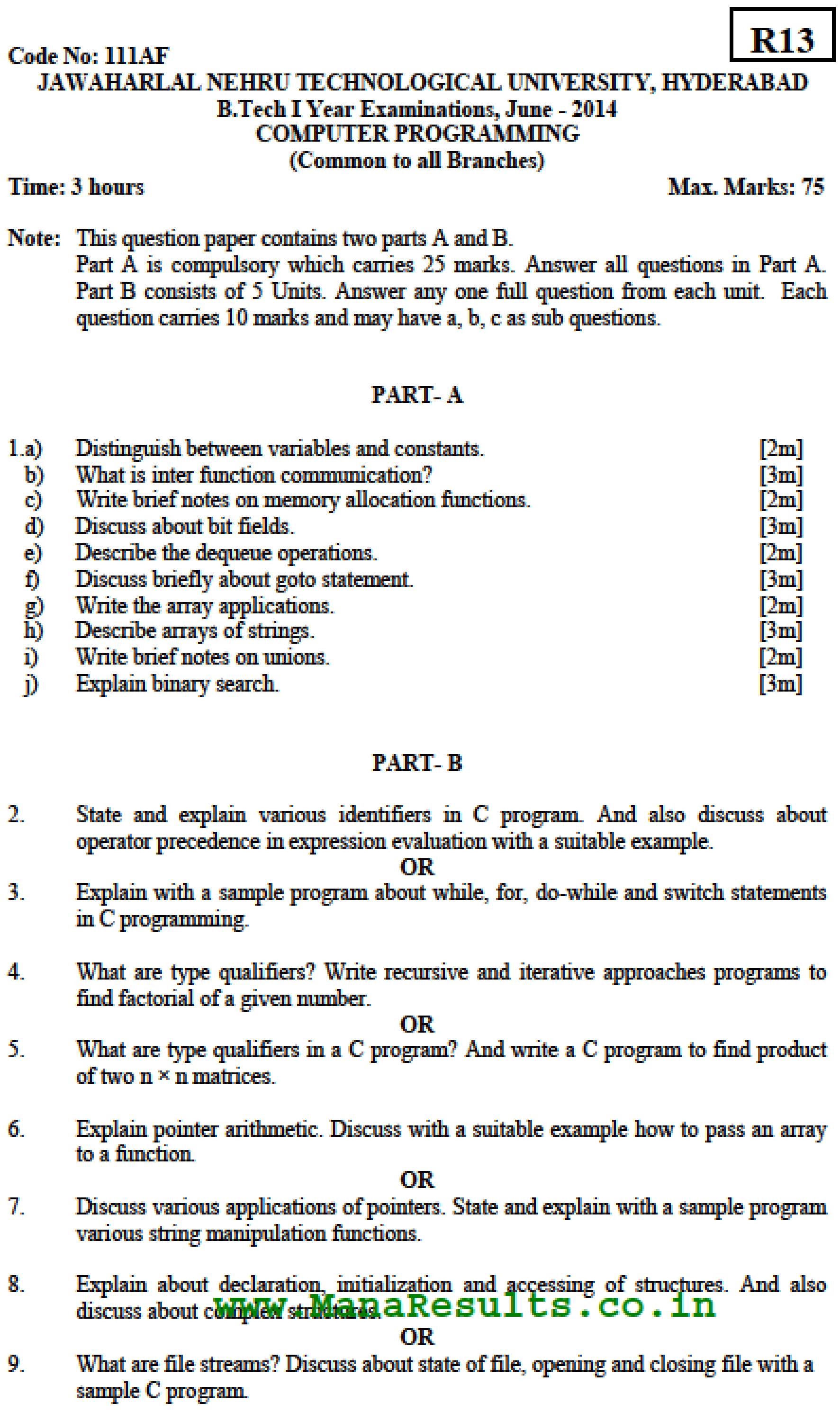
5) **Which of the following is not a stable sorting algorithm?**

a) Insertion sort

b) Selection sort

c) Bubble sort
d) Merge sort

EXTERNAL QUESTION PAPERS



R13

Code No: 111AF

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY, HYDERABAD
B.Tech I Year Examinations, June - 2014
COMPUTER PROGRAMMING
(Common to all Branches)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART- A

1.a)	Distinguish between variables and constants.	[2m]
b)	What is inter function communication?	[3m]
c)	Write brief notes on memory allocation functions.	[2m]
d)	Discuss about bit fields.	[3m]
e)	Describe the dequeue operations.	[2m]
f)	Discuss briefly about goto statement.	[3m]
g)	Write the array applications.	[2m]
h)	Describe arrays of strings.	[3m]
i)	Write brief notes on unions.	[2m]
j)	Explain binary search.	[3m]

PART- B

2. State and explain various identifiers in C program. And also discuss about operator precedence in expression evaluation with a suitable example.

OR

3. Explain with a sample program about while, for, do-while and switch statements in C programming.

4. What are type qualifiers? Write recursive and iterative approaches programs to find factorial of a given number.

OR

5. What are type qualifiers in a C program? And write a C program to find product of two n × n matrices.

6. Explain pointer arithmetic. Discuss with a suitable example how to pass an array to a function.

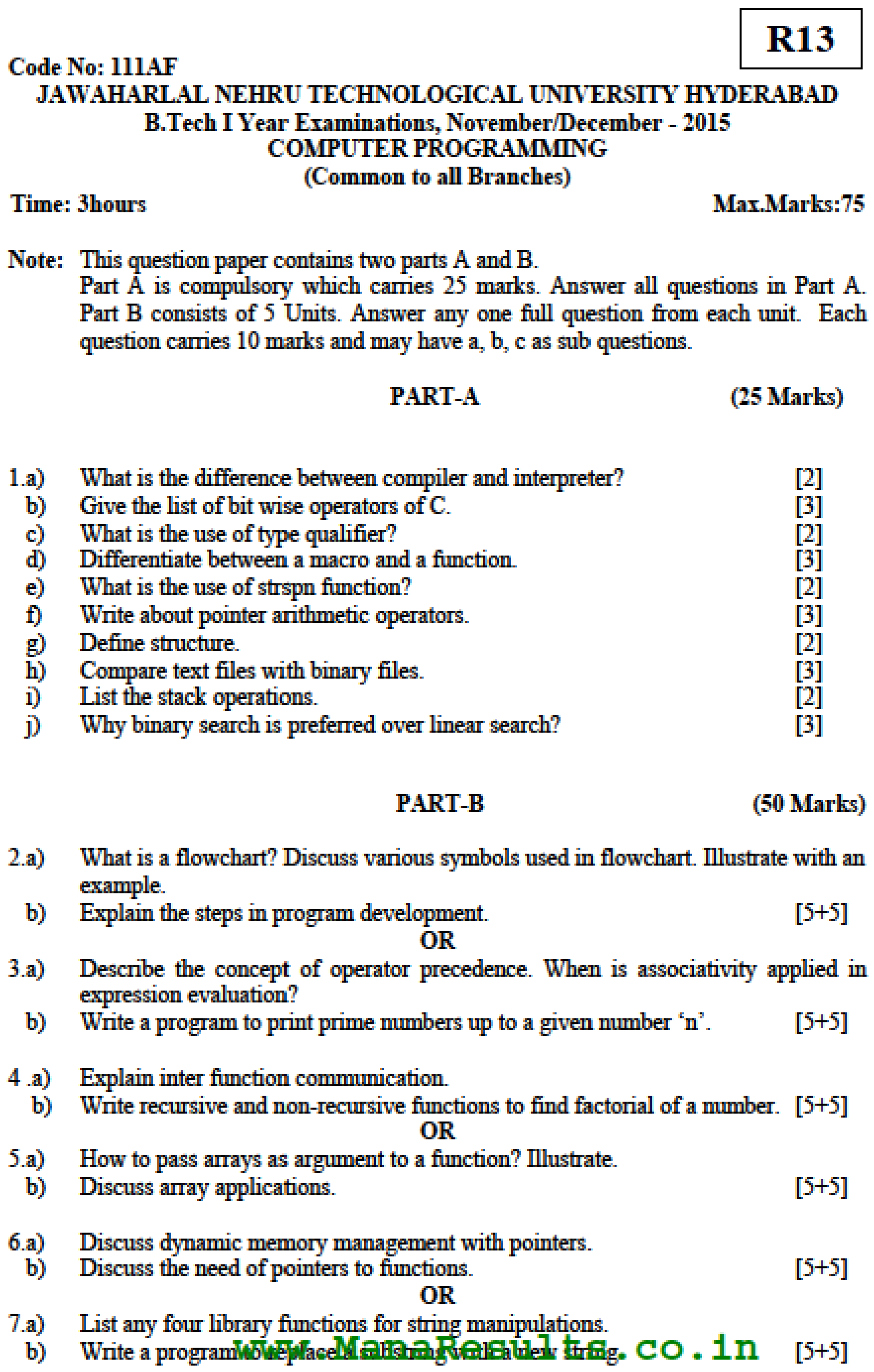
OR

7. Discuss various applications of pointers. State and explain with a sample program various string manipulation functions.

8. Explain about declaration, initialization and accessing of structures. And also discuss about complex structures.

OR

9. What are file streams? Discuss about state of file, opening and closing file with a sample C program.



R13

Code No: 111AF

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B.Tech I Year Examinations, November/December - 2015
COMPUTER PROGRAMMING
(Common to all Branches)

Time: 3hours **Max.Marks:75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART-A **(25 Marks)**

1.a) What is the difference between compiler and interpreter? [2]
b) Give the list of bit wise operators of C. [3]
c) What is the use of type qualifier? [2]
d) Differentiate between a macro and a function. [3]
e) What is the use of strspn function? [2]
f) Write about pointer arithmetic operators. [3]
g) Define structure. [2]
h) Compare text files with binary files. [3]
i) List the stack operations. [2]
j) Why binary search is preferred over linear search? [3]

PART-B **(50 Marks)**

2.a) What is a flowchart? Discuss various symbols used in flowchart. Illustrate with an example.
b) Explain the steps in program development. [5+5]

OR

3.a) Describe the concept of operator precedence. When is associativity applied in expression evaluation?
b) Write a program to print prime numbers up to a given number 'n'. [5+5]

4 .a) Explain inter function communication.
b) Write recursive and non-recursive functions to find factorial of a number. [5+5]

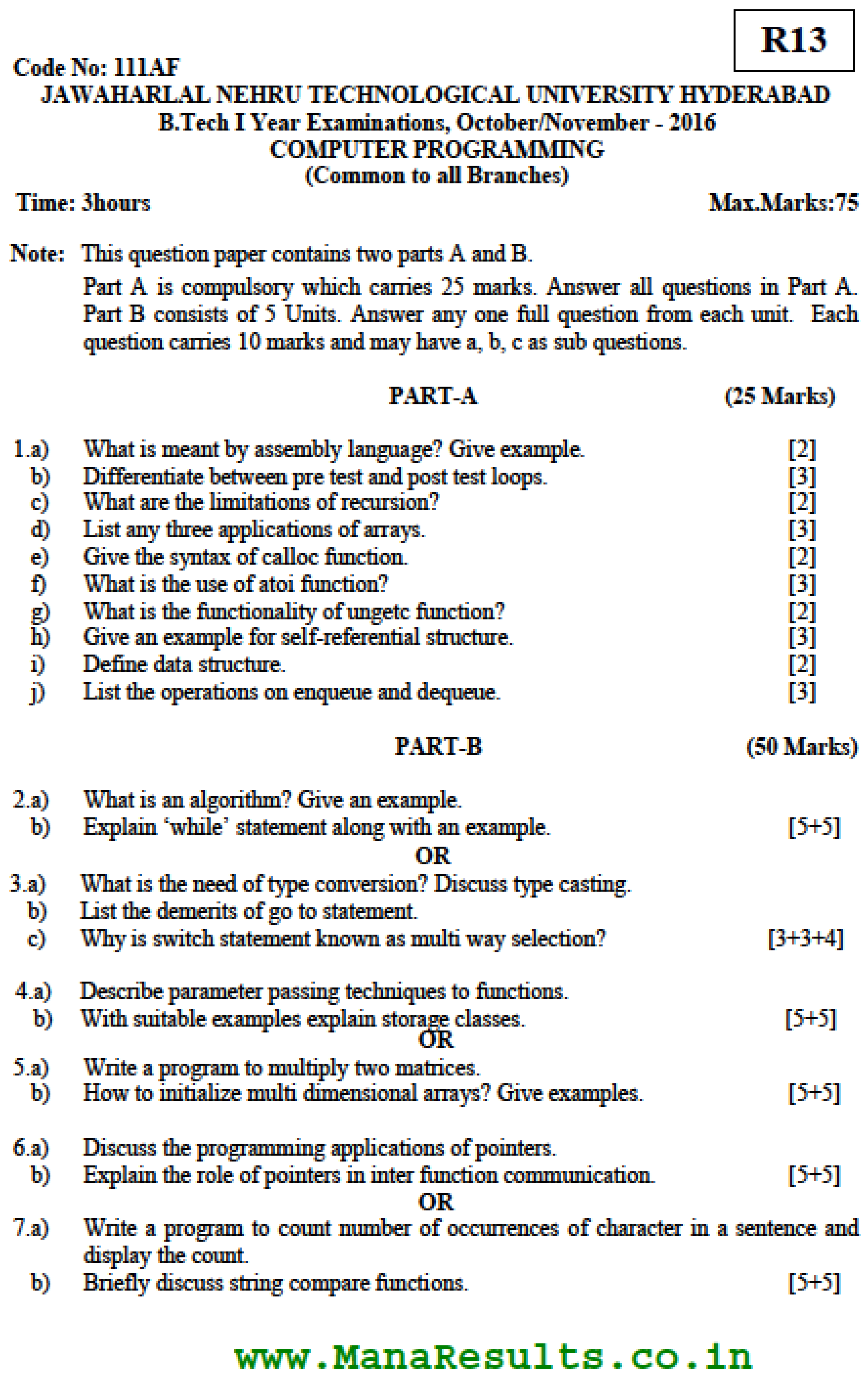
OR

5.a) How to pass arrays as argument to a function? Illustrate.
b) Discuss array applications. [5+5]

6.a) Discuss dynamic memory management with pointers.
b) Discuss the need of pointers to functions. [5+5]

OR

7.a) List any four library functions for string manipulations.
b) Write a program to replace a substring with a new string. [5+5]



R13

Code No: 111AF

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B.Tech I Year Examinations, October/November - 2016
COMPUTER PROGRAMMING
(Common to all Branches)

Time: 3hours **Max.Marks:75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART-A **(25 Marks)**

1.a) What is meant by assembly language? Give example. [2]
b) Differentiate between pre test and post test loops. [3]
c) What are the limitations of recursion? [2]
d) List any three applications of arrays. [3]
e) Give the syntax of calloc function. [2]
f) What is the use of atoi function? [3]
g) What is the functionality of ungetc function? [2]
h) Give an example for self-referential structure. [3]
i) Define data structure. [2]
j) List the operations on enqueue and dequeue. [3]

PART-B **(50 Marks)**

2.a) What is an algorithm? Give an example.
b) Explain 'while' statement along with an example. [5+5]

OR

3.a) What is the need of type conversion? Discuss type casting.
b) List the demerits of go to statement.
c) Why is switch statement known as multi way selection? [3+3+4]

4.a) Describe parameter passing techniques to functions.
b) With suitable examples explain storage classes. [5+5]

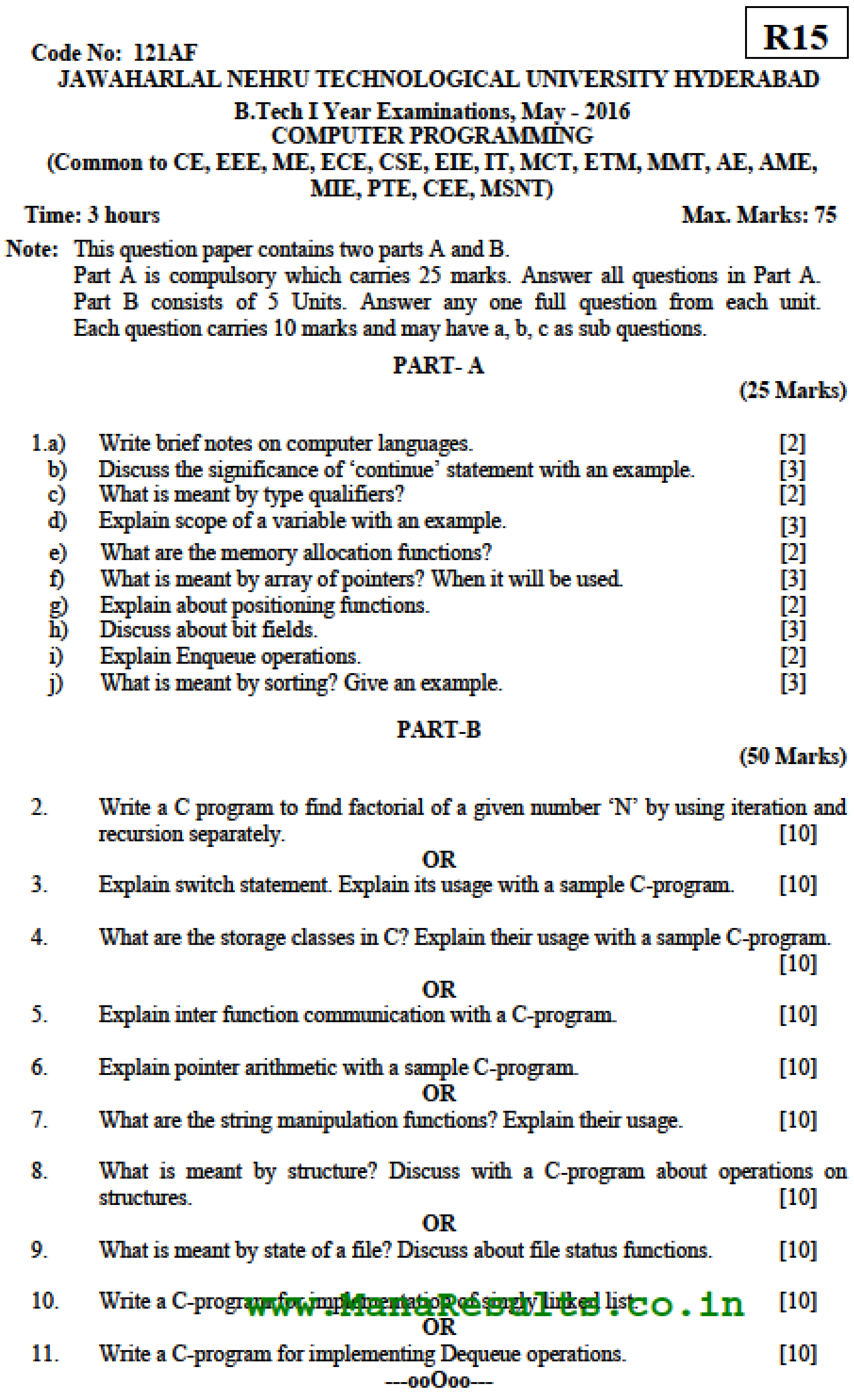
OR

5.a) Write a program to multiply two matrices.
b) How to initialize multi dimensional arrays? Give examples. [5+5]

6.a) Discuss the programming applications of pointers.
b) Explain the role of pointers in inter function communication. [5+5]

OR

7.a) Write a program to count number of occurrences of character in a sentence and display the count.
b) Briefly discuss string compare functions. [5+5]



R15

Code No: 121AF

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

B.Tech I Year Examinations, May - 2016
COMPUTER PROGRAMMING
(Common to CE, EEE, ME, ECE, CSE, EIE, IT, MCT, ETM, MMT, AE, AME, MIE, PTE, CEE, MSNT)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A.
Part B consists of 5 Units. Answer any one full question from each unit.
Each question carries 10 marks and may have a, b, c as sub questions.

PART- A

(25 Marks)

1.a) Write brief notes on computer languages. [2]
b) Discuss the significance of 'continue' statement with an example. [3]
c) What is meant by type qualifiers? [2]
d) Explain scope of a variable with an example. [3]
e) What are the memory allocation functions? [2]
f) What is meant by array of pointers? When it will be used. [3]
g) Explain about positioning functions. [2]
h) Discuss about bit fields. [3]
i) Explain Enqueue operations. [2]
j) What is meant by sorting? Give an example. [3]

PART-B

(50 Marks)

2. Write a C program to find factorial of a given number 'N' by using iteration and recursion separately. [10]

OR

3. Explain switch statement. Explain its usage with a sample C-program. [10]

4. What are the storage classes in C? Explain their usage with a sample C-program. [10]

OR

5. Explain inter function communication with a C-program. [10]

6. Explain pointer arithmetic with a sample C-program. [10]

OR

7. What are the string manipulation functions? Explain their usage. [10]

8. What is meant by structure? Discuss with a C-program about operations on structures. [10]

OR

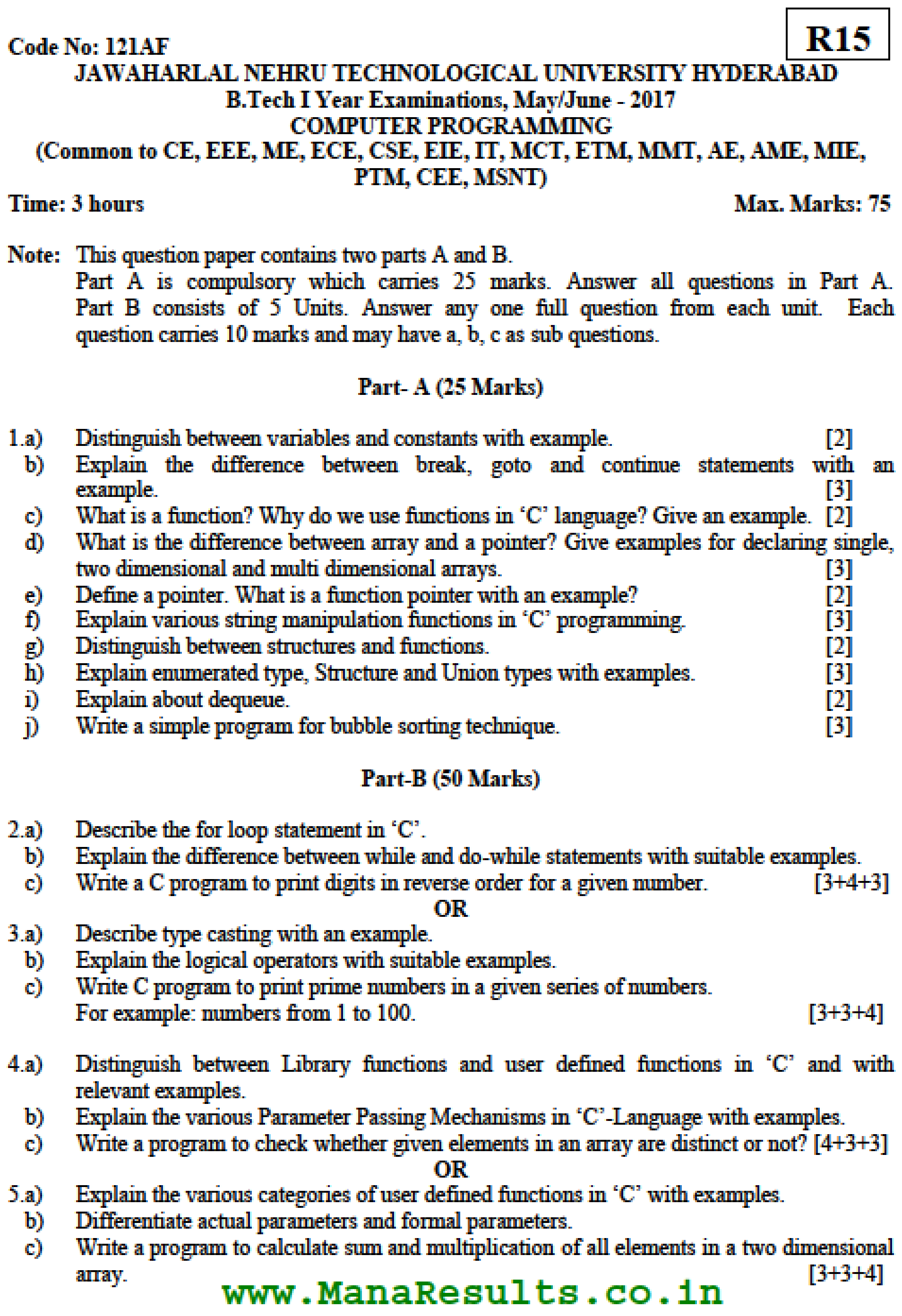
9. What is meant by state of a file? Discuss about file status functions. [10]

10. Write a C-program for implementation of singly linked list. [10]

OR

11. Write a C-program for implementing Dequeue operations. [10]

---ooOoo---



Code No: 121AF **R15**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B.Tech I Year Examinations, May/June - 2017
COMPUTER PROGRAMMING
(Common to CE, EEE, ME, ECE, CSE, EIE, IT, MCT, ETM, MMT, AE, AME, MIE, PTM, CEE, MSNT)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

Part- A (25 Marks)

1.a) Distinguish between variables and constants with example. [2]
b) Explain the difference between break, goto and continue statements with an example. [3]
c) What is a function? Why do we use functions in 'C' language? Give an example. [2]
d) What is the difference between array and a pointer? Give examples for declaring single, two dimensional and multi dimensional arrays. [3]
e) Define a pointer. What is a function pointer with an example? [2]
f) Explain various string manipulation functions in 'C' programming. [3]
g) Distinguish between structures and functions. [2]
h) Explain enumerated type, Structure and Union types with examples. [3]
i) Explain about dequeue. [2]
j) Write a simple program for bubble sorting technique. [3]

Part-B (50 Marks)

2.a) Describe the for loop statement in 'C'.
b) Explain the difference between while and do-while statements with suitable examples.
c) Write a C program to print digits in reverse order for a given number. [3+4+3]

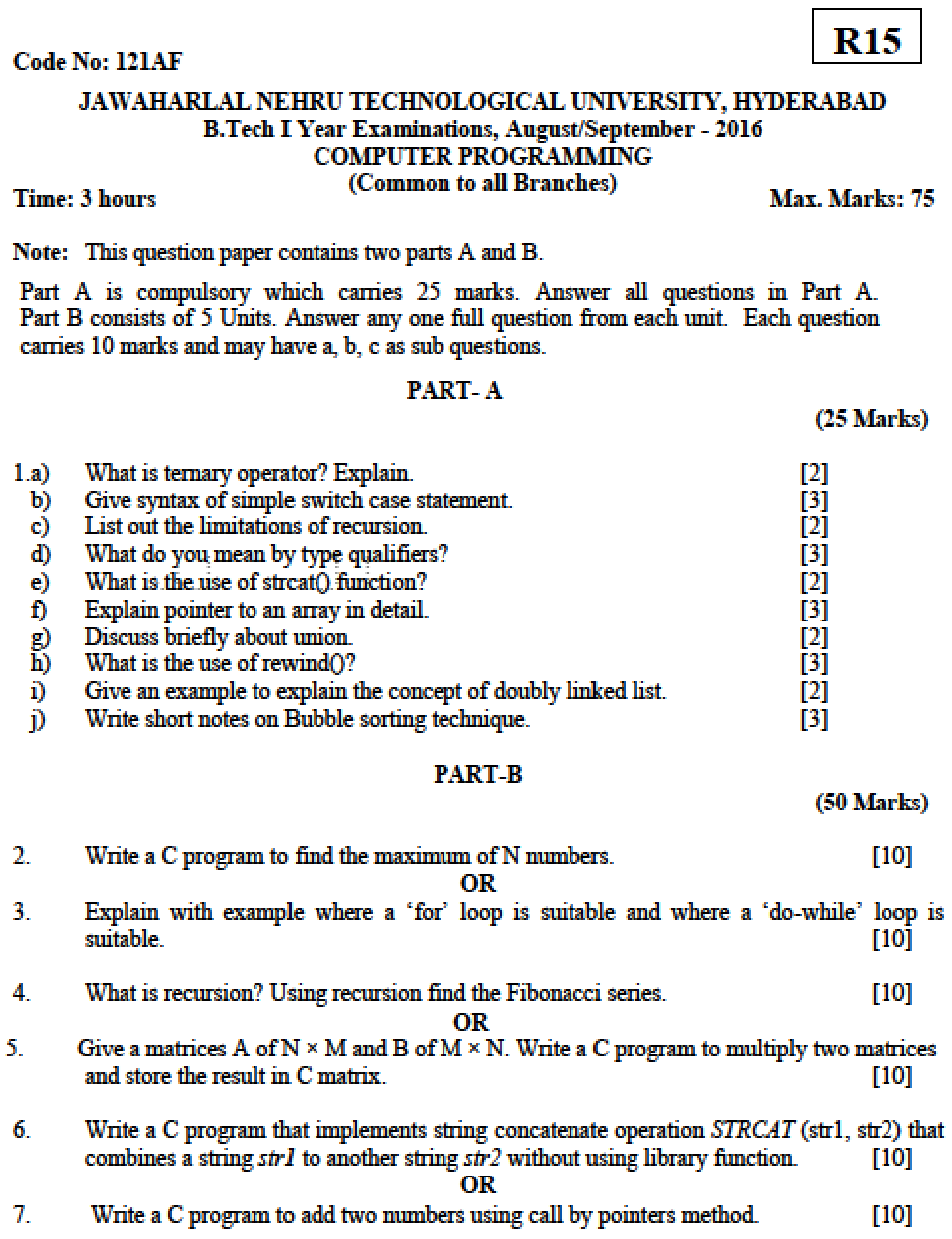
OR

3.a) Describe type casting with an example.
b) Explain the logical operators with suitable examples.
c) Write C program to print prime numbers in a given series of numbers.
For example: numbers from 1 to 100. [3+3+4]

4.a) Distinguish between Library functions and user defined functions in 'C' and with relevant examples.
b) Explain the various Parameter Passing Mechanisms in 'C'-Language with examples.
c) Write a program to check whether given elements in an array are distinct or not? [4+3+3]

OR

5.a) Explain the various categories of user defined functions in 'C' with examples.
b) Differentiate actual parameters and formal parameters.
c) Write a program to calculate sum and multiplication of all elements in a two dimensional array. [3+3+4]



Code No: 121AF **R15**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY, HYDERABAD
B.Tech I Year Examinations, August/September - 2016
COMPUTER PROGRAMMING
(Common to all Branches)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.

Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART- A

(25 Marks)

1.a)	What is ternary operator? Explain.	[2]
b)	Give syntax of simple switch case statement.	[3]
c)	List out the limitations of recursion.	[2]
d)	What do you mean by type qualifiers?	[3]
e)	What is the use of strcat() function?	[2]
f)	Explain pointer to an array in detail.	[3]
g)	Discuss briefly about union.	[2]
h)	What is the use of rewind()?	[3]
i)	Give an example to explain the concept of doubly linked list.	[2]
j)	Write short notes on Bubble sorting technique.	[3]

PART-B

(50 Marks)

2. Write a C program to find the maximum of N numbers. [10]

OR

3. Explain with example where a 'for' loop is suitable and where a 'do-while' loop is suitable. [10]

4. What is recursion? Using recursion find the Fibonacci series. [10]

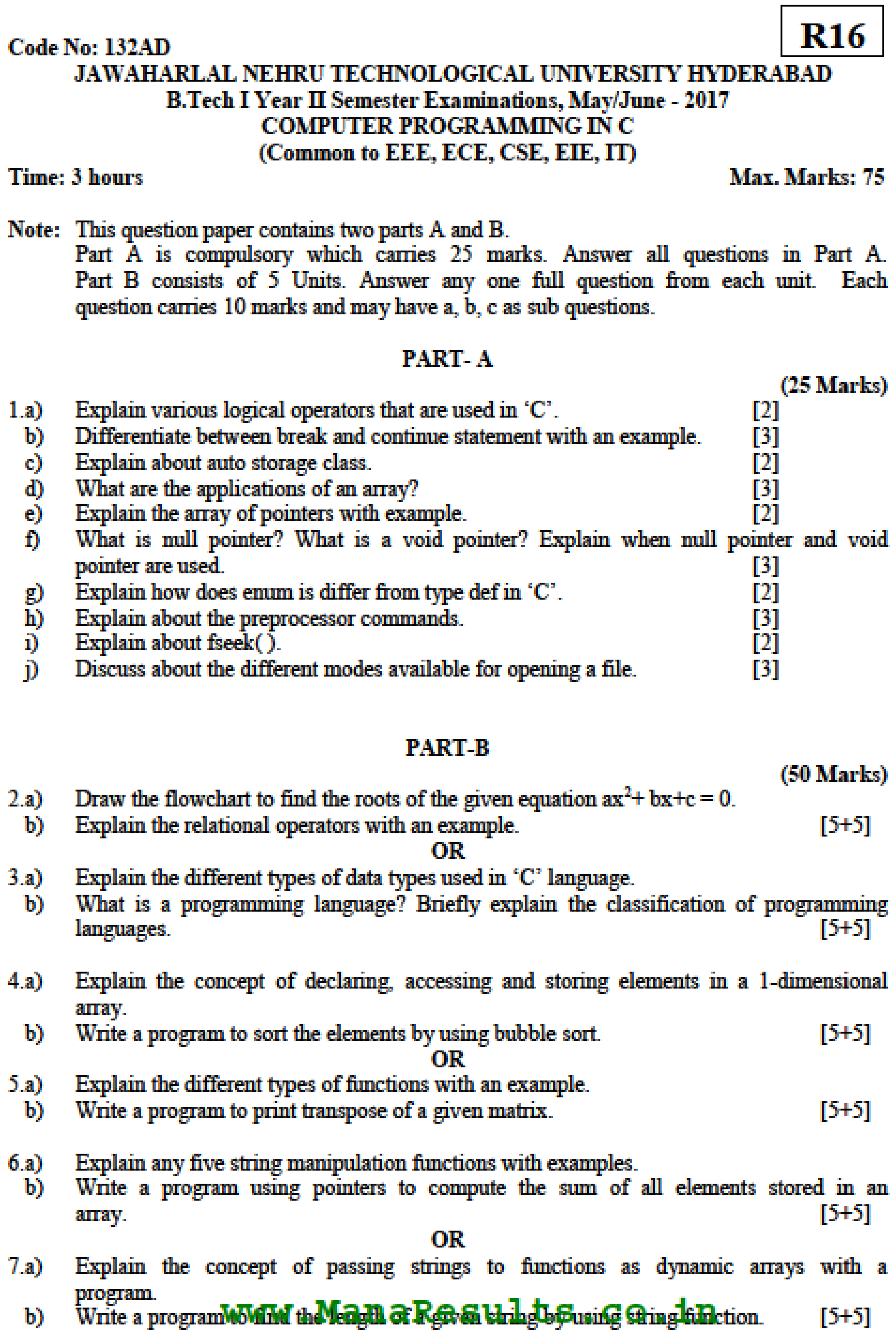
OR

5. Give a matrices A of N × M and B of M × N. Write a C program to multiply two matrices and store the result in C matrix. [10]

6. Write a C program that implements string concatenate operation *STRCAT* (str1, str2) that combines a string *str1* to another string *str2* without using library function. [10]

OR

7. Write a C program to add two numbers using call by pointers method. [10]



Code No: 132AD **R16**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B.Tech I Year II Semester Examinations, May/June - 2017
COMPUTER PROGRAMMING IN C
(Common to EEE, ECE, CSE, EIE, IT)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART- A

(25 Marks)

1.a) Explain various logical operators that are used in 'C'. [2]
b) Differentiate between break and continue statement with an example. [3]
c) Explain about auto storage class. [2]
d) What are the applications of an array? [3]
e) Explain the array of pointers with example. [2]
f) What is null pointer? What is a void pointer? Explain when null pointer and void pointer are used. [3]
g) Explain how does enum is differ from type def in 'C'. [2]
h) Explain about the preprocessor commands. [3]
i) Explain about fseek( ). [2]
j) Discuss about the different modes available for opening a file. [3]

PART-B

(50 Marks)

2.a) Draw the flowchart to find the roots of the given equation $ax^2+bx+c=0$.
b) Explain the relational operators with an example. [5+5]

OR

3.a) Explain the different types of data types used in 'C' language.
b) What is a programming language? Briefly explain the classification of programming languages. [5+5]

4.a) Explain the concept of declaring, accessing and storing elements in a 1-dimensional array.
b) Write a program to sort the elements by using bubble sort. [5+5]

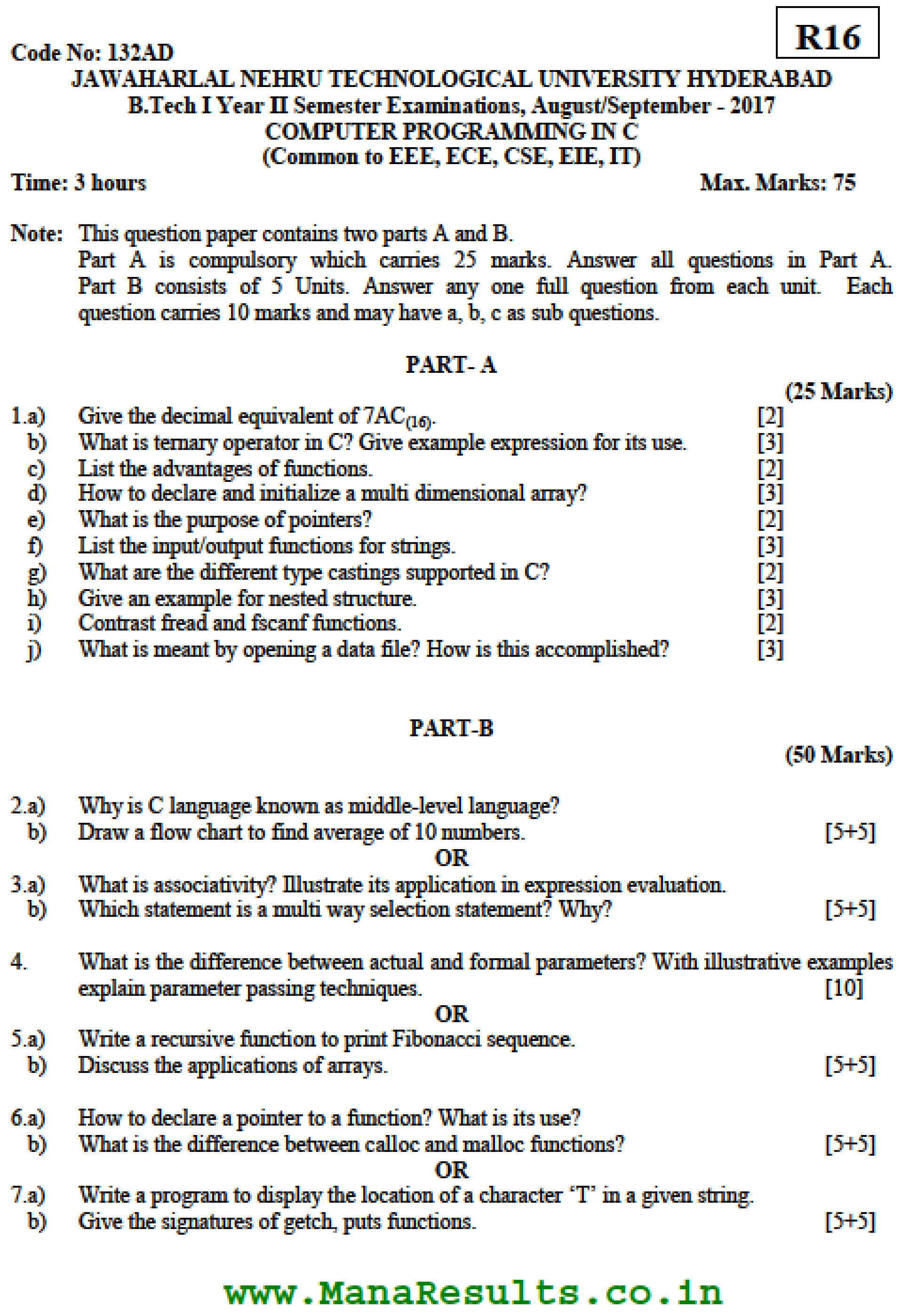
OR

5.a) Explain the different types of functions with an example.
b) Write a program to print transpose of a given matrix. [5+5]

6.a) Explain any five string manipulation functions with examples.
b) Write a program using pointers to compute the sum of all elements stored in an array. [5+5]

OR

7.a) Explain the concept of passing strings to functions as dynamic arrays with a program.
b) Write a program to find the length of a given string by using string function. [5+5]



R16

Code No: 132AD

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B.Tech I Year II Semester Examinations, August/September - 2017
COMPUTER PROGRAMMING IN C
(Common to EEE, ECE, CSE, EIE, IT)

Time: 3 hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART- A

(25 Marks)

1.a) Give the decimal equivalent of $7AC_{(16)}$. [2]
b) What is ternary operator in C? Give example expression for its use. [3]
c) List the advantages of functions. [2]
d) How to declare and initialize a multi dimensional array? [3]
e) What is the purpose of pointers? [2]
f) List the input/output functions for strings. [3]
g) What are the different type castings supported in C? [2]
h) Give an example for nested structure. [3]
i) Contrast fread and fscanf functions. [2]
j) What is meant by opening a data file? How is this accomplished? [3]

PART-B

(50 Marks)

2.a) Why is C language known as middle-level language?
b) Draw a flow chart to find average of 10 numbers. [5+5]

OR

3.a) What is associativity? Illustrate its application in expression evaluation.
b) Which statement is a multi way selection statement? Why? [5+5]

4. What is the difference between actual and formal parameters? With illustrative examples explain parameter passing techniques. [10]

OR

5.a) Write a recursive function to print Fibonacci sequence.
b) Discuss the applications of arrays. [5+5]

6.a) How to declare a pointer to a function? What is its use?
b) What is the difference between calloc and malloc functions? [5+5]

OR

7.a) Write a program to display the location of a character ‘T’ in a given string.
b) Give the signatures of getch, puts functions. [5+5]

REFERENCES

1. Computer Science: A Structured Programming Approach Using C, B. A. Forouzan and R. F. Gilberg, Third Edition, Cengage Learning.

2. Programming in C. P. Dey and M Ghosh , Second Edition, Oxford University Press.

3. The C Programming Language, B.W. Kernighan and Dennis M. Ritchie, Second Edition, Pearson education.

4. Programming with C, B. Gottfried, 3rd edition, Schaum's outlines, McGraw Hill Education (India) Pvt Ltd.

5. C From Theory to Practice, G S. Tselikis and N D. Tselikas, CRC Press.

6. Basic computation and Programming with C, Subrata Saha and S. Mukherjee,